1. Jb0103 Preface to Programming Fundamentals with Java
2. Jb0105: Java OOP: Similarities and Differences between Java and C++
3. Jb0110: Java OOP: Programming Fundamentals, Getting Started
4. Jb0110r Review
5. Jb0115: Java OOP: First Program
6. Jb0120: Java OOP: A Gentle Introduction to Java Programming
7. Jb0120r Review
8. Jb0130: Java OOP: A Gentle Introduction to Methods in Java
9. Jb0130r Review
10. Jb0140: Java OOP: Java comments
11. Jb0140r Review
12. Jb0150: Java OOP: A Gentle Introduction to Java Data Types
13. Jb0150r Review
14. Jb0160: Java OOP: Hello World
15. Jb0160r Review
16. Jb0170: Java OOP: A little more information about classes.
17. Jb0170r: Review
18. Jb0180: Java OOP: The main method.
19. Jb0180r Review
20. Jb0190: Java OOP: Using the System and PrintStream Classes
21. Jb0190r: Review
22. Jb0200: Java OOP: Variables
23. Jb0200r: Review
24. Jb0210: Java OOP: Operators
25. Jb0210r Review
26. Jb0220: Java OOP: Statements and Expressions
27. Jb0220r Review
28. Jb0230: Java OOP: Flow of Control

29. Jb0230r Review
30. Jb0240: Java OOP: Arrays and Strings
31. Jb0240r Review
32. Jb0250: Java OOP: Brief Introduction to Exceptions
33. Jb0260: Java OOP: Command-Line Arguments
34. Jb0260r Review
35. Jb0270: Java OOP: Packages
36. Jb0280: Java OOP: String and StringBuffer
37. Jb0280r Review
38. Jb0290: The end of Programming Fundamentals

Jb0103 Preface to Programming Fundamentals with Java
This Page is a preface to the book titled Programming Fundamentals with Java.

Revised: Sun Mar 27 10:29:44 CDT 2016

Note:
This Page is included in the following Books:

- Programming Fundamentals with Java
- Object-Oriented Programming (OOP) with Java

Table of Contents

Welcome

Welcome to Programming Fundamentals with Java.

This book is a compilation of material that I have published over the years for the benefit of those students who desire to enroll in my beginning OOP course but who don't have the required prerequisite knowledge for that course. If you fall in that category, or if you just want to get a good introduction to computer programming, you may find this material useful.

Even if you have completed a programming fundamentals course in another language, or you have considerable programming experience in another language, you may still find this material useful as an introduction to the Java programming language and its syntax.

In case you decide that you don't need to study the material in this group of modules, you may still find it useful to take a look at the following three modules. These three modules will show you how to configure your computer and get started programming in Java.

- Jb0110: Java OOP: Programming Fundamentals, Getting Started
- Jb0110r Review
- Jb0115: Java OOP: First Program

You may also find it useful to search the web for and study a few tutorials on the Windows "command prompt" as well as a few tutorials on Windows batch files. Here are a couple of possibilities that I found with a rudimentary search:

- Windows Command Prompt in 15 Minutes
- Windows Batch Scripting: Getting Started

If you are using a different operating system, you may need to find similar tutorials that match up with the operating system that you are using.

Most of the topics in this Book are divided into two modules -- a primary module and a review module. The review modules contain review questions and answers keyed to the material in the primary modules.

In addition to the modules contained in this group, you will find several of my other tutorials on programming fundamentals at Obg0510: Programming Fundamentals . Those tutorials are still in their original html format and you may need to go to the Legacy Site to access them fully. They are awaiting conversion to cnxml, which is a requirement for publishing them as modules on cnx.org.

As you work your way through the modules in this group, you should prepare yourself for the more challenging ITSE 2321 OOP tracks identified

below:

- Java 1600: Objects and Encapsulation
- Java 3000: The Guzdial-Ericson Multimedia Class Library
- Java 4010: Getting Started with Java Collections

The DrJava IDE and the Java Development K it

In order to work with the material in this group of *Programming Fundamentals* modules, you will need access to Oracle's Java Development Kit *(JDK)* . You will also need access to a text editor, preferably one that is tailored to the creation of Java programs. One such freely available text editor is named DrJava .

However, DrJava is more than just a text editor. It is an Integrated Development Environment *((IDE)* that is designed for use by students learning how to program in the Java programming language. I recommend it for use with this group of *Programming Fundamentals* modules.

See A Quick Start Guide to DrJava for instructions on downloading and installing both the DrJava IDE and Oracle's Java Development Kit *(JDK)* .

The Quick Start Guide also provides instructions for using the DrJava IDE.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0103 Preface to Programming Fundamentals with Java
- File: Jb0103.htm
- Published: 11/22/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0105: Java OOP: Similarities and Differences between Java and C++
This Page compares Java and C++ for the benefit of persons having familiarity with C++ and making the transition to Java.

Revised: Sun Mar 27 11:30:53 CDT 2016

Note:
This Page is included in the following Books:

- Programming Fundamentals with Java
- Object-Oriented Programming (OOP) with Java

Table of Contents

Preface

This module, which presents some of the similarities and differences between Java and C++, is provided solely for the benefit of those students who are already familiar with C++ and are making the transition from C++ into Java.

If you have some familiarity with C++, you may find the material in this module helpful. If not, simply skip this module and move on to the next module in the collection.

In general, students in Prof. Baldwin's Java/OOP courses are not expected to have any specific knowledge of C++.

This module is intended to be general in nature. Therefore, although a few update notes were added prior to publication at cnx.org, no significant effort has been made to keep it up to date relative to any particular version of the Java JDK or any particular version of C++. Changes have occurred in both Java and C++ since the first publication of this document in 1997. Those changes may not be reflected in this module.

Similarities and differences

This list of similarities and differences is based heavily on The Java Language Environment, A White Paper by James Gosling and Henry McGilton and *Thinking in Java* by Bruce Eckel, which was freely available on the web when this document was first published.

Java does not support **typedefs** , **defines** , or a **preprocessor** . Without a preprocessor, there are no provisions for including header files.

Since Java does not have a preprocessor there is no concept of **#define** macros or *manifest constants* . However, the declaration of named constants is supported in Java through use of the **final** keyword.

Java does not support **enums** but, as mentioned above, does support *named constants* . *(Note: the enum type was introduced into Java sometime between the first publication of this document and Java version 7.)*

Java supports *classes* , but does not support *structures* or *unions* .

All stand-alone C++ programs require a function named **main** and can have numerous other functions, including both stand-alone functions and functions that are members of a class. There are no stand-alone functions in Java. Instead, there are only functions that are members of a class, usually called methods. However, a Java application *(not a Java applet)* does require a class definition containing a **main** method.

Global functions and global data are not allowed in Java. However, variables that are declared **static** are shared among all objects instantiated from the class in which the **static** variables are declared. *(Generally, static has a somewhat different meaning in C++ and Java. For example, the concept of a static local variable does not exist in Java as it does in C++.)*

All classes in Java ultimately inherit from the class named **Object** . This is significantly different from C++ where it is possible to create inheritance trees that are completely unrelated to one another. All Java objects contain the eleven methods that are inherited from the **Object** class.

All function or method definitions in Java are contained within a class definition. To a C++ programmer, they may look like inline function definitions, but they aren't. Java doesn't allow the programmer to request that a function be made inline, at least not directly.

Both C++ and Java support class *(static)* methods or functions that can be called without the requirement to instantiate an object of the class.

The **interface** keyword in Java is used to create the equivalence of an abstract base class containing only method declarations and constants. No variable data members or method definitions are allowed in a Java interface definition. *(True abstract base classes can also be created in Java.)* The interface concept is not supported by C++ but can probably be emulated.

Java does not support multiple class inheritance. To some extent, the **interface** feature provides the desirable features of multiple class inheritance to a Java program without some of the underlying problems.

While Java does not support multiple class inheritance, single inheritance in Java is similar to C++, but the manner in which you implement inheritance differs significantly, especially with respect to the use of constructors in the inheritance chain.

In addition to the access modifiers applied to individual members of a class, C++ allows you to provide an additional access modifier when inheriting from a class. This latter concept is not supported by Java.

Java does not support the **goto** statement *(but goto is a reserved word)* . However, it does support labeled **break** and **continue** statements, a feature not supported by C++. In certain restricted situations, labeled **break** and **continue** statements can be used where a **goto** statement might otherwise be used.

Java does not support **operator overloading** .

Java does not support automatic type conversions *(except where guaranteed safe)* .

Unlike C++, Java has a **String** type, and objects of this type are immutable *(cannot be modified)* . *(Note, although I'm not certain, I believe that the equivalent of a Java String type was introduced into C++ sometime after the original publication of this document.)*

Quoted strings are automatically converted into **String** objects in Java. Java also has a **StringBuffer** type. Objects of this type can be modified, and a variety of string manipulation methods are provided.

Unlike C++, Java provides true arrays as first-class objects. There is a length member, which tells you how big the array is. An exception is thrown if you attempt to access an array out of bounds. All arrays are instantiated in dynamic memory and assignment of one array to another is allowed. However, when you make such an assignment, you simply have two references to the same array. Changing the value of an element in the array using one of the references changes the value insofar as both references are concerned.

Unlike C++, having two "pointers" or references to the same object in dynamic memory is not necessarily a problem *(but it can result in somewhat confusing results)* . In Java, dynamic memory is reclaimed automatically, but is not reclaimed until all references to that memory become NULL or cease to exist. Therefore, unlike in C++, the allocated dynamic memory cannot become invalid for as long as it is being referenced by any reference variable.

Java does not support **pointers** *(at least it does not allow you to modify the address contained in a pointer or to perform pointer arithmetic)* . Much of the need for pointers was eliminated by providing types for arrays and strings. For example, the oft-used C++ declaration **char* ptr** needed to point to the first character in a C++ null-terminated "string" is not required in Java, because a string is a true object in Java.

A class definition in Java looks similar to a class definition in C++, but there is no closing semicolon. Also *forward reference declarations* that are sometimes required in C++ are not required in Java.

The scope resolution operator (::) required in C++ is not used in Java. The dot is used to construct all fully-qualified references. Also, since there are no pointers, the pointer operator (->) used in C++ is not required in Java.

In C++, static data members and functions are called using the name of the class and the name of the static member connected by the scope resolution operator. In Java, the dot is used for this purpose.

Like C++, Java has primitive types such as **int** , **float** , etc. Unlike C++, the size of each primitive type is the same regardless of the platform. There is no unsigned integer type in Java. Type checking and type requirements are much tighter in Java than in C++.

Unlike C++, Java provides a true **boolean** type. *(Note, the C++ equivalent of the Java boolean type may have been introduced into C++ subsequent to the original publication of this document.)*

Conditional expressions in Java must evaluate to **boolean** rather than to integer, as is the case in C++. Statements such as

if(x+y)...

are not allowed in Java because the conditional expression doesn't evaluate to a **boolean** .

The **char** type in C++ is an 8-bit type that maps to the ASCII *(or extended ASCII)* character set. The **char** type in Java is a 16-bit type and uses the

Unicode character set *(the Unicode values from 0 through 127 match the ASCII character set)* . For information on the Unicode character set see http://www.unicode.org/ .

Unlike C++, the >> operator in Java is a "signed" right bit shift, inserting the sign bit into the vacated bit position. Java adds an operator that inserts zeros into the vacated bit positions.

C++ allows the instantiation of variables or objects of all types either at compile time in static memory or at run time using dynamic memory. However, Java requires all variables of primitive types to be instantiated at compile time, and requires all objects to be instantiated in dynamic memory at runtime. Wrapper classes are provided for all primitive types to allow them to be instantiated as objects in dynamic memory at runtime if needed.

C++ requires that classes and functions be declared before they are used. This is not necessary in Java.

The "namespace" issues prevalent in C++ are handled in Java by including everything in a class, and collecting classes into packages.

C++ requires that you re-declare static data members outside the class. This is not required in Java.

In C++, unless you specifically initialize variables of primitive types, they will contain garbage. Although local variables of primitive types can be initialized in the declaration, primitive data members of a class cannot be initialized in the class definition in C++.

In Java, you can initialize primitive data members in the class definition. You can also initialize them in the constructor. If you fail to initialize them, they will be initialized to zero *(or equivalent)* automatically.

Like C++, Java supports constructors that may be overloaded. As in C++, if you fail to provide a constructor, a default constructor will be provided for you. If you provide a constructor, the default constructor is not provided automatically.

All objects in Java are passed by reference, eliminating the need for the copy constructor used in C++.

(In reality, all parameters are passed by value in Java. However, passing a copy of a reference variable makes it possible for code in the receiving method to access the object referred to by the variable, and possibly to modify the contents of that object. However, code in the receiving method cannot cause the original reference variable to refer to a different object.)

There are no destructors in Java. Unused memory is returned to the operating system by way of a *garbage collector* , which runs in a different thread from the main program. This leads to a whole host of subtle and extremely important differences between Java and C++.

Like C++, Java allows you to overload functions *(methods)* . However, default arguments are not supported by Java.

Unlike C++, Java does not support templates. Thus, there are no generic functions or classes. *(Note, generics similar to C++ templates were introduced into Java in version 5 subsequent to the original publication of this document.)*

Unlike C++, several "data structure" classes are contained in the "standard" version of Java. *(Note, the Standard Template Library was introduced into the C++ world subsequent to the original publication of this document.)*

More specifically, several "data structure" classes are contained in the standard class library that is distributed with the Java Development Kit (JDK). For example, the standard version of Java provides the containers **Vector** and **Hashtable** that can be used to contain any object through recognition that any object is an object of type **Object** . However, to use these containers, you must perform the appropriate upcasting and downcasting, which may lead to efficiency problems. *(Note, the upcasting and downcasting requirements were eliminated in conjunction with the introduction of "generics" into Java mentioned earlier.)*

Multithreading is a standard feature of the Java language.

Although Java uses the same keywords as C++ for access control: **private** , **public** , and **protected** , the interpretation of these keywords is significantly different between Java and C++.

There is no **virtual** keyword in Java. All non-static methods use dynamic binding, so the virtual keyword isn't needed for the same purpose that it is used in C++.

Java provides the **final** keyword that can be used to specify that a method cannot be overridden and that it can be statically bound. *(The compiler may elect to make it inline in this case.)*

The detailed implementation of the exception handling system in Java is significantly different from that in C++.

Unlike C++, Java does not support operator overloading. However, the (+) and (+=) operators are automatically overloaded to concatenate strings, and to convert other types to string in the process.

As in C++, Java applications can call functions written in another language. This is commonly referred to as *native methods* . However, applets cannot call native methods.

Unlike C++, Java has built-in support for program documentation. Specially written comments can be automatically stripped out using a separate program named **javadoc** to produce program documentation.

Generally Java is more robust than C++ due to the following:

- Object handles *(references)* are automatically initialized to null.
- Handles are checked before accessing, and exceptions are thrown in the event of problems.
- You cannot access an array out of bounds.
- The potential for memory leaks is prevented *(or at least greatly reduced)* by automatic garbage collection.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0105: Java OOP: Similarities and Differences between Java and C++
- File: Jb0105.htm
- Originally published: 1997
- Published at cnx.org: 11/17/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0110: Java OOP: Programming Fundamentals, Getting Started
This module explains how to get started programming in Java.

> Revised: Sun Mar 27 11:59:05 CDT 2016
>
> **Note:**
> This Page is included in the following Books:
>
> - Programming Fundamentals with Java
> - Object-Oriented Programming (OOP) with Java

Table of Contents

Preface

General

This module is part of a sub-collection of modules designed to help you learn to program computers.

This module explains how to get started programming using the Java programming language.

Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/ or the documentation for the latest version of the JDK.)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.

- An understanding of all of the material covered in the earlier modules in this Book.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

Writing, compiling, and running Java programs

Writing Java code

Writing Java code is straightforward. You can write Java code using any plain text editor. You simply need to cause the output file to have an extension of .java.

There are a number of high-level *Integrated Development Environments (IDEs)* available, such as Eclipse and NetBeans, but they tend to be overkill for the relatively simple Java programs described in these modules.

There are also some low-level IDEs available, such as JCreator and DrJava, which are very useful. I normally use a free version of JCreator, mainly because it contains a color-coded editor.

So, just find an editor that you are happy with and use it to write your Java code.

Preparing to compile and run Java code

Perhaps the most complicated thing is to get your computer set up for compiling and running Java code in the first place.

Downloading the java development kit *(JDK)*

You will need to download and install the free Java JDK from the Oracle/Sun website. As of November, 2012, you will find that website at http://www.oracle.com/technetwork/java/javase/downloads/index.html

There is a 64-bit version of the JDK, which works well on my home computer and on my office computer. However, some older computers may not be able to handle the 64-bit version. A 32-bit version is provided to be used on older computers.

Whether you elect to use the 32-bit or 64-bit version is strictly up to you. Either of them should do the job very nicely.

Installing the JDK

As of November 2012, you will find installation instructions at http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html .

I strongly recommend that you read the instructions and pay particular attention to the information having to do with setting the **path** environment variable.

A word of caution

If you happen to be running Windows Vista or Windows 7, you may need to use something like the following when updating the PATH Environment Variable

```
... ;C:\Program Files (x86)\Java\jdk1.6.0_26\bin
```

in place of

```
... ;C:\Program Files\Java\jdk1.7.0\bin
```

as shown in the installation instructions.

I don't have any experience with any Linux version. Therefore, I don't have any hints to offer there.

The JDK documentation

It is very difficult to program in Java without access to the documentation for the JDK.

Several different types of Java documentation are available online at http://www.oracle.com/technetwork/java/javase/documentation/index.html .

Specific documentation for classes, methods, etc., is available online at http://download.oracle.com/javase/7/docs/api/ .

It is also possible to download the documentation and install it locally if you have room on your disk. The download links for JDK 6 and JDK 7 documentation are also shown on the page at http://www.oracle.com/technetwork/java/javase/downloads/index.html .

You may also want to search for and use the documentation for the latest version of the JDK.

Compiling and running Java code

There are a variety of ways to compile and run Java code. The way that I will describe here is the most basic and, in my opinion, the most reliable. These instructions apply to a Windows operating system. If you are using a different operating system, you will need to translate the instructions to your operating system.

Write your Java program

Begin by using your text editor to write your Java program into one or more text files, each with an extension of .java. *(Files of this type are often referred to as source code files.)* Save the source code files in an empty folder somewhere on your disk. Make sure that the name of the **class** containing the **main** method *(which you will learn about in a future module)* matches the name of the file in which that class is contained *(except for the extension of .java on the file name, which does not appear in the class name)* .

Create a batch file

Use your text editor to create a batch file *(or whatever the equivalent is for your operating system)* containing the text shown in Listing 1 *(with the modifications discussed below)* and store it in the same folder as your Java source code files..

Then execute the batch file, which in turn will execute the program if there are no compilation errors.

Listing 1 . Windows batch file.

Listing 1 . Windows batch file.

```
echo off
cls

del *.class

javac -cp .; hello.java
java -cp .; hello

pause
```

Comments regarding the batch file

The commands in the batch file of Listing 1 will

- Open a command-line screen for the folder containing the batch file.
- Delete all of the compiled class files from the folder. *(If the folder doesn't contain any class files, this will be indicated on the command-line screen.)*
- Attempt to compile the program in the file named **hello.java.**
- Attempt to run the compiled program using a compiled Java file named **hello.class** .
- Pause and wait for you to dismiss the command-line screen by pressing a key on the keyboard.

If errors occur, they will be reported on the command-line screen and the program won't be executed.

If your program is named something other than **hello** , *(which it typically would be)* substitute the new name for the word **hello** where it appears twice in the batch file.

Don't delete the pause command

The **pause** command causes the command-line window to stay on the screen until you dismiss it by pressing a key on the keyboard. You will need to examine the contents of the window if there are errors when you attempt to compile and run your program, so don't delete the pause command.

Translate to other operating systems

The format of the batch file in Listing 1 is a Windows format. If you are using a different operating system, you will need to translate the information in Listing 1 into the correct format for your operating system.

A test program

The test program in Listing 2 can be used to confirm that Java is properly installed on your computer and that you can successfully compile and execute Java programs.

Listing 2 . A test program.

```
class hello {
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//end class
```

Instructions

Copy the code shown in Listing 2 into a text file named **hello.java** and store in an empty folder somewhere on your disk.

Create a batch file named **hello.bat** containing the text shown in Listing 1 and store that file in the same folder as the file named **hello.java** .

Execute the batch file.

If everything is working, a command-line screen should open and display the following text:

```
Hello World
Press any key to continue . . .
```

Congratulations

If that happens, you have just written, compiled and executed your first Java program.

Oops

If that doesn't happen, you need to go back to the installation instructions and see if you can determine why the JDK isn't properly installed.

If you get an error message similar to the following, that probably means that you didn't set the **path** environment variable correctly.

```
'javac' is not recognized as an internal or
external command,
operable program or batch file.
```

Beyond that, I can't provide much advice in the way of troubleshooting hints.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0110: Java OOP: Programming Fundamentals, Getting Started
- File: Jb0110.htm
- Published: 11/16/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0110r Review
This module contains review questions and answers keyed to the module titled Jb0110: Java OOP: Programming Fundamentals, Getting Started.

> Revised: Sun Mar 27 18:44:45 CDT 2016
>
> **Note:**
> This Page is included in the following Books:
>
> - Programming Fundamentals with Java
> - Object-Oriented Programming (OOP) with Java

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0110: Java OOP: Programming Fundamentals, Getting Started .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? You need a special IDE to write Java code.

Answer 1

Question 2

True or false? All of the software that you need to create, compile, and run Java programs is free.

Answer 2

Question 3

True or false? Installing the Java JDK can be a little difficult.

Answer 3

Question 4

True or false? Java is so easy that you don't need documentation to program using Java.

Answer 4

Question 5

True or false? The most fundamental way to compile and run Java applications is from the command line.

Answer 5

Question 6

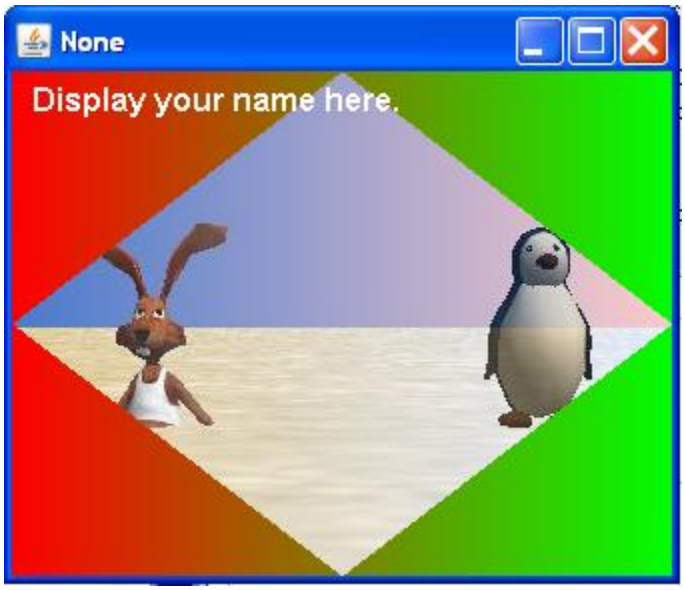
Write a simple test program that can be used to confirm that the JDK is properly installed on your system.

Answer 6

Listings

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



Here is another image that was inserted for the same purpose -- to insert space between the questions and the answers.

Answers

Answer 6

If you can compile and run the program code shown in Listing 1 , the JDK is probably installed properly on your computer.

Listing 1 . A Java test program.

```
class hello {
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//end class
```

Back to Question 6

Answer 5

True. Although a variety of IDEs are available that can be used to compile and run Java applications, the most fundamental way is to compile and run the programs from the command line. A batch file in Windows, or the equivalent in other operating systems, can be of some help in reducing the amount of typing required to compile and run a Java application from the command line.

Back to Question 5

Answer 4

False. Java uses huge class libraries, which few if any of us can memorize. Therefore, it is very difficult to program in Java without access to the documentation for the JDK.

As of November 2012, several different types of Java documentation are available online at http://www.oracle.com/technetwork/java/javase/documentation/index.html .

Back to Question 4

Answer 3

True. Installing the Java JDK can be a little difficult depending on your experience and knowledge. As of November 2012, you will find installation instructions at http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html .

Back to Question 3

Answer 2

True. You will need to download and install the **free** Java JDK from the Oracle/Sun website. As of November, 2012, you will find that website at http://www.oracle.com/technetwork/java/javase/downloads/index.html

Back to Question 2

Answer 1

False. You can write Java code using any plain text editor. You simply need to cause the output file to have an extension of .java.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0110r Review for Programming Fundamentals, Getting Started

- File: Jb0110r.htm
- Published: 11/20/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0115: Java OOP: First Program
The purpose of this module is to present the first complete Java program of the collection that previews the most common forms of the three pillars of procedural programming: sequence, selection, and loop. The program also previews calling a method, passing a parameter to the method, and receiving a returned value from the method.

Revised: Sun Mar 27 18:57:37 CDT 2016

Note:
This Page is included in the following Books:

- Programming Fundamentals with Java
- Object-Oriented Programming (OOP) with Java

Table of Contents

Preface

The purpose of this module is to present the first complete Java program of the collection that previews the most common forms of the three pillars of procedural programming:

- sequence
- selection
- loop

The program also illustrates

- calling a method,
- passing a parameter to the method, and
- receiving a returned value from the method.

As mentioned above, this is simply a preview. Detailed discussions of these topics will be presented in future modules.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

Listings

- Listing 1 . Source code for FirstProgram.

Discussion

Instructions for compiling and running the program

Assuming that the Java Development Kit (JDK) is properly installed on your computer *(see Jb0110: Java OOP: Programming Fundamentals, Getting Started)*, do the following to compile and run this program.

1. Copy the text from Listing 1 into a text file named **FirstProgram.java** and store the file in a folder on your disk.

2. Open a command-line window in the folder containing the file.

3. Type the following command at the prompt to compile the program:

```
javac FirstProgram.java
```

4. Type the following command at the prompt to run the program:

```
java FirstProgram
```

Comments

Any text in the program code that begins with // is a comment. The compiler will ignore everything from the // to the end of the line.

Comments were inserted into the program code to explain the code.

The compiler also ignores blank lines.

Note that this program was designed to illustrate the concepts while being as non-cryptic as possible.

Program output

The program should display the text shown in Figure 1 on the screen except that the time will be different each time you run the program.

Figure 1 . Program output.

```
value in = 5
Odd time = 1353849164875
countA = 0
countA = 1
countA = 2
countB = 0
countB = 1
countB = 2
value out = 10
```

Run the program

I encourage you to copy the code from Listing 1 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0115: Java OOP: First Program
- File: Jb0115.htm
- Published: 11/25/12

Complete program listing

A complete listing of the program follows.

Listing 1 . Source code for FirstProgram.

```
/* Begin block comment
This is the beginning of a block comment in Java.
```

Everything in this block comment is for human consumption
and will be ignored by the Java compiler.

File: FirstProgram.java

This program is designed to illustrate the most common
forms of the three pillars of procedural programming in
Java code:

sequence
selection
loop

The program also illustrates calling a method, passing
a parameter to the method, and receiving a returned
value from the method.

Assuming that the Java Development Kit (JDK) is properly
installed on your computer, do the following to compile
and run this program.

1. Copy this program into a file named FirstProgram.java
and store the file in a folder on your disk.
2. Open a command-line window in the folder containing
the file.
3. Type the following command to compile the program:

```
javac FirstProgram.java
4.4. Type the following command to run the
program:
java FirstProgram

Any text that begins with // in the following
program
code is a comment. The compiler will ignore
everything
from the // to the end of the line.

The compiler also ignores blank lines.

Note that this program was designed to illustrate
the
concepts while being as non-cryptic as possible.

The program should display the following text on
the
screen except that the time will be different each
time
that you run the program.

value in = 5
Odd time = 1353849164875
countA = 0
countA = 1
countA = 2
countB = 0
countB = 1
countB = 2
value out = 10

End block comment
**************************************/

//The actual program begins with the next line.
```

```
import java.util.*;

class FirstProgram{
  //The program consists of a sequence of
statements.

  //The next statement is the beginning of the
main
  // method, which is required in all Java
applications.
  public static void main(String[] args){
    //Program execution begins here.

    //Declare and initialize a variable.
    int var = 5;

    //Statements of the following type display
    // information on the screen
    System.out.println("value in = " + var);

    //Call a method and pass a parameter to the
method.
    //Save the returned value in var, replacing
what
    // was previously stored there.
    //Control is passed to the method named
firstMethod.
    var = firstMethod(var);

    //Control has returned from the method named
    // firstMethod.
    System.out.println("value out = " + var);

    //Program execution ends here
  }//end main method

  /****visual separator
```

```
comment*************************/

  public static int firstMethod(int inData){
    //Control is now in this method.

    //Illustrate selection
    //Get the elapsed time in milliseconds since
Jan 1970.
    long time = new Date().getTime();

    //Select even or odd time and display the
results
    if(time % 2 == 0){
      System.out.println("Even time = " + time);
    }else{
      System.out.println("Odd time = " + time);
    }//end if-else selection

    //Illustrate a while loop
    int countA = 0;
    while(countA < 3){
      System.out.println("countA = " + countA);
      //Increment the counter
      countA = countA + 1;
    }//end while loop

    //Illustrate a for loop
    for(int countB = 0; countB < 3; countB =
countB + 1){
      System.out.println("countB = " + countB);
    }//end for loop

    //Illustrate returning a value from a method
and
    // returning control back to the calling
method.
    return 2*inData;
```

```
  }//end firstMethod

}//end class FirstProgram
//The program ends with the previous line.
```

-end-

Jb0120: Java OOP: A Gentle Introduction to Java Programming
This module provides a gentle introduction to Java programming.

> Revised: Sun Mar 27 19:13:53 CDT 2016
>
> **Note:**
> This Page is included in the following Books:
>
> - Programming Fundamentals with Java
> - Object-Oriented Programming (OOP) with Java

Table of Contents

Preface

General

This module is part of a collection of modules designed to help you learn to program computers.

It provides a gentle introduction to Java programming.

Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

Listings

Discussion and sample code

Introduction

All data is stored in a computer in numeric form. Computer programs do what they do by executing a series of calculations on numeric data. It is the order and the pattern of those calculations that distinguishes one computer program from another.

Avoiding the detailed work

Fortunately, when we program using a high-level programming language such as Java, much of the detailed work is done for us behind the scenes.

Musicians or conductors

As programmers, we are more like conductors than musicians. The various parts of the computer represent the musicians. We tell them what to play, and when to play it, and if we do our job well, we produce a solution to a problem.

Compartments

As the computer program performs its calculations in the correct order, it is often necessary for it to store intermediate results someplace, and then come back and get them to use them in subsequent calculations later. The intermediate results are stored in memory, often referred to as RAM or *Random Access Memory* .

A mechanical analogy

We can think of random access memory as being analogous to a metal rack containing a large number of compartments. The compartments are all the same size and are arranged in a column. Each compartment has a numeric address printed above it. No two compartments have the same numeric address. Each compartment also has a little slot into which you can insert a name or a label for the compartment. No two compartments can have the same name.

Joe, the computer program

Think of yourself as a computer program. You have the ability to write values on little slips of paper and to put them into the compartments. You also have the ability to read the values written on the little slips of paper and to use those values for some purpose. However, there are two rules that you must observe:

- You may not remove a slip of paper from a compartment without replacing it by another slip of paper on which you have written a value.
- You may not put a slip of paper in a compartment without removing the one already there.

Checkout counter example

In understanding how you might behave as a human computer program, consider yourself to have a job working at the checkout counter of a small

grocery store in the 1930s.

You have two tools to work with:

- A mechanical adding machine
- The rack of compartments described above

Initialization

Each morning, the owner of the grocery store tells you to insert a name in the slot above each compartment and to place a little slip of paper with a number written on it inside each compartment. *(In programming jargon, we would refer to this as initialization.)*

Each of the names on the compartments represents a type of grocery such as

- Beans
- Apples
- Pears

No two compartments can have the same name.

No compartment is allowed to have more than one slip of paper inside it.

The price of a can of beans

When you place a new slip of paper in a compartment, you must be careful to remove and destroy the one that was already there.

Each slip of paper that you insert into a compartment contains the price for the type of grocery identified by the label on the compartment.

For example, the slip of paper in the compartment labeled **Beans** may contain the value 15, meaning that each can of beans costs 15 cents.

The checkout procedure

As each customer comes to your checkout counter during the remainder of the day, you execute the following procedure:

- Examine each grocery item to determine its type.
- Read the price stored in the compartment corresponding to that type of grocery.
- Add that price to that customer's bill using your mechanical adding machine.

In programming jargon, we would say that as you process each grocery item for the same customer, you are *looping* . We would also say that you are executing a procedure or an *algorithm* .

When you have processed all of the grocery items for a particular customer, you would

- Press the TOTAL key on the adding machine,
- Turn the crank, and
- Present the customer with the bill.

A schematic representation of the procedure

We might represent the procedure in schematic form as shown in Figure 1 .

Figure 1 . A checkout counter algorithm.

Figure 1 . A checkout counter algorithm.

```
For each customer, do the following:

  For each item, do the following:
    a. Identify the type of grocery item
    b. Get the price from the compartment
    c. Add the price to accumulated total
  End loop on grocery items

  Present customer with a bill

End loop on a specific customer
```

Common programming activities

This procedure illustrates the three activities commonly believed to be the fundamental activities of any computer program:

- sequence
- selection
- loop

Sequence

A sequence of operations is illustrated by the three items labeled a, b, and c in Figure 1 because they are executed in sequential order.

Selection

The process of identifying the type of grocery item is often referred to as *selection* . A selection operation is the process of selecting among two or more choices.

Loop

The process of repetitively examining each grocery item and processing it is commonly referred to as a *loop* . In the early days of programming, for a programming language named FORTRAN, this was referred to as a ***do loop*** .

An algorithm

The entire procedure is often referred to as an *algorithm* .

Modifying stored data

Sometimes during the day, the owner of the grocery store may come to you and say that he is going to increase the price of a can of Beans from 15 cents to 25 cents and asks you to take care of the change in price.

You write 25 on a slip of paper and put it in the compartment labeled Beans, being careful to remove and destroy the slip of paper that was previously in that compartment. For the rest of the day, the new price for Beans will be used in your calculations unless the owner asks you to change it again.

Not a bad analogy

This is a pretty good analogy to how random access memory is actually used by a computer program.

Names versus addresses

As a programmer using a high-level language such as Java, you usually don't have to be concerned about the numeric addresses of the compartments.

You are able to think about them and refer to them in terms of their names. *(Names are easier to remember than numeric addresses).* However, deep inside the computer, these names are cross-referenced to addresses and at the lowest level, the program works with memory addresses instead of names.

Execute an algorithm

A computer program always executes some sort of procedure, which is often called an *algorithm* . The algorithm may be very simple as described in the checkout counter example described above, or it may be very complex as would be the case for a spreadsheet program. As the program executes its algorithm, it uses the random access memory to store and retrieve the data that is needed to execute the algorithm.

Why is it called RAM?

The reason this kind of memory is called *RAM* or *random access memory* is that it can be accessed in any order.

Sequential memory

Some types of memory, such as a magnetic tape, must be accessed in sequential order. This means that to get a piece of data *(the price of beans, for example)* from deep inside the memory, it is necessary to start at the beginning and examine every piece of data until the correct one is found.

Combination random/sequential

Other types of memory, such as disks, provide a combination of sequential and random access. For example, the data on a disk is stored in tracks that form concentric circles on the disk. The tracks can be accessed in random order, but the data within a track must be accessed sequentially starting at a specific point on the track.

Sequential memory isn't very good for use by most computer programs because access to each particular piece of data is quite slow.

Sample program

Listing 1 shows a sample Java program that illustrates the use of memory for the storage and retrieval of data.

Listing 1 . Program named Memory01.

```
//File Memory01.java
class Memory01 {
  public static void main(String[] args){
    int beans;
    beans = 25;
    System.out.println(beans);
  }//end main
}//End Memory01 class
```

Listing 2 shows a batch file that you can use to compile and run this program.

Listing 2 . Batch file for Memory01.

```
echo off
cls

del *.class

javac -cp .; Memory01.java
java -cp .; Memory01

pause
```

Using the procedure that you learned in the *Getting Started* module, you should be able to compile and execute this program. When you do, the program should display 25 on your computer screen.

Variables

You will learn in a future lesson that the term *variable* is synonymous with the term *compartment* that I have used for illustration purposes in this lesson.

The important lines of code

The use of memory is illustrated by the three lines of code in Listing 1 that begin with **int** , **beans** , and **System** . We will ignore the other lines in the program in this module and learn about them in future modules.

Declaring a variable

A memory compartment *(or variable)* is set aside and given the name **beans** by the line that begins with **int** in Listing 1 .

In programmer jargon, this is referred to as *declaring a variable* . The process of declaring a variable

- causes memory to be set aside to contain a value, and
- causes that chunk of memory to be given a name.

That name can be used later to refer to the value stored in that chunk of memory or variable.

This declaration in Listing 1 specifies that any value stored in the variable must be of type **int** . Basically, this means that the value must be an integer. Beyond that, don't worry about what the *type* means at this point. I will explain the concept of type in detail in a future module.

Storing a value in the variable

A value of 25 is stored in the variable named **beans** by the line in Listing 1 that begins with the word **beans** .

In programmer jargon, this is referred to as *assigning a value to a variable* .

From this point forward, when the code in the program refers to this variable by its name, **beans** , the reference to the variable will be interpreted to mean the value stored there.

Retrieving a value from the variable

The line in Listing 1 that begins with the word **System** reads the value stored in the variable named **beans** by referring to the variable by its name.

This line also causes that value to be displayed on your computer screen. However, at this point, you needn't worry about what causes it to be displayed. You will learn those details in a future module. Just remember that the reference to the variable by its name, **beans** , reads the value stored in the variable.

The remaining details

Don't be concerned at this point about the other details in the program. They are there to make it possible for you to compile and execute the program. You will learn about them in future modules.

Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0120: Java OOP: A Gentle Introduction to Java Programming
- File: Jb0120.htm
- Published: 11/16/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0120r Review
This module contains review questions and answers keyed to the module titled Jb0120: Java OOP: A Gentle Introduction to Java Programming.

Revised: Sun Mar 27 19:22:43 CDT 2016

Note:
This Page is included in the following Books:

- Programming Fundamentals with Java
- Object-Oriented Programming (OOP) with Java

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0120: Java OOP: A Gentle Introduction to Java Programming .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? All data is stored in a computer in numeric form. Computer programs do what they do by executing a series of calculations on numeric data. It is the order and the pattern of those calculations that distinguishes one computer program from another.

Answer 1

Question 2

True or false? When we program using Java, we must perform most of the detailed work.

Answer 2

Question 3

True or false? As the computer program performs its calculations in the correct order, it is often necessary for it to store intermediate results someplace, and then come back and get them to use them in subsequent calculations later.

Answer 3

Question 4

True or false? The structured solution to a computer programming problem is often called an algorithm.

Answer 4

Question 5

Which, if any of the following activities is not commonly believed to be fundamental activities of any computer program:

- A. sequence
- B. selection
- C. loop

Answer 5

Question 6

True or false? As a programmer using a high-level language such as Java, you usually don't have to be concerned about the numeric memory addresses of variables.

Answer 6

Question 7

Why is modern computer memory often referred to as RAM?

Answer 7

Question 8

True or false? The process of declaring a variable

- causes memory to be set aside to contain a value, and
- causes that chunk of memory to be given an address.

Answer 8

Question 9

True or false? A value of the type **int** must be an integer.

Answer 9

Question 10

True or false? In programmer jargon, storing a value in a variable is also referred to as assigning a value to a variable.

Answer 10

Question 11

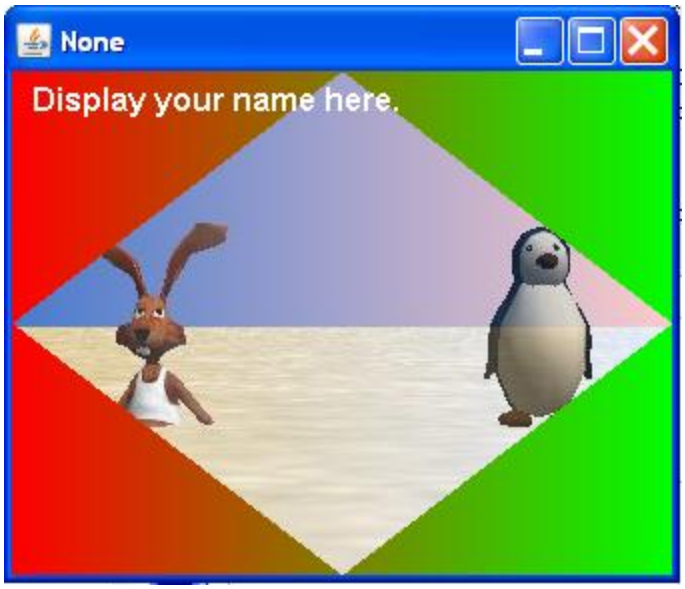
True or false? A reference to a variable name in Java code returns the value stored in the variable.

Answer 11

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 11

True.

Back to Question 11

Answer 10

True.

Back to Question 10

Answer 9

True.

Back to Question 9

Answer 8

False. The process of declaring a variable

- causes memory to be set aside to contain a value, and
- causes that chunk of memory to be given a **name** .

Back to Question 8

Answer 7

Modern computer memory is often called *RAM* or *random access memory* because it can be accessed in any order.

Back to Question 7

Answer 6

True. You are able to think about variables and refer to them in terms of their names. *(Names are easier to remember than numeric addresses).* However, deep inside the computer, these names are cross-referenced to addresses and at the lowest level, the program works with memory addresses instead of names.

Back to Question 6

Answer 5

None. All three are commonly believed to be the fundamental activities of any computer program.

Back to Question 5

Answer 4

True.

Back to Question 4

Answer 3

True.

Back to Question 3

Answer 2

False. Fortunately, when we program using a high-level programming language such as Java, much of the detailed work is done for us behind the scenes.

Back to Question 2

Answer 1

True.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0120r Review for A Gentle Introduction to Java Programming.
- File: Jb0120r.htm
- Published: 12/20/12

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0130: Java OOP: A Gentle Introduction to Methods in Java
This module provides a gentle introduction to Java methods.

> Revised: Sun Mar 27 20:13:16 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

General

This module is part of a collection of modules designed to help you learn to program computers.

It provides a gentle introduction to Java programming methods.

Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

Discussion and sample code

Introduction

Methods have been used in computer programming since the early days of programming. Methods are often called functions, procedures, subroutines, and various other names.

Calculate the square root

Suppose that your program needs to calculate the square root of a number. Referring back to your high-school algebra book, you could refresh your memory on how to calculate a square root. Then you could construct the algorithm describing that process.

Having the algorithm available, you could write the code to calculate the square root and insert it into your program code. Then you could compile, and run your program. If you did it all correctly, your program should calculate the square root. *(For reasons that will become apparent later, I will refer to the code that you inserted as in-line code.)*

Oops, need to do it all over again

Suppose that further on in your program you discover that you need to calculate the square root of another number. And later, you discover that you need to calculate the square root of still another number. Obviously, with a few changes, you could copy your original code and insert it as *in-*

line code at each location in your program where you need to calculate the square root of a number.

Is there a better way?

However, after doing this a few times, you might start asking if there is a better way. The answer is *"yes, there is a better way."*

A method provides a better way

The better way is to create a separate program module that has the ability to calculate the square root and make that module available for use as a helper to your main program each time your main program needs to calculate a square root. In Java, this separate program module is called a **method** .

Standard methods

The Java programming language contains a large number of methods *(in the class libraries)* that are already available for your use. *(Later, I will illustrate the use of a standard method for calculating the square root of a number.)*

In addition to the standard methods that are already available, if you need a method to perform some function and there is no standard method already available to perform that function, you can write your own method.

Passing parameters

Make the method general

Normally, when designing and writing a method such as one that can calculate the square root of a number, it is desirable to write it in such a way that it can calculate the square root of any number *(as opposed to only one specific number)* . This is accomplished through the use of something called *parameters* .

The process of causing a method to be executed is commonly referred to as *calling the method* .

Pass me the number please

When your program calls the square-root method, it will need to tell the method the value for which the square root is needed.

In general, many methods will require that you provide certain kinds of information when you *call* them. The code in the method needs this information to be able to accomplish its purpose.

Passing parameters

This process of providing information to a method when you call it is commonly referred to as *passing parameters* to the method. For the square-root method, you need to pass a parameter whose value is the value of the number for which you need the square root.

Returning values

A method will usually

- perform an action
- send back an answer. or
- some combination of the two

Performing an action

An example of a method that performs an action is the standard method named **println** . We used the **println** method in an earlier module to cause information to be displayed on the computer screen. This method does not need to send back an answer, because that is not the objective of the method. The objective is simply to display some information.

Sending back an answer

On the other hand, a method that is designed to calculate the square root of a number needs to be able to send the square-root value back to the program that called the method. After all, it wouldn't be very useful if the method calculated the square root and then kept it a secret. The process of sending back an answer is commonly referred to as *returning a value* .

Returned values can be ignored

Methods can be designed in such a way that they either will or will not return a value. When a method does return a value, the program that called the method can either pay attention to that value and use it for some purpose, or ignore it entirely.

For example, in some cases where a method performs an action and also returns a value, the calling program may elect to ignore the returned value. On the other hand, if the sole purpose of a method is to return a value, it wouldn't make much sense for a program to call that method and then ignore the value that is returned *(although that would be technically possible)* .

Writing your own methods

As mentioned earlier, you can write your own methods in Java. I mention this here so you will know that it is possible. I will have more to say about writing your own methods in future modules.

Sample program

A complete listing of a sample program named **SqRt01.java** is provided in Listing 5 near the end of the lesson. A batch file that you can use to compile and run the program is provided in Listing 6 .

When you compile and run the program, the following output should appear on your computer screen:

```
5.049752469181039
6.0
```

As you will see shortly, these are the square root values respectively for 25.5 and 36.

Interesting code fragments

I will explain portions of this program in fragments. I will explain only those portions of the program that are germane to this module. Don't worry about the other details of the program. You will learn about those details in future modules.

You may find it useful to open this lesson in another browser window so that you can easily scroll back and forth among the fragments while reading the discussion.

The first code fragment that I will explain is shown in Listing 1 .

Listing 1 . The price of beans.

```
double beans;
beans = 25.5;
```

What is the price of beans?

The code fragment shown in Listing 1 declares a *variable* named **beans** and assigns a value of 25.5 to the variable. *(I briefly discussed the declaration of*

variables in a previous module. I will discuss them in more detail in a future module.)

What is that double thing?

In an earlier module, I declared a variable with a type named **int** . At that time, I explained that only integer values could be stored in that variable.

The variable named **beans** in Listing 1 is declared to be of the type **double** . I will explain the concept of data types in detail in a future module. Briefly, **double** means that you can store any numeric value in this variable, with or without a decimal part. In other words, you can store a value of 3 or a value of 3.33 in this variable, whereas a variable with a declared type of **int** won't accept a value of 3.33.

Every method has a name

Every method, every variable, and some other things as well have names. The names in Java are *case sensitive* . By case sensitive, I mean that the method named **amethod** is not the same as the method named **aMethod** .

A few words about names in Java

There are several rules that define the format of allowable names in Java. You can dig into this in more detail on the web if you like, but if you follow these two rules, you will be okay:

- Use only letters and numbers in Java names.
- Always make the first character a letter.

A standard method named sqrt

Java provides a **Math** library that contains many standard methods. Included in those methods is a method named **sqrt** that will calculate and return the square root of a number that is passed as a parameter when the method is called.

The **sqrt** method is called on the right-hand side of the equal sign (=) in the code fragment in Listing 2 .

Listing 2 . Compute the square root of the price of beans.

```
double sqRtBns = Math.sqrt(beans);
```

Calling the sqrt method

I'm not sure why you would want to do this, but the code fragment in Listing 2

- calls the **sqrt** method and
- passes a copy of the value stored in the **beans** variable as a parameter.

The **sqrt** method calculates and returns the square root of the number that it receives as its incoming parameter. In this case, it returns the square root of the price of a can of beans.

A place to save the square root

I needed some place to save the square root value until I could display it on the computer screen later in the program. I declared another variable named **sqRtBns** in the code fragment in Listing 2 . I also caused the value returned from the **sqrt** method to be stored in, or assigned to, this new variable named **sqRtBns** .

How should we interpret this code fragment?

You can think of the process implemented by the code fragment in Listing 2 as follows.

First note that there is an equal sign (=) near the center of the line of code. *(Later we will learn that this is called the assignment operator.)*

The code on the left-hand side of the assignment operator causes a new chunk of memory to be set aside and named **sqRtBns** . *(We call this chunk of code a variable.)*

The code on the right-hand side of the assignment operator calls the **sqrt** method, passing a copy of the value stored in the **beans** variable to the method.

When the **sqrt** method returns the value that is the square root of its incoming parameter, the assignment operator causes that value to be stored and saved in *(assigned to)* the variable named **sqRtBns** .

Now display the square root value

The code in the fragment in Listing 3 causes the value now stored in **sqRtBns** to be displayed on the computer screen.

Listing 3 . Display the square root value.

```
System.out.println(sqRtBns);
```

Another method is called here

The display of the square root value is accomplished by

- calling another standard method named **println** and
- passing a copy of the value stored in **sqRtBns** as a parameter to the method.

The **println** method performs an action *(displaying something on the computer screen)* and doesn't return a value.

A method exhibits behavior

We say that a method exhibits behavior. The behavior of the **sqrt** method is to calculate and return the square root of the value passed to it as a parameter.

The behavior of the **println** method is to cause its incoming parameter to be displayed on the computer screen.

What do we mean by syntax?

Syntax is a word that is often used in computer programming. The thesaurus in the editor that I am using to type this document says that a synonym for syntax is grammar.

I also like to think of syntax as meaning something very similar to format.

Syntax for passing parameters

Note the syntax in Listing 2 and Listing 3 for passing a parameter to the method. The syntax consists of following the name of the method with a pair of matching parentheses that contain the parameter. If more than one parameter is being passed, they are all included within the parentheses and separated by commas. Usually, the order of the parameters is important if more than one parameter is being passed.

Reusing the methods

The purpose of the code fragment in Listing 4 is to illustrate the reusable nature of methods.

Listing 4 . Calling the same methods again.

Listing 4 . Calling the same methods again.

```
double peas;
peas = 36.;
double sqRtPeas = Math.sqrt(peas);
System.out.println(sqRtPeas);
```

The code in this fragment calls the same **sqrt** method that was called before. In this case, the method is called to calculate the square root of the value stored in the variable named **peas** instead of the value stored in the variable named **beans** .

This fragment saves the value returned from the **sqrt** method in a new variable named **sqRtPeas** . Then the fragment calls the same **println** method as before to display the value now stored in the variable named **sqRtPeas** .

Write once and use over and over

Methods make it possible to write some code once and then use that code many times in the same program. This is the opposite of *in-line code* , which requires you to write essentially the same code multiple times in order to accomplish the same purpose more than once in a program.

Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listings

Listing 5 is a complete listing of the program named **SqRt01** .

Listing 5 . The program named SqRt01.

```
//File SqRt01.java
class SqRt01 {
  public static void main(String[] args){
    double beans;
    beans = 25.5;
    double sqRtBns = Math.sqrt(beans);
    System.out.println(sqRtBns);
    double peas;
    peas = 36.;
    double sqRtPeas = Math.sqrt(peas);
    System.out.println(sqRtPeas);
  }//end main
}//End SqRt01 class
```

Listing 6 contains the commands for a batch file that can be used to compile and run the program named **SqRt01** .

Listing 6 . A batch file for compiling and running the program named SqRt01.

Listing 6 . A batch file for compiling and running the program named SqRt01.

```
echo off
cls

del *.class

javac -cp .; SqRt01.java
java -cp .; SqRt01

pause
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0130: Java OOP: A Gentle Introduction to Methods in Java
- File: Jb0130.htm
- Published: 12/16/12

should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0130r Review
This module contains review questions and answers keyed to the module titled Jb0130: Java OOP: A Gentle Introduction to Methods in Java

> Revised: Sun Mar 27 20:24:28 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0130: Java OOP: A Gentle Introduction to Methods in Java .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? Methods are often called functions, procedures, subroutines, and various other names.

Answer 1

Question 2

True or false? A Java method can be thought of as a separate program module that has the ability to do something useful. Having written the method, you can make it available for use as a helper to your main program each time your main program needs to have that useful thing done.

Answer 2

Question 3

True or false? In Java, you must write all of the methods that you need.

Answer 3

Question 4

True or false? In the following statement, **sqRtPeas** is the name of a method.

System.out.println(sqRtPeas);

Answer 4

Question 5

True or false? Java only allows you to use the pre-written methods in the class libraries.

Answer 5

Question 6

Normally, when designing and writing a method such as one that can calculate the square root of a number, it is desirable to write it in such a way that it can calculate the square root of any number *(as opposed to only one specific number)* . How is that accomplished?

Answer 6

Question 7

True or false? According to common programming jargon, the process of causing a method to be executed is commonly referred to as *setting* the method.

Answer 7

Question 8

True or false? This process of providing information to a method when you call it is commonly referred to as *sending a message* to the method.

Answer 8

Question 9

True or false? When called, a method will usually

- perform an action
- send back an answer. or
- some combination of the two

Answer 9

Question 10

True or false? A value of type **double** can be *(almost)* any numeric value, positive or negative, with or without a decimal part.

Answer 10

Question 11

True or false? Java is not a case-sensitive programming language.

Answer 11

Question 12

True or false? The following two rules will generally suffice to keep you out of trouble when defining variable and method names in Java:

- Use only letters and numbers in Java names.
- Always make the first character a letter.

Answer 12

Question 13

True or false? In Java, the assignment operator is the % character.

Answer 13

Question 14

True or false? The behavior of the **sqrt** method is to calculate and display the square root of the value passed to it as a parameter.

Answer 14

Question 15

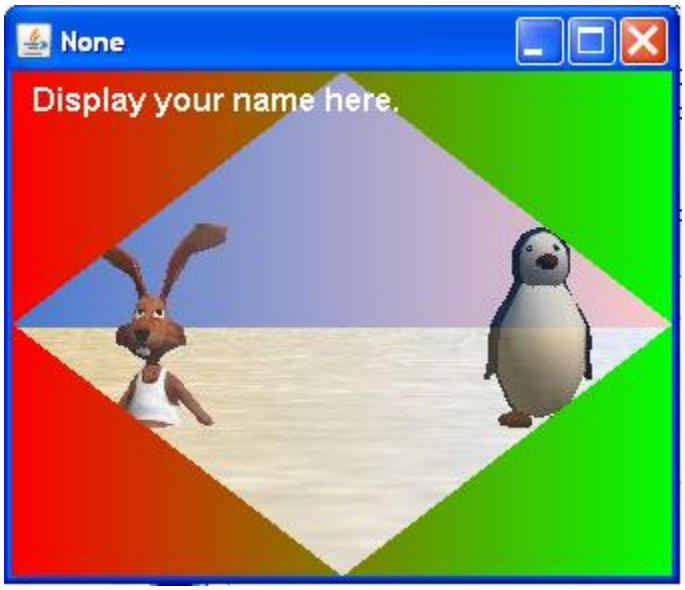
True or false? The syntax for passing parameters to a method consists of following the name of the method with a pair of matching parentheses that contain the parameter or parameters. If more than one parameter is being passed, they are all included within the parentheses and separated by commas. The order of the parameters is not important.

Answer 15

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 15

False. Normally the order in which parameters are passed to a method is very important.

Back to Question 15

Answer 14

False. The behavior of the **sqrt** method is to calculate and ***return*** the square root of the value passed to it as a parameter.

Back to Question 14

Answer 13

False. In Java, the assignment operator is the = character.

Back to Question 13

Answer 12

True.

Back to Question 12

Answer 11

False. Just like C, C++, and C#, Java is very much a case-sensitive programming language.

Back to Question 11

Answer 10

True.

Back to Question 10

Answer 9

True.

Back to Question 9

Answer 8

False. If you continue in this field of study, you will learn that we *send messages* to objects by calling methods that belong to the objects. The process of providing information to a method when you call it is commonly referred to as *passing parameters* to the method.

Back to Question 8

Answer 7

False. The process of causing a method to be executed is commonly referred to as ***calling*** or possibly ***invoking*** the method.

Back to Question 7

Answer 6

That is accomplished through the use of something called *method parameters* .

Back to Question 6

Answer 5

False. In addition to the standard methods that are already available, if you need a method to perform some function and there is no standard method already available to perform that function, you can write your own method.

Back to Question 5

Answer 4

False. In the following statement, **println** is the name of a method. **sqRtPeas** is the name of a variable whose contents are being passed as a parameter to the **println** method.

System.out.println(sqRtPeas);

Back to Question 4

Answer 3

False. The Java programming environment contains a large number of methods *(in the class libraries)* that are already available for you to use when you need them.

Back to Question 3

Answer 2

True.

Back to Question 2

Answer 1

True.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0130r Review: A Gentle Introduction to Methods in Java
- File: Jb0130r.htm
- Published: 12/20/12

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0140: Java OOP: Java comments
This module explains Java comments.

> Revised: Sun Mar 27 20:34:41 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

General

This module is part of a collection of modules designed to help you learn to program computers.

It explains Java comments.

Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

Listings

Discussion and sample code

Comments

Producing and using a Java program consists of the following steps:

1. Write the source code.
2. Compile the source code.
3. Execute the program.

The source code consists of a set of instructions that will later be presented to a special program called a compiler for the purpose of producing a program that can be executed. In other words, when you write the source code, you are writing instructions that the compiler will use to produce the executable program.

Some things should be ignored

Sometimes, when you are writing source code, you would like to include information that may be useful to you, but should be ignored by the compiler. Information of that sort is called a **comment** .

Three styles of comments

Java supports the three styles of comments shown in Figure 1 .

Figure 1 . Three styles of comments.

```
/** special documentation comment
used by the javadoc tool */

/* This is a
multi-line comment */

//Single-line comment
program code // Another single-line comment
```

The javadoc tool

The javadoc tool mentioned in Figure 1 is a special program that is used to produce documentation for Java programs. Comments of this style begin with /** and end with */ as shown in Figure 1 .

The compiler ignores everything in the source code that begins and ends with this pattern of characters. Documentation produced using the javadoc program is very useful for on-line or on-screen documentation.

Multi-line comments

Multi-line comments begin with /* and end with */ as shown in Figure 1 . As you have probably already guessed, the compiler also ignores everything in the source code that matches this format. *(A javadoc comment is simply a multi-line comment insofar as the compiler knows. Only the special program named javadoc.exe cares about javadoc comments.)*

The multi-line comment style is particularly useful for creating large blocks of information that should be ignored by the compiler. This style can be used to produce a comment consisting of a single line of text as well. However, the single-line comment style discussed in the next section requires less typing.

Single-line comments

Single-line comments begin with // and end at the end of the line. The compiler ignores the // and everything following the slash characters to the end of the line.

This style is particularly useful for inserting short comments throughout the source code. In this case, the // can appear at the beginning of the line as shown in Figure 1 , or can appear anywhere in the line, including at the end of some valid source code *(also shown in Figure 1)* .

Sample program

The purpose of the program named **Comments01** , which is shown in Listing 3 near the end of the module, is to illustrate the use of single and multi-line comments. The program does not contain any javadoc comments.

The commands for a batch file that you can use to compile and run this program are provided in Listing 4 .

When you compile and run the program, the following text should appear on your command-line screen:

```
Hello World
```

Interesting code fragments

I will explain this program in fragments, and will explain only those portions of the program that are germane to this module. Don't worry about

the other details of the program at this time. You will learn about those details in future modules.

A multi-line comment

Listing 1 , shows a multi-line comment, which consists of three lines of text.

As required, this multi-line comment begins with /* and ends with */. The extra stars on the third line are simply part of the comment.

You will often see formats similar to this being used to provide a visual separation between multi-line comments and the other parts of a program.

Listing 1 . A multi-line comment.

```
/*File Comments01.java
This is a multi-line comment.
*****************************************/
```

Single-line comments

Listing 2 shows three single-line comments. Can you spot them? Remember, single-line comments begin with //.

Listing 2 . Three single-line comments.

```
class Comments01 {
  //This is a single-line comment
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//End class
```

One of the comments in Listing 2 starts at the beginning of the line. The other two comments follow some program code.

Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listings

Listing 3 contains a complete listing of the program named **Comments01** .

Listing 3 . The program named Comments01.

Listing 3 . The program named Comments01.

```
/*File Comments01.java
This is a multi-line comment.
**************************************/
class Comments01 {
  //This is a single-line comment
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//End class
```

Listing 4 contains the commands for a batch file that can be used to compile and run the program named **Comments01** .

Listing 4 . Batch file to compile and run the program named Comments01.

Listing 4 . Batch file to compile and run the program named Comments01.

```
echo off
cls

del *.class

javac -cp .; Comments01.java
java -cp .; Comments01

pause
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: JJb0140: Java OOP: Java comments
- File: Jb0140.htm
- Published: 11/16/12

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0140r Review
This module contains review questions and answers keyed to the module titled Jb0110: Jb0140: Java OOP: Java comments

> Revised: Sun Mar 27 20:40:28 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0140: Java OOP: Java comments .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? Comments in your source code are ignored by the compiler.

Answer 1

Question 2

True or false? Java supports the four styles of comments.

Answer 2

Question 3

True or false? The **javadoc** tool is a special program that is used to compile Java programs.

Answer 3

Question 4

True or false? Comments recognized by the **javadoc** tool begin with /** and end with */

Answer 4

Question 5

True or false? Multi-line comments begin with /# and end with #/

Answer 5

Question 6

True or false? The multi-line comment style is particularly useful for creating large blocks of information that should be ignored by the compiler.

Answer 6

Question 7

True or false? The multi-line comment style cannot be used to produce a comment consisting of a single line of text.

Answer 7

Question 8

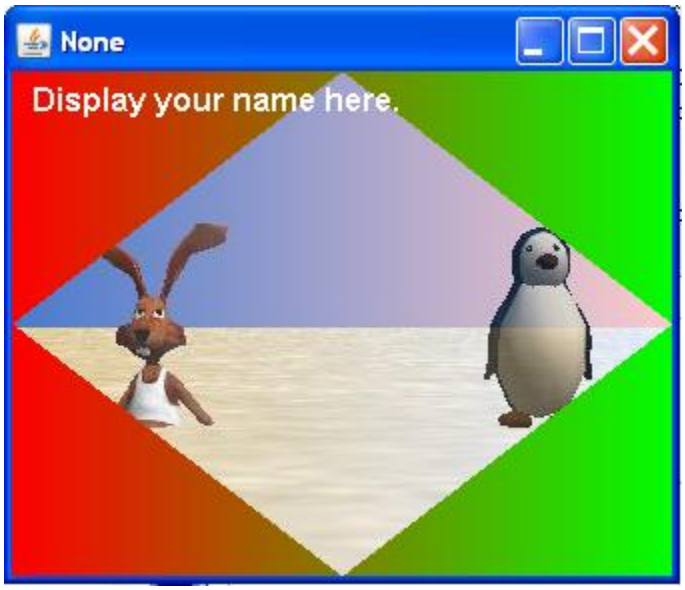
True or false? Single-line comments begin with // and end at the end of the line.

Answer 8

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 8

True.

Back to Question 8

Answer 7

False. The multi-line comment style can be used to produce a comment consisting of none, one, or more lines of text.

Back to Question 7

Answer 6

True.

Back to Question 6

Answer 5

False. Multi-line comments begin with /* and end with */

Back to Question 5

Answer 4

True.

Back to Question 4

Answer 3

False. The **javadoc** tool is a special program that is used to produce documentation for Java program.

Back to Question 3

Answer 2

False. Java supports the three styles of comments.

Back to Question 2

Answer 1

True.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0140r Review: Java comments
- File: Jb0140r.htm
- Published: 11/21/12

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0150: Java OOP: A Gentle Introduction to Java Data Types
This module introduces Java data types.

> Revised: Sun Mar 27 21:20:50 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

General

This module is part of a collection of modules designed to help you learn to program computers.

It introduces Java data types.

Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

Figures

Discussion

Introduction

Type-sensitive languages

Java and some other modern programming languages make heavy use of a concept that we refer to as *type* , or *data type* .

We refer to those languages as *type-sensitive languages* . Not all languages are type-sensitive languages. In particular, some languages hide the concept of type from the programmer and automatically deal with type issues behind the scenes.

So, what do we mean by type?

One analogy that comes to my mind is international currency. For example, many years ago, I spent a little time in Japan and quite a long time on an island named Okinawa *(Okinawa is now part of Japan)* .

Types of currency

At that time, as now, the type of currency used in the United States was the dollar. The type of currency used in Japan was the yen, and the type of currency used on the island of Okinawa was also the yen. However, even

though two of the currencies had the same name, they were different types of currency, as determined by the value relationships among them.

The exchange rate

As I recall, at that time, the exchange rate between the Japanese yen and the U.S. dollar was 360 yen for each dollar. The exchange rate between the Okinawan yen and the U.S. dollar was 120 yen for each dollar. This suggests that the exchange rate between the Japanese yen and the Okinawan yen would have been 3 Japanese yen for each Okinawan yen.

Analogous to different types of data

So, why am I telling you this? I am telling you this to illustrate the concept that different types of currency are roughly analogous to different data types in programming.

Purchasing transactions were type sensitive

In particular, because there were three different types of currency involved, the differences in the types had to be taken into account in any purchasing transaction to determine the price in that particular currency. In other words, the purchasing process was sensitive to the type of currency being used for the purchase *(type sensitive)* .

Different types of data

Type-sensitive programming languages deal with different types of data. Some data types such at type **int** involve whole numbers only *(no fractional parts are allowed)* .

Other data types such as **double** involve numbers with fractional parts.

Some data types conceptually have nothing to do with numeric values, but deal only with the concept of true or false *(**boolean**)* or with the concept of the letters of the alphabet and the punctuation characters *(**char**)* .

Type specification

For every different type of data used with a particular programming language, there is a specification somewhere that defines two important characteristics of the type:

1. What is the set of all possible data values that can be stored in an instance of the type *(we will learn some other names for instance later)* ?
2. Once you have an instance of the type, what are the operations that you can perform on that instance alone, or in combination with other instances?

What do I mean by an instance of a type?

Think of the type specification as being analogous to the plan or blueprint for a model airplane. Assume that you build three model airplanes from the same set of plans. You will have created three *instances* of the plans.

We might say that an *instance* is the physical manifestation of a plan or a type.

Using mixed types

Somewhat secondary to the specifications for the different types, but also extremely important, is a set of rules that define what happens when you perform an operation involving mixed types *(such as making a purchase using some yen currency in combination with some dollar currency)* .

The short data type

For example, in addition to the integer type **int** , there is a data type in Java known as **short** . The **short** type is also an integer type.

If you have an instance of the **short** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -32,768 to +32,767.

This constitutes a set of 65,536 different values, including the value zero. No other value can be stored in an instance of the type **short** . For example,

you cannot store the value 35,000 in an instance of the type **short** in Java. If you need to store that value, you will need to use some type other than **short** .

Kind of like an odometer

This is somewhat analogous to the odometer in your car *(the thing that records how many miles the car has been driven)* . For example, depending on the make and model of car, there is a specified set of values that can appear in the odometer. The value that appears in the odometer depends on how many miles your car has been driven.

It is fairly common for an odometer to be able to store and to display the set of all positive values ranging from zero to 99999. If your odometer is designed to store that set of values and if you drive your car more than 99999 miles, it is likely that the odometer will roll over and start back at zero after you pass the 99999-mile mark. In other words, that particular odometer does not have the ability to store a value of 100,000 miles. Once you pass the 99999-mark, the data stored in the odometer is corrupt.

Now let's return to the Java type named short

Assume that you have two instances of the type **short** in a Java program. What are the operations that you can perform on those instances? For example:

- You can add them together.
- You can subtract one from the other.
- You can multiply one by the other.
- You can divide one by the other.
- You can compare one with the other to determine which is algebraically larger.

There are some other operations that are allowed as well. In fact, there is a well-defined set of operations that you are allowed to perform on those instances. That set of operations is defined in the specification for the type **short** .

What if you want to do something different?

However, if you want to perform an operation that is not allowed by the type specification, then you will have to find another way to accomplish that purpose.

For example, some programming languages allow you to raise whole-number types to a power *(examples: four squared, six cubed, nine to the fourth power, etc.)* . However, that operation is not allowed by the Java specification for the type **short** . If you need to do that operation with a data value of the Java **short** type, you must find another way to do it.

Two major categories of type

Java data types can be subdivided into two major categories:

- Primitive types
- User-defined or reference types

These categories are discussed in more detail in the following sections.

Primitive types

Java is an extensible programming language

What this means is that there is a core component to the language that is always available. Beyond this, individual programmers can extend the language to provide new capabilities. The primitive types discussed in this section are the types that are part of the core language. A later section will discuss user-defined types that become available when a programmer extends the language.

More subdivision

It seems that when teaching programming, I constantly find myself subdividing topics into sub-topics. I am going to subdivide the topic of Primitive Types into four categories:

- Whole-number types
- Floating-point types
- Character types
- Boolean types

Hopefully this categorization will make it possible for me to explain these types in a way that is easier for you to understand.

Whole-number types

The whole-number types, often called *integer* types, are relatively easy to understand. These are types that can be used to represent data without fractional parts.

Applesauce and hamburger

For example, consider purchasing applesauce and hamburger. At the grocery store where I shop, I am allowed to purchase cans of applesauce only in whole-number or integer quantities.

Can purchase integer quantities only

For example, the grocer is happy to sell me one can of applesauce and is even happier to sell me 36 cans of applesauce. However, she would be very unhappy if I were to open a can of applesauce in the store and attempt to purchase 6.3 cans of applesauce.

Counting doesn't require fractional parts

A count of the number of cans of applesauce that I purchase is somewhat analogous to the concept of whole-number data types in Java. Applesauce is not available in fractional parts of cans *(at my grocery store)* .

Fractional pounds of hamburger are available

On the other hand, the grocer is perfectly willing to sell me 6.3 pounds of hamburger. This is somewhat analogous to *floating-point data types* in Java.

Accommodating applesauce and hamburger in a program

Therefore, if I were writing a program dealing with quantities of applesauce and hamburger, I might elect to use a whole number type to represent cans of applesauce and to use a floating-point type to represent pounds of hamburger.

Different whole-number types

In Java, there are four different whole-number types:

- byte
- short
- int
- long

(The char type is also a whole number type, but since it is not intended to be used for arithmetic, I discuss it later as a character type.)

The four types differ primarily in terms of the range of values that they can accommodate and the amount of computer memory required to store instances of the types.

Differences in operations?

Although there are some subtle differences among the four whole-number types in terms of the operations that you can perform on them, I will defer a discussion of those differences until a more advanced module. *(For example some operations require instances of the **byte** and **short** types to be converted to type **int** before the operation takes place.)*

Algebraically signed values

All four of these types can be used to represent algebraically signed values ranging from a specific negative value to a specific positive value.

Range of the byte type

For example, the **byte** type can be used to represent the set of whole numbers ranging from -128 to +127 inclusive. *(This constitutes a set of 256 different values, including the value zero.)*

The **byte** type cannot be used to represent any value outside this range. For example, the **byte** type cannot be used to represent either -129 or +128.

No fractional parts allowed by the byte type

Also, the **byte** type cannot be used to represent fractional values within the allowable range. For example, the byte type cannot be used to represent the value of 63.5 or any other value that has a fractional part.

Like a strange odometer

To form a crude analogy, the byte type is sort of like a strange odometer in a new *(and unusual)* car that shows a mileage value of -128 when you first purchase the car. As you drive the car, the negative values shown on the odometer increment toward zero and then pass zero. Beyond that point they increment up toward the value of +127.

Oops, numeric overflow!

When the value passes *(or attempts to pass)* +127 miles, something bad happens. From that point forward, the value shown on the odometer is not a reliable indicator of the number of miles that the car has been driven.

Ranges for each of the whole-number types

Figure 1 shows the range of values that can be accommodated by each of the four whole-number types supported by the Java programming language:

Figure 1 . Range of values for whole-number types.

Figure 1 . Range of values for whole-number types.

```
byte
-128 to +127

short
-32768 to +32767

int
-2147483648 to +2147483647

long
-9223372036854775808 to +9223372036854775807
```

Can represent some fairly large values

As you can see, the **int** and **long** types can represent some fairly large values. However, if your task involves calculations such as distances in interstellar space, these ranges probably won't accommodate your needs. This will lead you to consider using the *floating-point* types discussed in the upcoming sections. I will discuss the operations that can be performed on whole-number types more fully in future modules.

Floating-point types

Floating-point types are a little more complicated than whole-number types. I found the definition of floating-point shown in Figure 2 in the *Free On-Line Dictionary of Computing* at this URL .

Figure 2 . Definition of floating point.

A number representation consisting of a mantissa, M, an exponent, E, and an (assumed) radix *(or "base")* . The number represented is M*R^E where R is the radix - usually ten but sometimes 2.

So what does this really mean?

Assuming a base or radix of 10, I will attempt to explain it using an example.

Consider the following value:

623.57185

I can represent this value in any of the ways shown in Figure 3 *(where * indicates multiplication).*

Figure 3 . Different ways to represent 623.57185.

Figure 3 . Different ways to represent 623.57185.

```
.62357185*1000
6.2357185*100
62.357185*10
623.57185*1
6235.7185*0.1
62357.185*0.01
623571.85*0.001
6235718.5*0.0001
62357185.*0.00001
```

In other words, I can represent the value as a mantissa (62357185) multiplied by a factor where the purpose of the factor is to represent a left or right shift in the position of the decimal point.

Now consider the factor

Each of the factors shown in Figure 3 represents the value of ten raised to some specific power, such as ten squared, ten cubed, ten raised to the fourth power, etc.

Exponentiation

If we allow the following symbol (^) to represent exponentiation *(raising to a power)* and allow the following symbol (/) to represent division, then we can write the values for the above factors in the ways shown in Figure 4 .

Note in particular the characters following the first equal character (=) on each line, which I will refer to later as the exponents.

Figure 4 . Relationships between multiplicative factors and exponentiation.

```
1000 = 10^+3 = 1*10*10*10
100 = 10^+2 = 1*10*10
10 = 10^+1 = 1*10
1 = 10^+0 = 1
0.1 = 10^-1 = 1/10
0.01 = 10^-2 = 1/(10*10)
0.001 = 10^-3 = 1/(10*10*10)
0.0001 = 10^-4 = 1/(10*10*10*10)
0.00001 = 10^-5 = 1/(10*10*10*10*10)
```

In the above notation, the term 10^+3 means 10 raised to the third power.

The zeroth power

By definition, the value of any value raised to the zeroth power is 1. *(Check this out in your high-school algebra book.)*

The exponent and the factor

Hopefully, at this point you will understand the relationship between the exponent and the factor introduced earlier in Figure 3 .

Different ways to represent the same value

Having reached this point, by using substitution, I can rewrite the original set of representations of the value 623.57185 in the ways shown in Figure 5 .

(It is very important to for you to understand that these are simply different ways to represent the same value.)

Figure 5 . Other ways to represent the same information.

```
.62357185*10^+3
6.2357185*10^+2
62.357185*10^+1
623.57185*10^+0
6235.7185*10^-1
62357.185*10^-2
623571.85*10^-3
6235718.5*10^-4
62357185.*10^-5
```

A simple change in notation

Finally, by making a simplifying change in notation where I replace (*10^) by (E) I can rewrite the different representations of the value of 623.57185 in the ways shown in Figure 6 .

Figure 6 . Still other ways to represent 623.57185.

Figure 6 . Still other ways to represent 623.57185.

```
.62357185E+3
6.2357185E+2
62.357185E+1
623.57185E+0
6235.7185E-1
62357.185E-2
623571.85E-3
6235718.5E-4
62357185.E-5
```

Getting the true value

Floating point types represent values as a mantissa containing a decimal point along with an exponent value which tells how many places to shift the decimal point to the left or to the right in order to determine the true value.

Positive exponent values mean that the decimal point should be shifted to the right. Negative exponent values mean that the decimal point should be shifted to the left.

Maintaining fractional parts

One advantage of floating-point types is that they can be used to maintain fractional parts in data values, such as 6.3 pounds of hamburger.

Accommodating a very large range of values

Another advantage is that a very large range of values can be represented using a reasonably small amount of computer memory for storage of the values.

Another example

For example *(assuming that I counted the number of digits correctly)* the following very large value

```
623571850000000000000000000000000000.0
```

can be represented as

```
6.2357185E+37
```

Similarly, again assuming that I counted the digits correctly, the following very small value

```
0.000000000000000000000000000062357185
```

can be represented as

```
6.2357185E-30
```

When would you use floating-point?

If you happen to be working in an area where you

- need to keep track of fractional parts *(such as the amount of hamburger in a package)* ,
- have to work with extremely large numbers *(distances between galaxies)* , or
- have to work with extremely small values *(the size of atomic particles)* ,

then you will need to use the floating-point types.

Don't use floating-point in financial transactions

You probably don't want to use floating-point in financial calculations, however, because there is a lot of rounding that takes place in floating-point calculations. In other words, floating point calculations provide answers that are very close to the truth but the answers are often not exact.

Two floating-point types

Java supports two different floating point types:

- float
- double

These two types differ primarily in terms of the range of values that they can support.

Range of values for floating point types

The table in **Figure 7** shows the smallest and largest values that can be accommodated by each of the floating-point types. Values of either type can be either positive or negative.

Figure 7 . Range of values for floating-point types.

```
float
1.4E-45 to 3.4028235E38

double
4.9E-324 to 1.79769313486231570E308
```

I will discuss the operations that can be performed on floating-point types in a future module.

The character type

Computers deal only in numeric values. They don't know how to deal directly with the letters of the alphabet and punctuation characters. This

gives rise to a type named **char** .

Purpose of the char type

The purpose of the character type is to make it possible to represent the letters of the alphabet, the punctuation characters, and the numeric characters internally in the computer. This is accomplished by assigning a numeric value to each character, much as you may have done to create secret codes when you were a child.

A single character type

Java supports a single character type named **char** . The char type uses a standard character representation known as **Unicode** to represent up to 65,535 different characters.

Why so many characters?

The reason for the large number of possible characters is to make it possible to represent the characters making up the alphabets of many different countries and many different spoken languages.

What are the numeric values representing characters?

As long as the characters that you use in your program appear on your keyboard, you usually don't have a need to know the numeric value associated with the different characters. If you are curious, however, the upper-case A is represented by the value 65 in the Unicode character set.

Representing a character symbolically

In Java, you usually represent a character in your program by surrounding it with apostrophes as shown below:

'A'.

The Java programming tools know how to cross reference that specific character symbol against the Unicode table to obtain the corresponding numeric value. *(A discussion of the use of the **char** type to represent*

characters that don't appear on your keyboard is beyond the scope of this module.)

I will discuss the operations that can be performed on the **char** type in a future module.

The boolean type

The boolean type is the simplest type supported by Java. It can have only two values:

- true
- false

Generally speaking, about the only operations that can be directly applied to an instance of the **boolean** type are to change it from **true** to **false** , and vice versa. However, the **boolean** type can be included in a large number of somewhat higher-level operations.

The **boolean** type is commonly used in some sort of a test to determine what to do next, such as that shown in Figure 8 .

Figure 8 . Example of the use of the boolean type.

Figure 8 . Example of the use of the boolean type.

```
Perform a test that returns a value of type
boolean.
if that value is true,
  do one thing
otherwise (meaning that value is false)
  do a different thing
```

I will discuss the operations that can be performed on the **boolean** type in more detail in a future module.

User-defined or reference types

Extending the language

Java is an *extensible* programming language. By this, I mean that there is a core component to the language that is always available. Beyond the core component, different programmers can extend the language in different ways to meet their individual needs.

Creating new types

One of the ways that individual programmers can extend the language is to create new types. When creating a new type, the programmer must define the set of values that can be stored in an instance of the type as well as the operations that can be performed on instances of the type.

No magic involved

While this might initially seem like magic, once you get to the heart of the matter, it is really pretty straightforward. New types are created by combining instances of primitive types along with instances of other user-

defined types. In other words, the process begins with the primitive types explained earlier and builds upward from there.

An example

For example, a **String** type, which can be used to represent a person's last name, is just a grouping of a bunch of instances of the primitive **char** or character type.

A user-defined **Person** type, which could be used to represent both a person's first name and their last name, might simply be a grouping of two instances of the user-defined **String** type. *(The **String** type is part of the Java standard library. However, the standard library doesn't have a type named **Person** . If you need that type, you will have to define it yourself.)*

Differences

The biggest conceptual difference between the **String** type and the **Person** type is that the **String** type is contained in the standard Java library while the **Person** type isn't in that library. However, you could put it in a library of your own design if you choose to do so.

Removing types

You could easily remove the **String** type from your copy of the standard Java library if you choose to do so, although that would probably be a bad idea. However, you cannot remove the primitive **double** type from the core language without making major modifications to the language.

The company telephone book

A programmer responsible for producing the company telephone book might create a new type that can be used to store the first and last names along with the telephone number of an individual. That programmer might choose to give the new type the name **Employee** .

The programmer could create an instance of the **Employee** type to represent each employee in the company, populating each such instance with the

name and telephone number for an individual employee. *(At this point, let me sneak a little jargon in and tell you that we will be referring to such instances as objects.)*

A comparison operation

The programmer might define one of the allowable operations for the new **Employee** type to be a comparison between two objects of the new type to determine which is greater in an alphabetical sorting sense. This operation could be used to sort the set of objects representing all of the employees into alphabetical order. The set of sorted objects could then be used to print a new telephone book.

A name-change operation

Another allowable operation that the programmer might define would be the ability to change the name stored in an object representing a particular employee. For example when Suzie Smith marries Tom Jones, she might elect to thereafter be known as

- Suzie Smith
- Suzie Jones,
- Suzie Smith-Jones,
- Suzie Jones-Smith, or
- something entirely different.

In this case, there would be a need to modify the object that represents Suzie in order to reflect her newly-elected surname. *(Or perhaps Tom Jones might elect to thereafter be known as Tom Jones-Smith, in which case it would be necessary to modify the object that represents him.)*

An updated telephone book

The person charged with maintaining the database could

- use the name-changing operation to modify the object and change the name,
- make use of the sorting operation to re-sort the set of objects, and

- print and distribute an updated version of the telephone book.

Many user-defined types already exist

Unlike the primitive types which are predefined in the core language, I am unable to give you much in the way of specific information about user-defined types, simply because they don't exist until a user defines them.

I can tell you, however, that when you obtain the Java programming tools from Sun, you not only receive the core language containing the primitive types, you also receive a large library containing several thousand user-defined types that have already been defined. A large documentation package is available from Sun to help you determine the individual characteristics of these user-defined types.

The most important thing

At this stage in your development as a Java programmer, the most important thing for you to know about user-defined types is that they are possible.

You can define new types. Unlike earlier procedural programming languages such as C and Pascal, you are no longer forced to adapt your problem to the available tools. Rather, you now have the opportunity to extend the tools to make them better suited to solve your problem.

The class definition

The specific mechanism that makes it possible for you to define new types in Java is a mechanism known as the *class definition* .

In Java, whenever you define a new class, you are at the same time defining a new type. Your new type can be as simple, or as complex and powerful as you want it to be.

An object *(instance)* of your new type can contain a very small amount of data, or it can contain a very large amount of data. The operations that you

allow to be performed on an object of your new type can be rudimentary, or they can be very powerful.

It is all up to you

Whenever you define a new class *(type)* you not only have the opportunity to define the data definition and the operations, you also have a responsibility to do so.

Much to learn and much to do

But, you still have much to learn and much to do before you will need to define new types.

There are a lot of fundamental programming concepts that we will need to cover before we seriously embark on a study involving the definition of new types.

For the present then, simply remember that such a capability is available, and if you work to expand your knowledge of Java programming one small step at a time, when we reach the point of defining new types, you will be ready and eager to do so.

Sample program

I'm not going to provide a sample program in this module. Instead, I will be using what you have learned about Java data types in the sample programs in future modules.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0150: Java OOP: A Gentle Introduction to Java Data Types
- File: Jb0150.htm
- Published: 11/17/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0150r Review
This module contains review questions and answers keyed to the module titled Jb0150: Java OOP: A Gentle Introduction to Java Data Types.

> Revised: Sun Mar 27 23:06:45 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0150: Java OOP: A Gentle Introduction to Java Data Types .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? Java is a type-sensitive language.

Answer 1

Question 2

True or false? Data type **double** involves whole numbers only *(no fractional parts are allowed)* .

Answer 2

Question 3

True or false? Type **double** involves numbers with fractional parts.

Answer 3

Question 4

True or false? All Java data types conceptually have something to do with numeric values.

Answer 4

Question 5

True or false? The Java **char** type deals conceptually with the letters of the alphabet, the numeric characters, and the punctuation characters.

Answer 5

Question 6

True or false? For every different type of data used with a particular programming language, there is a specification somewhere that defines two important characteristics of the type:

1. What is the set of all possible data values that can be stored in an instance of the type?
2. Once you have an instance of the type, what are the operations that you can perform on that instance alone, or in combination with other instances?

Answer 6

Question 7

True or false? If you have an instance of the **byte** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -256 to +255.

Answer 7

Question 8

Name or describe four of the operations that you can perform with data of type **short** .

Answer 8

Question 9

True or false? Java data types can be subdivided into two major categories:

- Primitive types
- User-defined or reference types

Answer 9

Question 10

True or false? The primitive types are not part of the core language.

Answer 10

Question 11

True or false? For purposes of discussion, primitive types can be subdivided into four categories:

- Whole-number types
- Floating-point types
- Character types
- Boolean types

Answer 11

Question 12

True or false? In Java, there are three different whole-number types:

- byte
- short
- int

Answer 12

Question 13

True or false? The whole-number types differ in terms of the range of values that they can accommodate and the amount of computer memory required to store instances of the types.

Answer 13

Question 14

True or false? Java provides an unsigned version of all of the primitive whole-number types.

Answer 14

Question 15

True or false? Floating point types represent values as a mantissa containing a decimal point along with an exponent value that tells how many places to shift the decimal point to the left or to the right in order to determine the true value.

Answer 15

Question 16

True or false? With a floating point type, positive exponent values mean that the decimal point should be shifted to the left. Negative exponent values mean that the decimal point should be shifted to the right.

Answer 16

Question 17

True or false? Java supports two different floating point types:

- float
- double

Answer 17

Question 18

True or false? The purpose of the **char** type is to make it possible to represent the letters of the alphabet, the punctuation characters, and the numeric characters internally in the computer. This is accomplished by assigning a numeric value to each character.

Answer 18

Question 19

True or false? The **char** type uses a standard character representation known as **Unicode** to represent up to 65,535 different characters.

Answer 19

Question 20

True or false? In Java, you usually represent a character in your program by surrounding it with quotation marks as shown below:

"A".

Answer 20

Question 21

True or false? The boolean type can have three values:

- true
- false
- maybe

Answer 21

Question 22

True or false? Java is an *extensible* programming language, meaning that there is a core component to the language that is always available. Beyond the core component, different programmers can extend the language in different ways to meet their individual needs.

Answer 22

Question 23

True or false? As is the case in C++, one of the ways that individual programmers can extend the Java language is to create overloaded operators for the primitive types.

Answer 23

Question 24

True or false? One of the ways that individual programmers can extend the Java language is to create new types.

Answer 24

Question 25

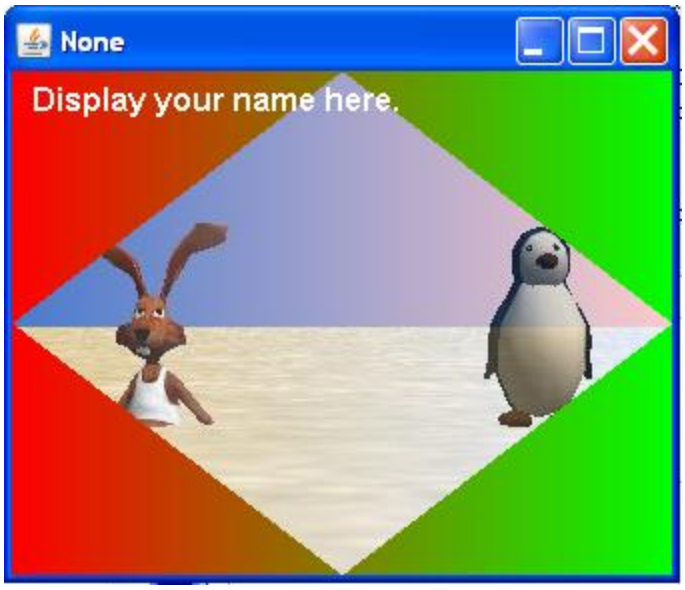
True or false? The specific Java mechanism that makes it possible for programmers to define new types is a mechanism known as the *class definition* .

Answer 25

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 25

True.

Back to Question 25

Answer 24

True.

Back to Question 24

Answer 23

False. Java does not allow programmers to create overloaded operators for the primitive types.

Back to Question 23

Answer 22

True.

Back to Question 22

Answer 21

False. The boolean type can have only two values:

- true
- false

Back to Question 21

Answer 20

False. In Java, you usually represent a character in your program by surrounding it with apostrophes as shown below:

'A'.

Back to Question 20

Answer 19

True.

Back to Question 19

Answer 18

True.

Back to Question 18

Answer 17

True.

Back to Question 17

Answer 16

False. With a floating point type, positive exponent values mean that the decimal point should be shifted to the ***right*** . Negative exponent values mean that the decimal point should be shifted to the ***left*** .

Back to Question 16

Answer 15

True.

Back to Question 15

Answer 14

False. Other than type **char** , there are no unsigned whole-number primitive types in Java.

Back to Question 14

Answer 13

True.

Back to Question 13

Answer 12

False. In Java, there are five different whole-number types:

- byte
- short
- int
- long
- char

Back to Question 12

Answer 11

True.

Back to Question 11

Answer 10

False. The primitive types are part of the core language.

Back to Question 10

Answer 9

True.

Back to Question 9

Answer 8

Four of the possible operations are:

- You can add them together.
- You can subtract one from the other.
- You can multiply one by the other.
- You can divide one by the other.

Back to Question 8

Answer 7

False. If you have an instance of the **byte** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -128 to +127.

Back to Question 7

Answer 6

True.

Back to Question 6

Answer 5

True.

Back to Question 5

Answer 4

False. In Java, data type ***boolean*** conceptually has nothing to do with numeric values, but deals only with the concept of ***true*** or ***false*** .

Back to Question 4

Answer 3

True.

Back to Question 3

Answer 2

False. Some data types such at type **int** involve whole numbers only *(no fractional parts are allowed)* .

Back to Question 2

Answer 1

True.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0150r Review: A Gentle Introduction to Java Data Types
- File: Jb0150r.htm
- Published: 11/21/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0160: Java OOP: Hello World
A traditional Hello World in Java provides interesting insights into the structure of a Java application.

Revised: Sat Sep 03 18:11:42 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

It is traditional in introductory programming courses to write and explain a simple program that prints the text ***"Hello World"*** on the computer screen.

This module continues that tradition.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1 . How to compile and run a Java application.

Listings

- Listing 1 . Beginning of the class named hello1.
- Listing 2 . Beginning of the main method.
- Listing 3 . Display the string Hello World.
- Listing 4 . End of the class named hello1.
- Listing 5 . Complete program listing.

Introduction

This module introduces you to Java programming by presenting and discussing a traditional *Hello World* program.

Two approaches

Java programs can be written and executed in several different ways, including the following:

- Stand-alone application from the command line.
- Applet that runs under control of a Java-capable browser.

It is also possible in many cases to write applets, which can be run in a stand-alone mode from the command line, or can be run under control of a Java-capable browser. An example of such an applet will be presented in a future module.

Applets vs. applications

Programming an *"application"* in Java is significantly different from programming an *"applet."* Applets are designed to be downloaded and executed on-line under control of a browser.

Restrictions on applets

Their functionality of an applet is usually restricted in an attempt to prevent downloaded applets from damaging your computer or your data. No such restrictions apply to the functionality of a Java application.

Class definitions

All Java programs consist of one or more **class** definitions. In this course, I will often refer to the *primary* class definition for a Java *application* as the *controlling class* .

The main method

A stand-alone Java application requires a method named **main** in its *controlling class* .

An **Applet** does not require a **main** method. The reason that a Java **Applet** does not require a **main** method will be explained in a future module.

Getting started

Figure 1 shows the steps for compiling and running a Java application.

Figure 1. How to compile and run a Java application.

Figure 1. How to compile and run a Java application.

Here are the steps for compiling and running a Java application, based on the assumption that you are running under Windows. If you are running under some other operating system, you will need to translate these instructions to that OS.

1. Download and install the JDK from Oracle. Also consider downloading and installing the documentation, which is a separate download.

2. Using any editor that can produce a plain text file *(such as Notepad)*, create a source code file with the extension on the file name being .java This file contains your actual Java instructions. *(You can copy some sample programs from the early lessons in this collection to get started.)*

3. Open a command-line window and change directory to the directory containing the source file. It doesn't really matter which directory the source file is in, but I normally put my Java files in a directory all their own.

4. Assume that the name of the file is ***joe.java*** , just to have something definitive to refer to.

5. To compile the file, enter the following command at the prompt:

```
javac joe.java
```

6. Correct any compiler errors that show up. Once you have corrected all compiler errors, the **javac** program will execute and return immediately to the prompt with no output. At that point, the directory should also contain a file named ***joe.class*** and possibly some other files with a .class extension as well. These are the compiled Java files.

7. To run the program, enter the following command:

```
java joe
```

8. If your program produces the correct output, congratulations. You have written, compiled, and executed a Java application. If not, you will need to determine why not.

The Java version of Hello World

The class file

Compiled Java programs are stored in *"bytecode"* form in a file with an extension of **.class** where the name of the file is the same as the name of the *controlling class (or other class)* in the program.

The main method is static

The **main** method in the controlling class of an application must be *static* , which results in **main** being a *class* method.

Class methods can be called without a requirement to instantiate an object of the class.

When a Java application is started, the *Java Virtual Machine or JVM (an executable file named java.exe)* finds and calls the **main** method in the class whose name matches the name of the class file specified on the command line.

Running an application

For example, to start the JVM and run a Java application named **hello1** , a command such as the following must be executed at the operating system prompt:

```
java hello1
```

This command instructs the operating system to start the JVM, and then instructs the JVM to find and execute the java application stored in the file named **hello1.class** . *(Note that the .class extension is not included in the command.)*

This sample program is a Java application named **hello1.java** .

When compiled, it produces a class file named **hello1.class** .

When the program is run, the JVM calls the **main** method defined in the *controlling class* .

The **main** method is a *class* method.

Class methods can be called without a requirement to instantiate an object of the class.

The program displays the following words on the screen:

```
Hello World
```

Interesting code fragments

I will explain this program code in fragments. A complete listing of the program is provided in Listing 5 .

The code fragment in Listing 1 shows the first line of the class definition for the controlling class named **hello1** . *(I will discuss class definitions in detail in a future module.)*

Listing 1 . Beginning of the class named hello1.

```
class hello1 { //define the controlling class
```

The code fragment in Listing 2 begins the definition of the **main** method. I will also discuss method definitions in detail in a future module.

Listing 2 . Beginning of the main method.

```
public static void main(String[] args){
```

The fragment in Listing 3 causes the string **Hello World** to be displayed on the command-line screen.

The statement in Listing 3 is an extremely powerful statement from an object-oriented programming viewpoint. When you understand how it works, you will be well on your way to understanding the Java version of Object-Oriented Programming (OOP).

I will discuss this statement in more detail later in a future module.

Listing 3 . Display the string Hello World.

Listing 3 . Display the string Hello World.

```
    System.out.println("Hello World");
```

Listing 4 ends the **main** method and also ends the class definition for the class named **hello1** .

Listing 4 . End of the class named hello1.

```
  }//end main
}//End hello1 class
```

The complete program listing

As mentioned earlier, a complete listing of the program is provided in Listing 5 near the end of the module.

General information

This program illustrates several general aspects of Java programming.

Overall skeleton of java program

The overall skeleton of any Java program consists of one or more class definitions.

All methods and variables must be defined inside a **class** definition. There can be no freestanding methods or global variables.

File names and extensions

The name of the *controlling class* should be the same as the name of the source file that contains it.

Files containing source code in Java have an extension of *java* .

The main method

The controlling class definition for an application must contain the **main** method.

The primary class file

The file produced by compiling the file containing the controlling class has the same name as the controlling class, and has an extension of **class** .

Many class files may be produced

The java compiler produces a separate file for every class definition contained in an application or applet, even if two or more class definitions are contained in the same source file.

Thus, the compilation of a large application can produce many different *class* files.

What are jar files?

A feature known as a *jar* file can be used to consolidate those class files into a single file for more compact storage, distribution, and transmission. Such a file has an extension of **.jar** . *(A jar file is similar to a zip file except that it is specialized for use with Java programs.)*

The main method is static

The controlling class for a Java application must contain a ***static*** method named **main** .

When you run the application using the JVM, you specify the name of the *class* file that you want to run.

The JVM then calls the **main** method defined in the *class* file having that name. This is possible because a *class method* can be called without a requirement to instantiate an object of the class.

The **main** method defined in that class definition controls the flow of the program.

Run the program

I encourage you to copy the code from Listing 5 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0160: Java OOP: Hello World
- File: Jb0160.htm
- Originally published: 1997
- Published at cnx.org: 11/17/12

Complete program listing

A complete listing of the program discussed in this module is provided in Listing 5 .

Listing 5 . Complete program listing.

Listing 5 . Complete program listing.

```
/*File hello1.java Copyright 1997, R.G.Baldwin
This is a Java application program .

When compiled, this program produces the class named:

hello1.class

When the Java interpreter is called upon the application's
controlling class using the following statement at the
command line:

java hello1

the interpreter starts the program by calling the main
method defined in the controlling class. The main method
is
a class method which can be called without the requirement
to instantiate an object of the class.

The program displays the following words on the screen:

Hello World

**********************************************************/
class hello1 { //define the controlling class
  //define main method
  public static void main(String[] args){
    //display text string
    System.out.println("Hello World");
  }//end main
}//End hello1 class.
```

-end-

Jb0160r Review
This module contains review questions and answers keyed to the module titled Jb0160: Java OOP: Hello World.

> Revised: Mon Mar 28 00:15:46 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0160: Java OOP: Hello World .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? Applications are designed to be downloaded and executed on-line under control of a web browser, while applets are designed to be executed in a stand-alone mode from the command line.

Answer 1

Question 2

True or false? All applications and applets written in Java require a **main** method.

Answer 2

Question 3

Explain the relationship between the name of the class file for a Java application and the location of the **main** method in the application.

Answer 3

Question 4

Explain how you cause a method to be a *class* method in Java.

Answer 4

Question 5

True or false? *Class* methods can be called without the requirement to instantiate an object of the class:

Answer 5

Question 6

Write the source code for a Java application that will display your name and address on the standard output device. Show the command-line statement that would be required to execute a compiled version of your application.

Answer 6

Question 7

Show the three styles of comment indicators that are supported by Java.

Answer 7

Question 8

True or false? Java allows free-standing methods outside of a class definition?

Answer 8

Question 9

What is the relationship between the name of the *controlling class* in an application and the names of the files that comprise that application.

Answer 9

Question 10

What is the relationship between the number of classes in an application and the number of separate files with the ***class*** extension that go to make up that application? How does this change when all the classes are defined in the same source file?

Answer 10

Question 11

True or false? ***Class*** methods in Java can only be called relative to a specific object.

Answer 11

Question 12

Write the signature line for the main method in a Java application.

Answer 12

Question 13

Write a Java application that will display your name on the screen.

Answer 13

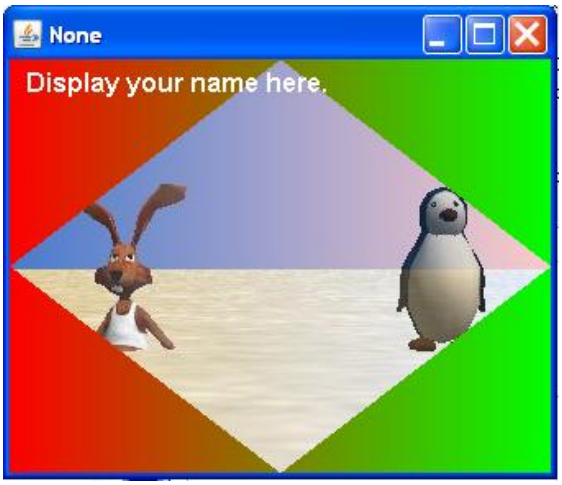
Listings

- Listing 1 . Listing for Answer 13.
- Listing 2 . Listing for Answer 7.
- Listing 3 . Listing for Answer 6.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 13

Listing 1 . Listing for Answer 13.

```
/*File SampProg02.java from lesson 10
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write an
application that will display your name on the screen.
**************************************************/
class SampProg02 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("Dick Baldwin");
  }//end main
}//End SampProg02 class.
```

Back to Question 13

Answer 12

Note:

```
public static void main(String[] args)
```

Back to Question 12

Answer 11

False. ***Class*** methods can be called by joining the name of the class with the name of the method using a period.

Back to Question 11

Answer 10

Each class definition results in a separate ***class*** file regardless of whether or not the classes are defined in separate source files.

Back to Question 10

Answer 9

One of the files must have the same name as the *controlling class* with an extension of ***class*** .

Back to Question 9

Answer 8

False.

Back to Question 8

Answer 7

Listing 2. Listing for Answer 7.

```
/** special documentation comment used by the JDK javadoc
tool */
/* C/C++ style multi-line comment */
// C/C++// C/C++ style single-line comment
```

Back to Question 7

Answer 6

Listing 3. Listing for Answer 6.

```
/*File Name01.java
This is a Java application that will display a
name on the standard output device.

The command required at the command line to execute this
program is:

java Name01

**********************************************************/

class Name01 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println(
     "Dick Baldwin\nAustin Community College\nAustin, TX");
  }//end main
}//End Name01 class.
```

Note that the ***\n*** characters in Listing 3 cause the output display to advance to the next line.

Back to Question 6

Answer 5

True.

Back to Question 5

Answer 4

Preface or precede the name of the method with the **static** keyword.

Back to Question 4

Answer 3

The name of the class file must be the same as the name of the class that contains the **main** method *(sometimes called the controlling class)* .

Back to Question 3

Answer 2

False. Applets do not require a **main** method while applications do require a **main** method.

Back to Question 2

Answer 1

False. Applications are for stand-alone use while applets are for browsers.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0160r Review: Hello World
- File: Jb0160r.htm
- Published: 11/18/12

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0170: Java OOP: A little more information about classes.
Class definitions form the foundation for Java OOP. They are discussed in increasing detail in subsequent modules. This module sheds just a little more light on classes.

> Revised: Mon Mar 28 00:23:43 CDT 2016
>
> *This page is included in the following Books:*
>
> - *Programming Fundamentals with Java*
> - *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module is part of a collection of modules designed to help you learn to program computers.

This module sheds a little more light on the Java construct called a *class* .

Listings

Introduction

New types

Java makes extensive use of classes. When a class is defined in Java, a new *type* comes into being. The new type definition can then be used to instantiate *(create instances of)* one or more objects of that new type.

A blueprint

The class definition provides a *blueprint* that describes the *data* contained within, and the *behavior* of objects instantiated according to the new type.

The data

The data is contained in variables defined within the class *(often called member variables, data members, attributes, fields, properties, etc.*).

The behavior

The behavior is controlled by methods defined within the class.

State and behavior

An object is said to have *state* and *behavior* . At any instant in time, the *state* of an object is determined by the values stored in its *variables* and its behavior is determined by its *methods* .

Class vs. instance

It is possible to define:

- instance variables and instance methods
- static or *class* variables and static or *class* methods.

Instance variables and instance methods can only be accessed through an object instantiated from the class. They belong to the individual objects, *(which is why they are called instance variables and instance methods)* .

Class variables and *class* methods can be accessed without first instantiating an object. They are shared among all of the objects instantiated

from the class and are even accessible in the total absence of an object of the class.

The class name alone is sufficient for accessing *class* variables and *class* methods by joining the name of the class to the name of the variable or method using a period.

Defining a class in Java

The general syntax for defining a class in Java is shown in Listing 1 .

Listing 1 . General syntax for defining a Java class.

```
class MyClassName{
  . . .
} //End of class definition.
```

This syntax defines a class and creates a new type named **MyClassName** .

The definitions of variables, methods, constructors, and a variety of other members are inserted between the opening and closing curly brackets.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0170: Java OOP: A little more information about classes.
- File: Jb0170.htm
- Originally published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0170r: Review
This module contains review questions and answers keyed to the module titled Jb0170: Java OOP: A little more information about classes.

Revised: Mon Mar 28 10:56:27 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0170: Java OOP: A little more information about classes .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

List two of the many names commonly used for variables defined within a class in Java.

Answer 1

Question 2

List two of the many names commonly used for the functions defined within a class in Java.

Answer 2

Question 3

An object is said to have *state* and *behavior* . At any instant in time, the *state* of an object is determined by the values stored in its (a)_____ and its behavior is determined by its (b)_____.

Answer 3

Question 4

What keyword is used to cause a variable or method to become a *class* variable or *class* method in Java?

Answer 4

Question 5

True or false? *Instance* variables and *instance* methods can only be accessed through an object of the class in Java.

Answer 5

Question 6

True or false? In Java, the class name alone is sufficient for accessing *class* variables and *class* methods by joining the name of the class with the name of the variable or method using a colon.

Answer 6

Question 7

True or false? Show the general syntax of an empty class definition in Java.

Answer 7

Question 8

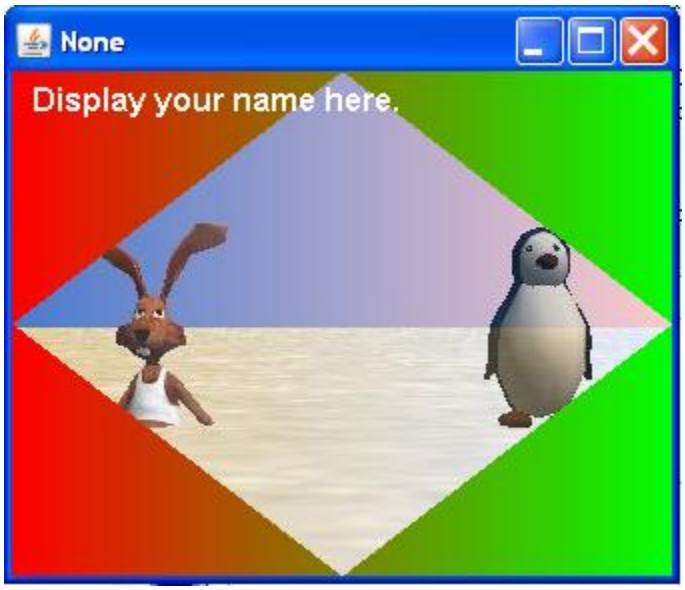
True or false? The syntax for a class definition in Java requires a semicolon following the closing curly bracket.

Answer 8

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 8

False. Java does not require the use of a semicolon following the closing curly bracket in a class definition.

Back to Question 8

Answer 7

class NameOfClass{}

Back to Question 7

Answer 6

False. A colon is not used in Java. Instead, a period is used in Java.

Back to Question 6

Answer 5

True.

Back to Question 5

Answer 4

static

Back to Question 4

Answer 3

- (a) instance variables
- (b) methods

Back to Question 3

Answer 2

Member functions and instance methods.

Back to Question 2

Answer 1

Instance variables and attributes.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0170r: Review: A little more information about classes
- File: Jb0170r.htm
- Published: 11/21/12

should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0180: Java OOP: The main method.
Every Java application requires a class containing a method named main.
This module provides information on the main method.

Revised: Mon Mar 28 11:21:22 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module is part of a collection of modules designed to help you learn to program computers.

Every Java application requires a class containing a method named **main** .
This module provides information on the **main** method.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

Figures

The main method in Java

There must be a **main** method in the controlling class in every Java application.

The method signature

The Java literature frequently refers to the signature of a method, or the *method signature* .

Exploring Java by Patrick Niemeyer and Joshua Peck (O'Reilly) provides the definition of a method signature shown in Figure 1 .

Figure 1. The method signature according to Niemeyer and Peck .

"A method signature is a collection of information about the method, as in a C prototype or a forward function declaration in other languages. It includes the method's name, type, and visibility, as well as its arguments and return type."

Type

Apparently in this definition, the authors are referring to the *type* of the method as distinguishing between static and non-static. *(Other literature refers to the type of a function or method as being the return type which according to the above definition is a separate part of the signature.)*

Visibility

Apparently also the use of the word visibility in the above definition refers to the use of **public** , **private** , etc.

According to Oracle...

Oracle's Java Tutorials , on the other hand, describe the method signature as in Figure 2 .

Figure 2. The method signature according to Oracle .

Definition: Two of the components of a method declaration comprise the method signature -- the method's name and the parameter types.

As you can see, the Oracle definition is more restrictive than the Niemeyer and Peck definition.

Bottom line on method signature

The method signature can probably be thought of as providing information about the programming interface to the method. In other words, it provides the information that you need to be able to call the method in your program code.

Signature of main method

The controlling class of every Java application must contain a **main** method having one of the signatures shown in Figure 3 .

Figure 3 . Allowable signatures for the main method.

```
public static void main(String[] args)

public static void main(String args[])
```

*(I prefer the first signature in Figure 3 as being the most descriptive of an array of **String** references which is what is passed in as an argument.)*

public

The keyword **public** indicates that the method can be called by any object. A future module will discuss the keywords **public** , **private** , and **protected** in more detail.

static

The keyword **static** indicates that the method is a *class* method, which can be called without the requirement to instantiate an object of the class. This is used by the JVM to launch the program by calling the **main** method of the class identified in the command to start the program.

void

The keyword **void** indicates that the method doesn't return any value.

args

The formal parameter **args** is a reference to an array object of type **String** . The array elements contain references to **String** objects that encapsulate **String** representations of the arguments, if any, entered at the command line.

Note that the **args** parameter must be specified whether or not the user is required to enter command-line arguments and whether or not the code in

the program actually makes use of the argument. Also note that the name can be any legal Java identifier. It doesn't have to be **args** . It could be joe or sue, for example.

The length property

The parameter named **args** is a reference to an array object. Java array objects have a property named **length** , which specifies the number of elements in the array.

The runtime system monitors for the entry of command-line arguments by the user and constructs the **String** array containing those arguments.

Processing command-line arguments

The **args.length** property can be used by the code in the program to determine the number of arguments actually entered by the user.

If the **length** property is not equal to zero, the first string in the array corresponds to the first argument entered on the command line.

Command-line arguments along with strings and **String** arrays will be discussed in more detail in a future module.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0180: Java OOP: The main method.
- File: Jb0180.htm
- Originally published: 1997
- Published at cnx.org: 11/17/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0180r Review
This module contains review questions and answers keyed to the module titled Jb0180: Java OOP: The main method.

Revised: Mon Mar 28 11:31:59 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0180: Java OOP: The main method .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

Write the method signature for the **main** method in a Java application.

Answer 1

Question 2

Briefly explain the reason that the **main** method in a Java application is declared **public** .

Answer 2

Question 3

Explain the reason that the **main** method in a Java application must be declared **static** .

Answer 3

Question 4

Describe the purpose of the keyword **void** when used as the return type for the **main** method.

Answer 4

Question 5

True or false? If the Java application is not designed to use command-line arguments, it is not necessary to include a formal parameter for the **main** method.

Answer 5

Question 6

True or false? When using command-line arguments in Java, if the name of the string array is **args** , the **args.length** variable can be used by the code in the program to determine the number of arguments actually entered.

Answer 6

Question 7

True or false? The first string in the array of command-line arguments contains the name of the Java application

Answer 7

Question 8

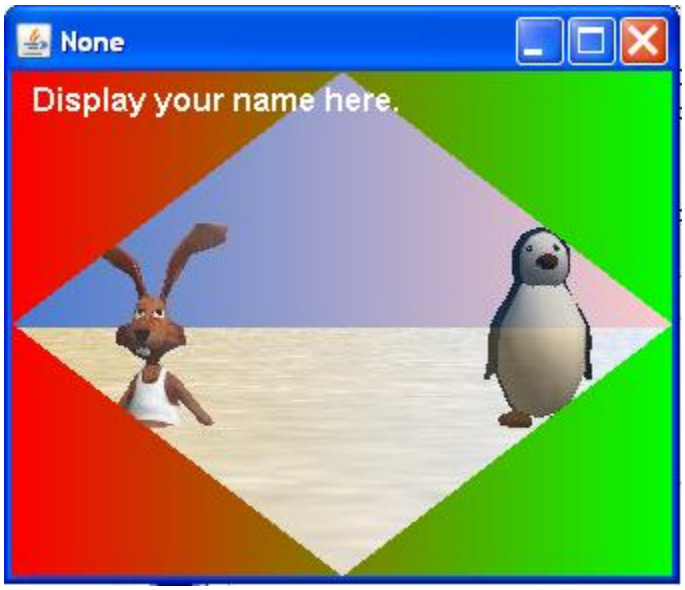
The *controlling class* of every Java application must contain a **main** method. Can other classes in the same application also have a **main** method? If not, why not? If so, why might you want to do that?

Answer 8

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 8

Any and all classes in a Java application can have a **main** method. Only the one in the *controlling class* for the program being executed is actually

called.

It is often desirable to provide a **main** method for a class that will not ultimately be the *controlling class* to allow the class to be tested in a stand-alone mode, independent of other classes.

Back to Question 8

Answer 7

False. Unlike C++, the first string in the array of command-line arguments in a Java application does **not** contain the name of the application.

Back to Question 7

Answer 6

True.

Back to Question 6

Answer 5

False. The **main** method in a Java program must always provide the formal argument list regardless of whether it is actually used in the program.

Back to Question 5

Answer 4

The **void** keyword when used as the return type for any Java method indicates that the method does not **return** anything.

Back to Question 4

Answer 3

The keyword **static** indicates that the method is a *class* method which can be called without the requirement to instantiate an object of the class. This is used by the Java virtual machine to launch the program by calling the **main** method of the class identified in the command to start the program.

Back to Question 3

Answer 2

The keyword **public** indicates that the method can be called by any object.

Back to Question 2

Answer 1

Note:

```
public static void main(String[] args)
```

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0180r Review: The main method
- File: Jb0180r.htm
- Published: 11/21/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0190: Java OOP: Using the System and PrintStream Classes
Take a preliminary look at the complexity of OOP by examining some aspects of the System and PrintStream classes.

Revised: Mon Mar 28 11:44:44 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module takes a preliminary look at the complexity of OOP by examining some aspects of the **System** and **PrintStream** classes.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

Introduction

This lesson introduces you to the use of the **System** and **PrintStream** classes in Java. This is our first introduction to the complexity that can accompany the **OOP** paradigm. It gets a little complicated, so you might need to pay special attention to the discussion.

Discussion

What does the main method do?

The **main** method in the controlling class of a Java application controls the flow of the program.

The **main** method can also access other classes along with the variables and methods of those classes and of objects instantiated from those classes.

The hello1 Application

Listing 1 shows a simple Java application named **hello1** .

(By convention, class definitions should begin with an upper-case character. However, the original version of this module was written and published in 1997, before that convention was firmly established.)

Listing 1 . The program named hello1.

```
/*File hello1.java Copyright 1997, R.G.Baldwin
**************************************************/
class hello1 { //define the controlling class
  //define main method
  public static void main(String[] args){
    //display text string
    System.out.println("Hello World");
  }//end main
}//End hello1 class.  No semicolon at end of Java class.
```

Does this program Instantiate objects?

This is a simple example that does not instantiate objects of any other class.

Does this program access another class?

However, it does access another class. It accesses the **System** class that is provided with the Java development kit. *(The **System** class will be discussed in more detail in a future module.)*

The variable named *out*

The variable named **out** , referred to in Listing 1 as **System.out** , is a *class variable* of the **System** class. *(A class variable is a variable that is declared to be static.)*

Recall that a class variable can be accessed without a requirement to instantiate an object of the class. As is the case with all variables, the class variable must be of some specific type.

Primitive variables vs. reference variables

A class variable may be a *primitive variable* , which contains a primitive value, or it may be a *reference variable* , which contains a reference to an object.

(I'll have more to say about the difference between primitive variables and reference variables in a future module.)

The variable named **out** in this case is a *reference variable* , which refers to an object of another type.

Accessing class variables

You access class variables or class methods in Java by joining the name of the class to the name of the variable or method with a period as shown below.

Note:
System.out
accesses the class variable named **out** in the Java class named **System** .

The PrintStream class

Another class that is provided with the Java development kit is the **PrintStream** class. The **PrintStream** class is in a package of classes that are used to provide stream input/output capability for Java.

What does the out variable refer to?

The **out** variable in the **System** class refers to *(points to)* an instance of the **PrintStream** class *(a **PrintStream** object)*, which is automatically instantiated when the **System** class is loaded into the application.

We will be discussing the **PrintStream** class along with a number of other classes in detail in a future module on input/output streams, so this is not intended to be an exhaustive discussion.

The println method

The **PrintStream** class has an *instance method* named **println** , which causes its argument to be displayed on the standard output device when it is called.

(Typically, the standard output device is the command-line window. However, it is possible to redirect it to some other device.)

Accessing an instance method

The method named **println** can be accessed by joining a **PrintStream** object's reference to the name of the method using a period.

Thus, *(assuming that the standard output device has not been redirected)* , the statement shown in Listing 2 causes the string "Hello World" *(without the quotation marks)* to be displayed in the command-line window.

Listing 2 . Display the string "Hello World".

```
System.out.println("Hello World");
```

This statement calls the **println** method of an object instantiated from the **PrintStream** class, which is referred to *(pointed to)* by the variable named **out** , which is a *class variable* of the **System** class.

Read the previous paragraph very carefully. As I indicated when I started this module, this is our first introduction to the complexity that can result from use of the OOP paradigm. *(It can get even more complicated.)* If this is not clear to you, go back over it and think about it until it becomes clear.

A word about class variables

How many instances of a class variable exist?

The runtime system allocates a class variable only once no matter how many instances *(objects)* of the class are instantiated.

All objects of the class share the same physical memory space for the class variable.

If a method in one object changes the value stored in the class variable, it is changed insofar as all of the objects are concerned. *(This is about as close to a global variable as you can get in Java.)*

Accessing a class variable

You can use the name of the class to access class variables by joining the name of the class to the name of the variable using a period.

You can also access a class variable by joining the name of a reference variable containing an object's reference to the name of the variable using a period as the joining operator.

Referencing object methods via class variables

Class variables are either primitive variables or reference variables. *(Primitive variables contain primitive values and reference variables contain references to objects.)*

A referenced object may provide methods to control the behavior of the object. In Listing 2 , we accessed the **println** method of an object of the **PrintStream** class referred to by the class variable named **out** .

Instance variables and methods

As a side note, in addition to class variables, Java provides *instance variables* and *instance methods* . Every instance of a class has its own set of instance variables. You can only access instance variables and instance methods through an object of the class.

Run the program

I encourage you to copy the code from Listing 1 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0190: Java OOP: Using the System and PrintStream Classes
- File: Jb0190.htm
- Originally published: 1997
- Published at cnx.org: 11/18/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0190r: Review
This module contains review questions and answers keyed to the module titled Jb0190: Java OOP: Using the System and PrintStream Classes

Revised: Mon Mar 28 11:58:41 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0190: Java OOP: Using the System and PrintStream Classes .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? The **main** method in the controlling class of a Java application controls the flow of the program.

Answer 1

Question 2

True or false? The **main** method cannot access the variables and methods of objects instantiated from other classes.

Answer 2

Question 3

True or false? The **main** method must instantiate objects of other classes in order for the program to execute.

Answer 3

Question 4

True or false? In order to be useful, the **System** class must be used to instantiate objects in a Java application.

Answer 4

Question 5

True or false? *Class* variables such as the **out** variable of the **System** class must be of some specific type.

Answer 5

Question 6

True or false? *Class* variables must be of a primitive type such as **int** or **float** .

Answer 6

Question 7

True or false? The **out** variable in the **System** class is of a primitive type.

Answer 7

Question 8

What does the following code fragment access?

Note:

```
System.out
```

Answer 8

Question 9

True or false? An object of type **PrintStream** is automatically instantiated when the **System** class is loaded into an application.

Answer 9

Question 10

True or false? The **out** variable in the **System** class refers to an instance of what class?

Answer 10

Question 11

True or false? The **println** method is an instance method of what class?

Answer 11

Question 12

What is the primary behavior of the **println** method?

Answer 12

Question 13

How can the **println** method be accessed?

Answer 13

Question 14

Assuming that the standard output device has not been redirected, write a code fragment that will cause your name to be displayed on the screen.

Answer 14

Question 15

Explain how your code fragment in Answer 14 produces the desired result.

Answer 15

Question 16

If you have a class named **MyClass** that has a class variable named **myClassVariable** that requires four bytes of memory and you instantiate ten objects of type **MyClass** , how much total memory will be allocated to contain the allocated variables *(assume that the class definition contains no other class, instance, or local variables)* .

Answer 16

Question 17

How many actual instances of the variable named **out** are allocated in memory by the following code fragment?

Note:

```
System.out.println("Dick Baldwin");
```

Answer 17

Question 18

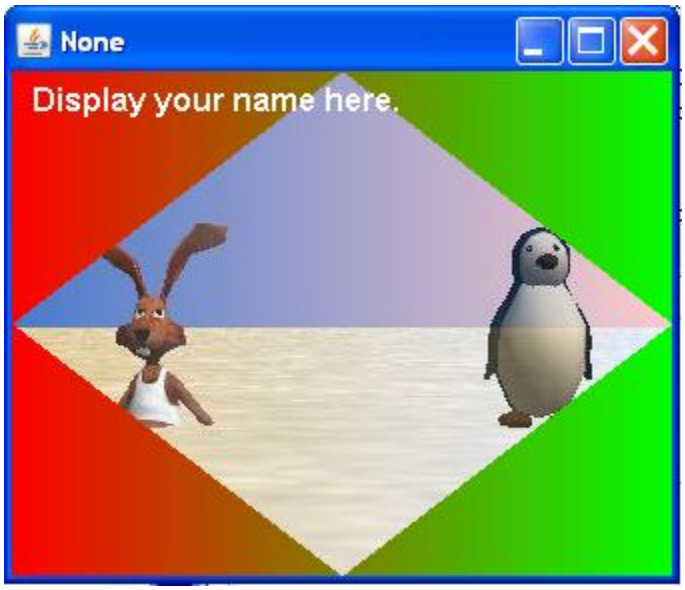
If you have a class named **MyClass** that has an instance variable named **myInstanceVariable** that requires four bytes of memory and you instantiate ten objects of type **MyClass** , how much total memory will be allocated to contain the allocated variables *(assume that the class definition contains no other class, instance, or local variables)* .

Answer 18

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 18

Every instance of a class has its own set of instance variables. You can only access *instance variables* and *instance methods* through an object of the class. In this case, forty bytes of memory would be required to contain the instance variables of the ten objects.

Back to Question 18

Answer 17

Only one, because **out** is a class variable of the **System** class.

Back to Question 17

Answer 16

The runtime system allocates a *class variable* only once no matter how many instances of the class are instantiated. Thus, all objects of the class share the same physical memory space for the class variable, and in this case, only four bytes of memory will be allocated to contain the allocated variables.

Back to Question 16

Answer 15

The statement in Answer 14 calls the **println** method belonging to an object of the **PrintStream** class, which is referenced *(pointed to)* by the **out** variable, which is a *class* variable of the **System** class.

Back to Question 15

Answer 14

Note:

```
System.out.println("Dick Baldwin");
```

Back to Question 14

Answer 13

The **println** method can be accessed by joining the name of a variable that references a **PrintStream** object to the name of the **println** method using a

period.

Back to Question 13

Answer 12

The **println** method causes its argument to be displayed on the standard output device. *(The standard output device is the screen by default, but can be redirected by the user at the operating system level.)*

Back to Question 12

Answer 11

The **println** method is an instance method of the **PrintStream** class.

Back to Question 11

Answer 10

The **out** variable in the **System** class refers to an instance of the **PrintStream** class *(a **PrintStream** object)*, which is automatically instantiated when the **System** class is loaded into the application.

Back to Question 10

Answer 9

True.

Back to Question 9

Answer 8

The code fragment accesses the contents of the *class* variable named **out** in the class named **System** .

Back to Question 8

Answer 7

False. the variable named **out** defined in the **System** class is a reference variable that points to an object of another type.

Back to Question 7

Answer 6

False. A *class* variable can be a primitive type, or it can be a reference variable that points to another object.

Back to Question 6

Answer 5

True.

Back to Question 5

Answer 4

False. The **System** class has several *class* variables *(including **out** and **in***) that are useful without the requirement to instantiate an object of the **System** class.

Back to Question 4

Answer 3

False. While it is probably true that the **main** method must instantiate objects of other classes in order to accomplish much that is of value, this is not a requirement. The **main** method in the "Hello World" program of this module does not instantiate objects of any class at all.

Back to Question 3

Answer 2

False. The **main** method can access the variables and methods of objects instantiated from other classes. Otherwise, the flow of the program would be stuck within the **main** method itself and wouldn't be very useful.

Back to Question 2

Answer 1

True.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0190r: Review: Using the System and PrintStream Classes
- File: Jb0190r.htm
- Published: 11/22/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0200: Java OOP: Variables
Earlier modules have touched briefly on the topic of variables. This module discusses Java variables in depth.

Revised: Mon Mar 28 12:30:19 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

Earlier modules have touched briefly on the topic of variables. This module discusses Java variables in depth.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

Listings

Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as

- variables
- types
- expressions
- flow-of-control, etc.

This and several future modules concentrate on that foundation.

A sample program

The module begins with a sample Java program named **simple1** . The user is asked to enter some text and to terminate with the # character.

(This program contains a lot of code that you are not yet prepared to understand. For the time being, just concentrate on the use of variables in the program. You will learn about the other aspects of the program in future modules.)

The program loops, saving individual characters until encountering the # character. When it encounters the # character, it terminates and displays the character entered immediately prior to the # character.

Sample program named simple1

A complete listing of the program named **simple1** is provided in Listing 1 . Discussions of selected portions of the program are presented later in the module.

Listing 1 . Source code for the program named simple1.

Listing 1 . Source code for the program named simple1.

```
/*File simple1.java Copyright 1997, R.G.Baldwin
This Java application reads bytes from the keyboard until
encountering the integer representation of '#'.  At
the end of each iteration, it saves the byte received and
goes back to get the next byte.

When the '#' is entered, the program terminates input and
displays the character which was entered before the #.
**********************************************************/

class simple1 { //define the controlling class

  //It is necessary to declare that this method
  // can throw the exception shown below (or catch it).
  public static void main(String[] args) //define main
                        throws java.io.IOException {

    //It is necessary to initialize ch2 to avoid a compiler
    // error (possibly uninitialized variable) at the
    // statement which displays ch2.
    int ch1, ch2 = '0';

    System.out.println(
                      "Enter some text, terminate with #");

    //Get and save individual bytes
    while( (ch1 = System.in.read() ) != '#')
      ch2 = ch1;

    //Display the character immediately before the #
    System.out.println(
          "The char before the # was " + (char)ch2);
  }//end main
}//End simple1 class.
```

Program output

The output produced by compiling and running this program is shown in Figure 1 . The second line of text in Figure 1 ending with the # character was typed by the user.

Figure 1 . Screen output from the program named simple1.

```
Enter some text, terminate with #
abcde#
The char before the # was e
```

Discussion of the simple1 program

Purpose

I will use the program shown in Listing 1 to discuss several important aspects of the structure of a Java program. I will also provide two additional sample programs that illustrate specific points not illustrated in the above program later in this module.

Variables

Note: What is a variable?
Variables are used in a Java program to contain data that changes during the execution of the program.

Declaring a variable

To use a variable, you must first notify the compiler of the *name* and the *type* of the variable. This is known as *declaring a variable* .

The syntax for declaring a variable is to precede the *name* of the variable with the *name of the type* of the variable as shown in Listing 2 . It is also possible *(but not always required)* to initialize a variable in Java when it is declared as shown in Listing 2 .

Listing 2 . Declaring and initializing two variables named ch1 and ch2.

Listing 2 . Declaring and initializing two variables named ch1 and ch2.

```
int ch1, ch2 = '0';
```

The statement in Listing 2 declares two variables of type **int** , initializing the second variable *(ch2)* to the value of the zero character (0). *(Note that I didn't say initialized to the value zero.)*

Note: Difference between zero and '0' - Unicode characters
The value of the zero character is not the same as the numeric value of zero, but hopefully you already knew that.
As an aside, characters in Java are 16-bit entities called Unicode characters instead of 8-bit entities as is the case with many programming languages. The purpose is to provide many more possible characters including characters used in alphabets other than the one used in the United States.

Initialization of the variable

Initialization of the variable named **ch2** in this case was necessary to prevent a compiler error. Without initialization of this variable, the compiler would recognize and balk at the possibility that an attempt might be made to execute the statement shown in Listing 3 with a variable named **ch2** that had not been initialized

Listing 3 . Display the character.

```
System.out.println("The char before the # was "
                                        + (char)ch2);
```

Error checking by the compiler

The strong error-checking capability of the Java compiler would refuse to compile this program until that possibility was eliminated by initializing the variable.

Using the cast operator

You should also note that the contents of the variable **ch2** is being *cast* as type **char** in Listing 3 .

(A cast is used to change the type of something to a different type.)

Recall that **ch2** is a variable of type **int** , containing the numeric value that represents a character.

We want to display the character that the numeric value represents and not the numeric value itself. Therefore, we must cast it *(purposely change its type for the evaluation of the expression)* . Otherwise, we would not see the character on the screen. Rather, we would see the numeric value that represents that character.

Note: Initialization of instance variables and local variables:
As another aside, *member variables* in Java are automatically initialized to zero or the equivalent of zero. However, *local variables* , of which ch2 is an example, are not automatically initialized.

Why declare the variables as type int?

It was necessary to declare these variables as type **int** because the statement in Listing 4 *(more specifically, the call to the System.in.read method)* returns a value of type **int** .

Listing 4 . Beginning of a while loop.

```
while( (ch1 = System.in.read() ) != '#') ch2 = ch1;
```

Java provides very strict type checking and generally refuses to compile statements with type mismatches.

(There is a lot of complicated code in Listing 4 that I haven't previously explained. I will explain that code later in this and future modules.)

Another variable declaration

The program in Listing 1 also makes another variable declaration shown by the statement in Listing 5 .

Listing 5 . Beginning of the main method.

```
public static void main(String[] args) //define main
method
```

An array of String references

In Listing 5 , the formal argument list of the **main** method declares an argument named **args** *(first cousin to a variable)* as a reference to an array object of type **String** .

Capturing command-line arguments in Java

As you learned in an earlier module, this is the feature of Java that is used to capture arguments entered on the command line, and is required whether arguments are entered or not. In this case, no command-line arguments were entered, and the variable named **args** is simply ignored by the remainder of the program.

The purpose of the type of a variable

Note: All variables must have a declared type
The type determines the set of values that can be stored in the variable and the operations that can be performed on the variable.

For example, the **int** type can only contain whole numbers *(integers)* . A whole host of operations are possible with an **int** variable including add, subtract, divide, etc.

Signed vs. unsigned variables

Unlike C++, all variables of type **int** in Java contain signed values. In fact, with the exception of type **char** , all primitive numeric types in Java contain signed values.

Platform independence

At this point in the history of Java, a variable of a specified type is represented exactly the same way regardless of the platform on which the application or applet is being executed.

This is one of the features that causes compiled Java programs to be platform-independent.

Primitive types

In Java, there are two major categories of data types:

- primitive types
- reference *(or object)* types.

Primitive variables contain a single value of one of the eight primitive types shown in Listing 2 .

Reference variables contain references to objects *(or null, meaning that they don't refer to anything)* .

The eight primitive types in Java?

The table in Figure 2 lists all of the primitive types in Java along with their size and format, and a brief description of each.

Figure 2 . Information about the primitive types in Java.

Figure 2 . Information about the primitive types in Java.

```
Type       Size/Format                 Description
byte      8-bit two's complement     Byte-length integer
short    16-bit two's complement     Short integer
int      32-bit two's complement     Integer
long     64-bit two's complement     Long Integer
float    32-bit IEEE 754 format      Single-precision
                                     floating point
double   64-bit IEEE 754 format     Double-precision
                                     floating point
char     16-bit Unicode character    Single character
boolean  true or false               True or False
```

The char type

The **char** type is a 16-bit Unicode character value that has the possibility of representing more than 65,000 different characters.

Evaluating a primitive variable

A reference to the name of a primitive variable in program code evaluates to the value stored in the variable. In other words, when you call out the name of a primitive variable in your code, what you get back is the value stored in the variable.

Object-oriented wrappers for primitive types

Primitive types are not objects

Primitive data types in Java *(int, double, etc.)* are not objects. This has some ramifications as to how they can be used *(passing to methods, returning from methods, etc.)* .

The generic Object type

Later on in this course of study, you will learn that much of the power of Java derives from the ability to deal with objects of any type as the generic type **Object** . For example, several of the standard classes in the API *(such as the powerful **Vector** class)* are designed to work only with objects of type **Object** .

*(Note that this document was originally published prior to the introduction of generics in Java. The introduction of generics makes it possible to cause the **Vector** class to deal with*

*objects of types other than **Object** . However, that doesn't eliminate the need for wrapper classes.)*

Converting primitives to objects

Because it is sometimes necessary to deal with a primitive value as though it were an object, Java provides *wrapper* classes that support object-oriented functionality for Java's primitive data types.

The Double wrapper class

This is illustrated in the program shown in Listing 6 that deals with a **double** type as an object of the class **Double** .

*(Remember, Java is a case-sensitive language. Note the difference between the primitive **double** type and the class named **Double** .)*

Listing 6 . The program named wrapper1.

Listing 6 . The program named wrapper1.

```
/*File wrapper1.java Copyright 1997, R.G.Baldwin
This Java application illustrates the use of wrappers
for the primitive types.

This program produces the following output:

My wrapped double is 5.5
My primitive double is 10.5

**************************************************************/
class wrapper1 { //define the controlling class
  public static void main(String[] args){//define main

    //The following is the declaration and instantiation of
    // a Double object, or a double value wrapped in an
    // object. Note the use of the upper-case D.
    Double myWrappedData = new Double(5.5);

    //The following is the declaration and initialization
    // of a primitive double variable. Note the use of the
    // lower-case d.
    double myPrimitiveData = 10.5;

    //Note the call to the doubleValue() method to obtain
    // the value of the double wrapped in the Double
    // object.
    System.out.println(
      "My wrapped double is " +
myWrappedData.doubleValue());
    System.out.println(
             "My primitive double is " + myPrimitiveData );
  }//end main
}//End wrapper1 class.
```

The operation of this program is explained in the comments, and the output from the program is shown in the comments at the beginning.

Reference types

Once again, what is a primitive type?

Primitive types are types where the name of the variable evaluates to the value stored in the variable.

What is a reference type?

Reference types in Java are types where the name of the variable evaluates to the address of the location in memory where the object referenced by the variable is stored.

Note: The above statement may not really be true?
However, we can think of it that way. Depending on the particular JVM in use, the reference variable may refer to a table in memory where the address of the object is stored. In that case the second level of indirection is handled behind the scenes and we don't have to worry about it.
Why would a JVM elect to implement another level of indirection? Wouldn't that make programs run more slowly?
One reason has to do with the need to compact memory when it becomes highly fragmented. If the reference variables all refer directly to memory locations containing the objects, there may be many reference variables that refer to the same object. If that object is moved for compaction purposes, then the values stored in every one of those reference variables would have to be modified.
However, if those reference variables all refer to a table that has one entry that specifies where the object is stored, then when the object is moved, only the value of that one entry in the table must be modified.
Fortunately, that all takes place behind the scenes and we as programmers don't need to worry about it.

Primitive vs. reference variables

We will discuss this in more detail in a future module. For now, suffice it to say that in Java, a variable is either a primitive type or a reference type, and cannot be both.

Declaring, instantiating, initializing, and manipulating a reference variable

The fragment of code shown in Listing 7 , *(which was taken from the program shown in Listing 6 that deals with wrappers)* does the following. It

- declares,
- instantiates,
- initializes, and

- manipulates a variable of a reference type named **myWrappedData** .

In Listing 7 , the variable named **myWrappedData** contains a reference to an object of type **Double** .

Listing 7 . Aspects of using a wrapper class.

```
Double myWrappedData = new Double(5.5);

//Code deleted for brevity

//Note the use of the doubleValue() method to obtain the
// value of the double wrapped in the Double object.
System.out.println
  ("My wrapped double is " + myWrappedData.doubleValue()
);
```

Variable names

The rules for naming variables are shown in Figure 3 .

Figure 3 . Rules for naming variables.

Figure 3 . Rules for naming variables.

- Must be a legal Java identifier *(see below)* consisting of a series of Unicode characters. Unicode characters are stored in sixteen bits, allowing for a very large number of different characters. A subset of the possible character values matches the 127 possible characters in the ASCII character set, and the extended 8-bit character set, ISO-Latin-1 *(The Java Handbook, page 60, by Patrick Naughton).*
- Must not be the same as a Java keyword and must not be true or false.
- Must not be the same as another variable whose declaration appears in the same scope.

The rules for legal identifiers are shown in Figure 4 .

Figure 4 . Rules for legal identifiers.

- In Java, a legal identifier is a sequence of Unicode letters and digits of unlimited length.
- The first character must be a letter.
- All subsequent characters must be letters or numerals from any alphabet that Unicode supports.
- In addition, the underscore character (_) and the dollar sign ($) are considered letters and may be used as any character including the first one.

Scope

What is the scope of a Java variable?

The scope of a Java variable is defined by the *block of code* within which the variable is accessible.

(Briefly, a block of code consists of none, one, or more statements enclosed by a pair of matching curly brackets.)

The scope also determines when the variable is created *(memory set aside to contain the data stored in the variable)* and when it possibly becomes a candidate for destruction *(memory returned to the operating system for recycling and re-use)* .

Scope categories

The scope of a variable places it in one of the four categories shown in Figure 5 .

Figure 5 . Scope categories.

- member variable
- local variable
- method parameter
- exception handler parameter

Member variable

A member variable is a member of a class *(class variable)* or a member of an object instantiated from that class *(instance variable)* . It must be declared within a class, but not within the body of a method or constructor of the class.

Local variable

A local variable is a variable declared within the body of a method or constructor or within a block of code contained within the body of a method or constructor.

Method parameters

Method parameters are the formal arguments of a method. Method parameters are used to pass values into and out of methods. The scope of a method parameter is the entire method for which it is a parameter.

Exception handler parameters

Exception handler parameters are arguments to exception handlers. Exception handlers will be discussed in a future module.

Illustrating different types of variables in Java

The Java program shown in Listing 8 illustrates

- member variables *(class and instance)* ,
- local variables, and
- method parameters.

An illustration of exception handler parameters will be deferred until exception handlers are discussed in a future module.

Listing 8 . The program named member1.

```
/*File member1.java Copyright 1997, R.G.Baldwin
Illustrates class variables, instance
variables, local variables, and method parameters.

Output from this program is:

Class variable is 5
Instance variable is 6
Method parameter is 7
Local variable is 8

**********************************************************/
class member1 { //define the controlling class
  //declare and initialize class variable
  static int classVariable = 5;
  //declare and initialize instance variable
  int instanceVariable = 6;

  public static void main(String[] args){ //main method
    System.out.println("Class variable is "
                                        + classVariable);

    //Instantiate an object of the class to allow for
    // access to instance variable and method.
    member1 obj = new member1();
    System.out.println("Instance variable is "
                                    + obj.instanceVariable);
    obj.myMethod(7); //call the method

    //declare and intitialize a local variable
    int localVariable = 8;
    System.out.println("Local variable is "
                                        + localVariable);
```

Listing 8 . The program named member1.

```
  }//end main

  void myMethod(int methodParameter){
    System.out.println("Method parameter is "
                                       + methodParameter);
  }//end myMethod
}//End member1 class.
```

Declaration of local variables

In Java, local variables are declared within the body of a method or within a block of code contained within the body of a method.

Scope of local variables

The scope of a local variable extends from the point at which it is declared to the end of the block of code in which it is declared.

What is a "block" of code?

A block of code is defined by enclosing it within curly brackets as in { ... }.

Therefore, the scope of a local variable can be the entire method, or can reduced by declaring it within a block of code within the method.

Note: Special case, scope within a for loop
Java treats the scope of a variable declared within the initialization clause of a **for** statement to be limited to the total extent of the **for** statement.
A future module will explain what is meant by a **for** statement or a **for** loop.

Initialization of variables

Initializing primitive local variables

Local variables of primitive types can be initialized when they are declared using statements such the one shown in Listing 9 .

Listing 9 . Initialization of variables.

```
int MyVar, UrVar = 6, HisVar;
```

Initializing member variables

Member variables can also be initialized when they are declared.

In both cases, the type of the value used to initialize the variable must match the type of the variable.

Initializing method parameters and exception handler parameters

Method parameters and exception handler parameters are initialized by the values passed to the method or exception handler by the calling program.

Run the programs

I encourage you to copy the code from Listing 1 , Listing 6 , and Listing 8 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0200: Java OOP: Variables
- File: Jb0200.htm
- Originally published: 1997
- Published at cnx.org: 11/18/12

version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0200r: Review
This module contains review questions and answers keyed to the module titled Jb0200: Java OOP: Variables.

Revised: Mon Mar 28 13:31:46 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0200: Java OOP: Variables .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

Write a Java application that reads characters from the keyboard until encountering the # character. Echo each character to the screen as it is read. Terminate the program when the user enters the # character.

Answer 1

Question 2

What is the common name for the Java program element that is used to contain data that changes during the execution of the program?

Answer 2

Question 3

What must you do to make a variable available for use in a Java program?

Answer 3

Question 4

True or false? In Java, you are required to initialize the value of all variables when they are declared.

Answer 4

Question 5

Show the proper syntax for declaring two variables and initializing one of them using a single Java statement.

Answer 5

Question 6

True or false? The Java compiler will accept statements with type mismatches provided that a suitable type conversion can be implemented by the compiler at compile time.

Answer 6

Question 7

Show the proper syntax for the declaration of a variable of type **String[]** in the argument list of the **main** method of a Java program and explain its purpose.

Answer 7

Question 8

Describe the purpose of the type definition in Java.

Answer 8

Question 9

True or false? Variables of type **int** can contain either signed or unsigned values.

Answer 9

Question 10

What is the important characteristic of type definitions in Java that strongly supports the concept of *platform independence* of compiled Java programs?

Answer 10

Question 11

What are the two major categories of types in Java?

Answer 11

Question 12

What is the maximum number of values that can be stored in a variable of a *primitive* type in Java?

Answer 12

Question 13

List the *primitive* types in Java.

Answer 13

Question 14

True or false? Java stores variables of type **char** according to the 8-bit extended ASCII table.

Answer 14

Question 15

True or false? In Java, the name of a *primitive* variable evaluates to the value stored in the variable.

Answer 15

Question 16

True or false? Variables of *primitive* data types in Java are true objects.

Answer 16

Question 17

Why do we care that variables of *primitive* types are not true objects?

Answer 17

Question 18

What is the name of the mechanism commonly used to convert variables of *primitive* types to true objects?

Answer 18

Question 19

How can you tell the difference between a *primitive* type and a *wrapper* for the primitive type when the two are spelled the same?

Answer 19

Question 20

Show the proper syntax for declaring a variable of type **double** and initializing its value to 5.5.

Answer 20

Question 21

Show the proper syntax for declaring a variable of type **Double** and initializing its value to 5.5.

Answer 21

Question 22

Show the proper syntax for extracting the value from a variable of type **Double** .

Answer 22

Question 23

True or false? In Java, the name of a reference variable evaluates to the address of the location in memory where the variable is stored.

Answer 23

Question 24

What is a *legal identifier* in Java?

Answer 24

Question 25

What are the rules for variable names in Java?

Answer 25

Question 26

What is meant by the *scope* of a Java variable?

Answer 26

Question 27

What are the four possible *scope* categories for a Java variable?

Answer 27

Question 28

What is a member variable?

Answer 28

Question 29

Where are *local variables* declared in Java?

Answer 29

Question 30

What is the scope of a local variable in Java?

Answer 30

Question 31

What defines a *block* of code in Java?

Answer 31

Question 32

What is the scope of a variable that is declared within a block of code that is defined within a method and which is a subset of the statements that make up the method?

Answer 32

Question 33

What is the scope of a variable declared within the initialization clause of a *for* statement in Java? Provide an example code fragment.

Answer 33

Question 34

What are *method parameters* and what are they used for?

Answer 34

Question 35

What is the scope of a *method parameter* ?

Answer 35

Question 36

What are *exception handler parameters* ?

Answer 36

Question 37

Write a Java application that illustrates member variables *(class and instance)* , local variables, and method parameters.

Answer 37

Question 38

True or false? Member variables in a Java class can be initialized when the class is defined.

Answer 38

Question 39

How are *method parameters* initialized in Java?

Answer 39

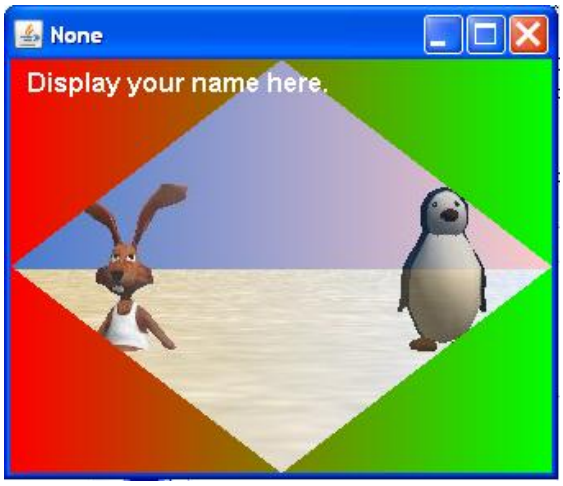
Listings

- Listing 1 . Listing for Answer 22.
- Listing 2 . Listing for Answer 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 39

Method parameters are initialized by the values passed to the method.

Back to Question 39

Answer 38

True.

Back to Question 38

Answer 37

See the application named **member1** in this module for an example of such an application.

Back to Question 37

Answer 36

Exception handler parameters are arguments to exception handlers, which will be discussed in a future module.

Back to Question 36

Answer 35

The scope of a method parameter is the entire method for which it is a parameter.

Back to Question 35

Answer 34

Method parameters are the formal arguments of a method. Method parameters are used to pass values into and out of methods.

Back to Question 34

Answer 33

Java treats the scope of a variable declared within the initialization clause of a *for* statement to be limited to the total extent of the *for* statement. A sample code fragment follows where **cnt** is the variable being discussed:

Note:

```
for(int cnt = 0; cnt < max; cnt++){
  //do something
}//end of
```

Back to Question 33

Answer 32

In Java, the scope can be reduced by placing it within a block of code within the method. The *scope* extends from the point at which it is declared to the end of the block of code in which it is declared.

Back to Question 32

Answer 31

A block of code is defined by enclosing it within curly brackets as shown below

{ ... } .

Back to Question 31

Answer 30

The *scope* of a local variable extends from the point at which it is declared to the end of the block of code in which it is declared.

Back to Question 30

Answer 29

In Java, *local variables* are declared within the body of a method or constructor, or within a block of code contained within the body of a method or constructor.

Back to Question 29

Answer 28

A *member variable* is a member of a class (*class* variable) or a member of an object instantiated from that class (*instance* variable). It must be declared within a class, but not within the body of a method or constructor of the class.

Back to Question 28

Answer 27

The *scope* of a variable places it in one of the following four categories:

- member variable
- local variable
- method parameter
- exception handler parameter

Back to Question 27

Answer 26

The *scope* of a Java variable is the block of code within which the variable is accessible.

Back to Question 26

Answer 25

The rules for Java variable names are as follows:

- Must be a legal Java identifier consisting of a series of *Unicode* characters.
- Must not be the same as a Java *keyword* and must not be *true* or *false*.
- Must not be the same as another variable whose declaration appears in the same scope.

Back to Question 25

Answer 24

In Java, a legal identifier is a sequence of Unicode letters and digits of unlimited length. The first character must be a letter. All subsequent characters must be letters or numerals from any alphabet that Unicode supports. In addition, the underscore character (**_**) and the dollar sign (**$**) are considered letters and may be used as any character including the first one.

Back to Question 24

Answer 23

False. The name of a reference variable evaluates to either null, or to information that can be used to access an object whose reference has been stored in the variable.

Back to Question 23

Answer 22

Later versions of Java support either syntax shown in Listing 1 .

Listing 1 . Listing for Answer 22.

```
class test{
  public static void main(String[] args){
    Double var1 = 5.5;
    double var2 = var1.doubleValue();
    System.out.println(var2);

    double var3 = var1;
    System.out.println(var3);
  }//end main
}//end class test
```

Back to Question 22

Answer 21

The proper syntax for early versions of Java is shown below. Note the upper-case **D** . Also note the instantiation of a new object of type **Double** .

Note:

```
Double myWrappedData = new Double(5.5);
```

Later versions of Java support the following syntax with the new object of type **Double** being instantiated automatically:

Note:

```
Double myWrappedData = 5.5;
```

Back to Question 21

Answer 20

The proper syntax is shown below. Note the lower-case **d** .

Note:

```
double myPrimitiveData = 5.5;
```

Back to Question 20

Answer 19

The name of the *primitive* type begins with a lower-case letter and the name of the *wrapper* type begins with an upper-case letter such as **double** and **Double** . Note that in some cases, however, that they are not spelled the same. For example, the **Integer** class is the wrapper for type **int** .

Back to Question 19

Answer 18

Wrapper classes

Back to Question 18

Answer 17

This has some ramifications as to how variables can be used *(passing to methods, returning from methods, etc.)* . For example, all variables of *primitive* types are passed by value to methods meaning that the code in the method only has access to a copy of the variable and does not have the ability to modify the variable.

Back to Question 17

Answer 16

False. Primitive data types in Java *(int, double, etc.)* are not true objects.

Back to Question 16

Answer 15

True.

Back to Question 15

Answer 14

False. The **char** type in Java is a 16-bit Unicode character.

Back to Question 14

Answer 13

- byte
- short
- int
- long
- float
- double
- char
- boolean

Back to Question 13

Answer 12

Primitive types contain a single value.

Back to Question 12

Answer 11

Java supports both *primitive* types and *reference* (or object) types.

Back to Question 11

Answer 10

In Java, a variable of a specified type is represented exactly the same way regardless of the platform on which the application or applet is being executed.

Back to Question 10

Answer 9

False. In Java, all variables of type **int** contain signed values.

Back to Question 9

Answer 8

All variables in Java must have a defined *type* . The definition of the *type* determines the set of values that can be stored in the variable and the operations that can be performed on the variable.

Back to Question 8

Answer 7

The syntax is shown in boldface below:

Note:
public static void main(**String[] args**)

In this case, the type of variable declared is an array of type **String** named **args** *(type String[])* . The purpose of the **String** array variable in the argument list is to make it possible to capture arguments entered on the command line.

Back to Question 7

Answer 6

False. Fortunately, Java provides very strict type checking and generally refuses to compile statements with type mismatches.

Back to Question 6

Answer 5

Note:

```
int firstVariable, secondVariable = 10;
```

Back to Question 5

Answer 4

False: In Java, it is possible to initialize the value of a variable when it is declared, but initialization is not required. *(Note however that in some situations, the usage of the variable may require that it be purposely initialized.)* .

Back to Question 4

Answer 3

To use a variable, you must notify the compiler of the name and the type of the variable *(declare the variable).*

Back to Question 3

Answer 2

variable

Back to Question 2

Answer 1

Listing 2 . Listing for Answer 1.

```
/*File simple4.java
This application reads characters from the keyboard until
encountering the # character and echoes each character to
the screen. The program terminates when the user enters
the # character.
**********************************************************/
class simple4 { //define the controlling class
  public static void main(String[] args)
                                  throws java.io.IOException {
    int ch1 = 0;
    System.out.println(
                       "Enter some text, terminate with #");
    while( (ch1 = System.in.read() ) != '#')
      System.out.print((char)ch1);
    System.out.println("Goodbye");
  }//end main
}//End simple4 class.
```

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0200r: Review: Variables
- File: Jb0200r.htm
- Published: 11/23/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0210: Java OOP: Operators
Earlier modules have touched briefly on the topic of operators. This module discusses Java operators in depth.

Revised: Mon Mar 28 13:56:51 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

Earlier modules have touched briefly on the topic of **operators** . This module discusses Java **operators** in depth.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

- Listing 1 . Illustration of prefix and postfix notation.
- Listing 2 . Illustration of relational operators.

Introduction

The first step in learning to use a new programming language is usually to learn the foundation concepts such as

- variables,
- operators,
- types,
- expressions,
- flow-of-control, etc.

This module concentrates on the **operators** used in Java.

Operators

Unary and binary operators

Java provides a set of operators that can be used to perform an action on one, two, or three operands. An operator that operates on one operand is called a *unary* operator. An operator that operates on two operands is called a *binary* operator. An operator that operates on three operands is called a *ternary* operator.

Some operators can behave either as a unary or as a binary operator. The best known such operator is probably the minus sign *(-)* . As a binary operator, the minus sign causes its right operand to be subtracted from its left operand. As a unary operator, the minus sign causes the algebraic sign of the right operand to be changed.

A ternary operator

Java has only one operator that takes three operands. It is a conditional operator, which I sometimes refer to as a cheap **if** statement.

The first operand is a **boolean** expression, which is followed by a question mark character *(?)* . The question mark is followed by a second operand, which is followed by a colon character *(:)* . The colon character is followed by the third operand.

If the **boolean** expression evaluates to true, the value of the operand following the ? is returned. Otherwise, the value of the operand following the : is returned.

An example of the syntax follows:

Note: Ternary operator syntax
boolean expression ? value1 : value2

Overloaded operators

Unlike C++, Java does not support the creation of overloaded operators in program code. *(If you don't know what this means, don't worry about it.)*

Operators from previous programs

The statements in the following note box illustrate the use of the following operators from Java **programs in earlier modules** :

- =
- !=
- +
- (char)

Note: Operators from previous programs

```
int ch1, ch2 = '0';
while( (ch1 = System.in.read() ) != '#') ch2 = ch1;
System.out.println("The char before the # was "
                                        + (char)ch2);
```

The plus and cast operators

Of particular interest in this list is the plus sign *(+)* and the cast operator *(char)* .

In Java, the plus sign can be used to perform arithmetic addition. It can also be used to concatenate strings. When the plus sign is used in the manner shown above , the operand on the right is automatically converted to a character string before being concatenated with the operand on the left.

The cast operator is used in this case to purposely convert the integer value contained in the **int** variable **ch2** to a character type suitable for concatenating with the string on the left of the plus sign. Otherwise, Java would attempt to convert and display the value of the **int**

variable as a series of digits representing the *numeric value* of the character because the character is stored in a variable of type **int** .

The increment operator

An extremely important *unary* operator is the increment operator identified by two plus characters with no space between them *(++)* .

The increment operator causes the value of its operand to be increased by one.

Note: The decrement operator
There is also a decrement operator *(--)* that causes the value of its operand to be decreased by one.

The increment and decrement operators are used in both *prefix* and *postfix* notation.

Prefix and postfix increment and decrement operators

With the *prefix* version, the operand appears to the right of the operator *(++X)* , while with the *postfix* version, the operand appears to the left of the operator *(X++)* .

What's the difference in prefix and postfix?

The difference in prefix and postfix has to do with the point in the sequence of operations that the increment *(or decrement)* actually occurs if the operator and its operand appear as part of a larger overall expression.

(There is effectively no difference if the operator and its operand do not appear as part of a larger overall expression.)

Prefix behavior

With the *prefix* version, the variable is incremented *(or decremented)* before it is used to evaluate the larger overall expression.

Postfix behavior

With the *postfix* version, the variable is used to evaluate the larger overall expression before it is incremented *(or decremented)* .

Illustration of prefix and postfix behavior

The use of both the *prefix* and *postfix* versions of the increment operator is illustrated in the Java program shown in Listing 1 . The output produced by the program is show in the comments at the beginning of the program.

Listing 1 . Illustration of prefix and postfix notation.

Listing 1 . Illustration of prefix and postfix notation.

```
/*File incr01.java Copyright 1997, n
Illustrates the use of the prefix and the postfix
increment
operator.

The output from the program follows:

a = 5
b = 5
a + b++ = 10
b = 6

c = 5
d = 5
c + ++d = 11
d = 6

*********************************************************/
class incr01 { //define the controlling class
  public static void main(String[] args){ //main method
    int a = 5, b = 5, c = 5, d = 5;
    System.out.println("a = " + a );
    System.out.println("b = " + b );
    System.out.println("a + b++ = " + (a + b++) );
    System.out.println("b = " + b );
    System.out.println();

    System.out.println("c = " + c );
    System.out.println("d = " + d );
    System.out.println("c + ++d = " + (c + ++d) );
    System.out.println("d = " + d );
  }//end main
}//End incr01 class.
```

Binary operators and infix notation

Binary operators use *infix* notation, which means that the operator appears between its operands.

General behavior of an operator

As a result of performing the specified action, an operator can be said to return a value *(or evaluate to a value)* of a given type. The type of value returned depends on the operator and the type of the operands.

Note: Evaluating to a value
To evaluate to a value means that after the action is performed, the operator and its operands are effectively replaced in the expression by the value that is returned.

Operator categories

I will divide Java's operators into the following categories for further discussion:

- arithmetic operators
- relational and conditional *(logical)* operators
- bitwise operators
- assignment operators

Arithmetic operators

Java supports various arithmetic operators on all floating point and integer numbers.

The binary arithmetic operators

The following table lists the *binary* arithmetic operators supported by Java.

Note: The binary arithmetic operators

```
Operator    Description

   +        Adds its operands
   -        Subtracts the right operand from the left
            operand
   *        Multiplies the operands
   /        Divides the left operand by the right operand
   %        Remainder of dividing the left operand by
            the right operand
```

String concatenation

As mentioned earlier, the plus operator *(+)* is also used to concatenate strings as in the following code fragment:

Note: String concatenation

```
"MyVariable has a value of  "
                      + MyVariable + " in this program."
```

Coercion

Note that this operation also coerces the value of **MyVariable** to a string representation for use in the expression only. However, the value stored in the variable is not modified in any lasting way.

Unary arithmetic operators

Java supports the following *unary* arithmetic operators.

Note:
Unary arithmetic operators

```
Operator    Description

   +        Indicates a positive value
   -        Negates, or changes algebraic sign
   ++       Adds one to the operand,
            both prefix and postfix
   --       Subtracts one from operand,
            both prefix and postfix
```

The result of the increment and decrement operators being either *prefix* or *postfix* was discussed earlier .

Relational and conditional (logical) operators

Binary Relational operators

Java supports the set of *binary* relational operators shown in the following table. Relational operators in Java return either *true* or *false* as a **boolean** type.

Note: Binary Relational operators

```
Operator    Returns true if
   >        Left operand is greater than right operand
   >=       Left operand is greater than or equal to
            right operand
   <        Left operand is less than right operand
   <=       Left operand is less than or equal to
            right operand
   ==       Left operand is equal to right operand
   !=       Left operand is not equal to right operand
```

Conditional expressions

Relational operators are frequently used in the conditional expressions of control statement such as the one in the code fragment shown below.

Note: Conditional expressions

```
if(LeftVariable <= RightVariable). . .
```

Illustration of relational operators

The program shown in Listing 2 illustrates the result of applying relational operators in Java. The output is shown in the comments at the beginning of the program. Note that the program automatically displays **true** and **false** as a result of applying the relational operators.

Listing 2 . Illustration of relational operators.

```
/*File relat01.java Copyright 1997, R.G.Baldwin
Illustrates relational operators.

Output is

The relational 6<5 is false
The relational 6>5 is true

**************************************************************/
class relat01 { //define the controlling class
  public static void main(String[] args){ //main method
    System.out.println("The relational 6<5 is "
      +(6<5));
    System.out.println("The relational 6>5 is "
      +(6>5));
  }//end main
}//End relat01 class.
```

Conditional operators

The relational operators are often combined with another set of operators *(referred to as conditional or logical operators)* to construct more complex expressions.

Java supports three such operators as shown in the following table.

Note: Conditional or logical operators

```
Operator  Typical Use             Returns true if
```

```
  &&     Left && Right     Left and Right are both true
  ||     Left || Right     Either Left or Right is true
  !      ! Right           Right is false
```

The operands shown in the table must be **boolean** types, or must have been created by the evaluation of an expression that returns a **boolean** type.

Left to right evaluation

An important characteristic of the behavior of the logical ***and*** and the logical ***or*** operators is that the expressions are evaluated from left to right, and the evaluation of the expression is terminated as soon as the result of evaluating the expression can be determined.

For example, in the following expression, if the variable **a** is less than the variable **b** , there is no need to evaluate the right operand of the || to determine that the result of evaluating the entire expression would be **true** . Therefore, evaluation will terminate as soon as the answer can be determined.

Note: Left to right evaluation

```
(a < b) || (c < d)
```

Don't confuse bitwise and with logical and

As discussed in the next section, the symbols shown below are the bitwise ***and*** and the bitwise ***or*** .

Note: Bitwise *and* and bitwise *or*

```
& bitwise and
| bitwise or
```

One author states that in Java, the bitwise ***and*** operator can be used as a synonym for the logical ***and*** and the bitwise ***or*** can be used as a synonym for the logical ***inclusive or*** if both of the operands are **boolean** . *(I recommend that you don't do that because it could cause confusion for someone reading your code.)*

Note however that according to a different author, in this case, the evaluation of the expression is not terminated until all operands have been evaluated, thus eliminating the possible advantage of the left-to-right evaluation.

Bitwise operators

Java provides a set of operators that perform actions on their operands one bit at a time as shown in the following table.

Note: Bitwise operators

```
Operator   Typical Use         Operation

 >>     OpLeft >> Dist      Shift bits of OpLeft right
                            by Dist bits (signed)
 <<     OpLeft << Dist      Shift bits of OpLeft left
                            by Dist bits
 >>>    OpLeft >>> Dist     Shift bits of OpLeft right
                            by Dist bits (unsigned)
 &      OpLeft & OpRight    Bitwise and of the
                            two operands
 |      OpLeft | OpRight    Bitwise
```

Populating vacated bits for shift operations

The *signed* right shift operation populates the vacated bits with the sign bit, while the left shift and the *unsigned* right shift populate the vacated bits with zeros.

In all cases, bits shifted off the end are lost.

The rule for bitwise *and*

The bitwise **and** operation operates according to the rule that the bitwise **and** of two 1 bits is a 1 bit.

Any other combination results in a 0 bit.

Bitwise inclusive *or*

For the *inclusive or* , if either bit is a 1, the result is a 1.

Otherwise, the result is a 0.

Bitwise exclusive *or*

For the *exclusive or* , if either but not both bits is a 1, the result is a 1.

Otherwise, the result is a 0.

Another way to state this is if the bits are different, the result is a 1. If the two bits are the same, the result is a 0.

The complement operator

Finally, the complement operator changes each 1 to a 0 and changes each 0 to a 1.

Assignment operators

Simple assignment operator

The *(=)* is a value assigning *binary* operator in Java. The value stored in memory and represented by the right operand is copied into the memory represented by the left operand.

Using the assignment operator with reference variables

You need to be careful and think about what you are doing when you use the assignment operator with reference variables in Java. If you assign one reference variable to another, you simply end up with two reference variables that refer to the same object. You do not end up with two different objects.

*(If what you need is another copy of the object, you may be able to use the **clone** method to accomplish that.)*

Shortcut assignment operators

Java supports the following list of *shortcut* assignment operators. These operators allow you to perform an assignment and another operation with a single operator.

Note: Shortcut assignment operators

```
+=
-=
*=
/=
%=
&=
|=
^=
<<=
>>=
>>>=
```

For example, the two statements that follow perform the same operation.

Note: Illustration of shortcut assignment operation

```
x += y;
x = x + y;
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0210: Java OOP: Operators
- File: Jb0210
- Originally published: 1997
- Published at cnx.org: 11/23/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0210r Review
This module contains review questions and answers keyed to the module titled Jb0210: Java OOP: Operators.

Revised: Mon Mar 28 14:04:36 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0210: Java OOP: Operators .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

An operator performs an action on what? Provide the name.

Answer 1

Question 2

What do we call an operator that operates on only one operand?

Answer 2

Question 3

What do we call an operator that operates on two operands?

Answer 3

Question 4

Is the minus sign a *unary* or a *binary* operator, or both? Explain your answer.

Answer 4

Question 5

Describe operator overloading.

Answer 5

Question 6

True or false? Java programmers may overload operators.

Answer 6

Question 7

Show the symbols used for the following operators in Java: *assignment* , *not equal* , *addition* , *cast* .

Answer 7

Question 8

Are any operators automatically overloaded in Java? If so, identify one and describe its overloaded behavior.

Answer 8

Question 9

What is the purpose of the cast operator?

Answer 9

Question 10

True or false? The increment operator is a *binary* operator.

Answer 10

Question 11

Show the symbol for the increment operator.

Answer 11

Question 12

Describe the appearance and the behavior of the increment operator with both *prefix* and *postfix* notation. Show example, possibly incomplete, code fragments illustrating both notational forms.

Answer 12

Question 13

Show the output that would be produced by the Java application in Listing 1.

Listing 1 . Listing for Question 13.

```
class incr01 { //define the controlling class
  public static void main(String[] args){ //define main
    int x = 5, X = 5, y = 5, Y = 5;
    System.out.println("x = " + x );
    System.out.println("X = " + X );
    System.out.println("x + X++ = " + (x + X++) );
    System.out.println("X = " + X );
    System.out.println();
    System.out.println("y = " + y );
    System.out.println("Y = " + Y );
    System.out.println("y + ++Y = " + (y + ++Y) );
    System.out.println("Y = " + Y );
  }//end main
}//End incr01 class.  Note no semicolon required
//End Java application
```

Answer 13

Question 14

True or false? *Binary* operators use *outfix* notation. If your answer is False, explain why.

Answer 14

Question 15

In practice, what does it mean to say that an operator that has performed an action returns a value *(or evaluates to a value)* of a given type?

Answer 15

Question 16

Show and describe at least five of the *binary arithmetic* operators supported by Java (Clarification: *binary* operators does not mean *bitwise* operators).

Answer 16

Question 17

In addition to arithmetic addition, what is another use for the plus operator *(+)* ? Show an example code fragment to illustrate your answer. The code fragment need not be a complete statement.

Answer 17

Question 18

When the plus operator *(+)* is used as a concatenation operator, what is the nature of its behavior if its left operand is of type **String** and its right operand is not of type **String** ? If the right operand is a variable that is not of type **String** , what is the impact of this behavior on that variable.

Answer 18

Question 19

Show and describe four *unary* arithmetic operators supported by Java.

Answer 19

Question 20

What is the type returned by *relational* operators in Java?

Answer 20

Question 21

Show and describe six different *relational* operators supported by Java.

Answer 21

Question 22

Show the output that would be produced by the Java application shown in Listing 2 .

Listing 2 . Listing for Question 22.

```
class relat01 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("The relational 6<5 is " + (6<5
));
    System.out.println("The relational 6>5 is " + (6>5
));
  }//end main
}//End relat01 class.  Note no semicolon required
//End Java application
```

Answer 22

Question 23

Show and describe three operators *(frequently referred to as conditional or logical operators)* that are often combined with relational operators to construct more complex expressions *(often called conditional expressions)* . Hint: The || operator returns true if either the left operand, the right operand, or both operands are true. What are the other two and how do they behave?

Answer 23

Question 24

Describe the special behavior of the || operator in the following expression for the case where the value of the variable **a** is less than the value of the variable **b** .

Note:

```
(a < b) || (c < d)
```

Answer 24

Question 25

Show the symbols used for the bitwise *and* operator and the bitwise *inclusive or* operator.

Answer 25

Question 26

Show and describe seven operators in Java that perform actions on the operands one bit at a time *(bitwise operators)* .

Answer 26

Question 27

True or false? In Java, the *signed* right shift operation populates the vacated bits with the zeros, while the left shift and the *unsigned* right shift populate the vacated bits with the sign bit. If your answer is False, explain why.

Answer 27

Question 28

True or false? In a *signed* right-shift operation in Java, the bits shifted off the right end are lost. If your answer is False, explain why.

Answer 28

Question 29

Using the symbols 1 and 0, construct a truth table showing the four possible combinations of 1 and 0. Using a 1 or a 0, show the result of the *bitwise and* operation on these four combinations of 1 and 0.

Answer 29

Question 30

Using the symbols 1 and 0 construct a truth table showing the four possible combinations of 1 and 0. Using a 1 or a 0, show the result of the *bitwise inclusive or* operation on these four combinations of 1 and 0.

Answer 30

Question 31

Using the symbols 1 and 0 construct a truth table showing the four possible combinations of 1 and 0. Using a 1 or a 0, show the result of the *bitwise exclusive or* operation on these four combinations of 1 and 0.

Answer 31

Question 32

True or false? For the *exclusive or* , if the two bits are different, the result is a 1. If the two bits are the same, the result is a 0. If your answer is False, explain why.

Answer 32

Question 33

Is the *assignment* operator a *unary* operator or a *binary* operator. Select one or the other.

Answer 33

Question 34

True or false? In Java, when using the assignment operator, the value stored in memory and represented by the right operand is copied into the memory represented by the left operand. If your answer is False, explain why.

Answer 34

Question 35

Show two of the *shortcut* assignment operators and explain how they behave by comparing them with the regular *(non-shortcut)* versions. Hint: the *(^=)* operator is a *shortcut* assignment operator.

Answer 35

Question 36

Write a Java application that clearly illustrates the difference between the prefix and the postfix versions of the increment operator. Provide a termination message that displays your name.

Answer 36

Question 37

Write a Java application that illustrates the use of the following relational operators:

Note:

<
>
<=
>=
==
!=

Provide appropriate text in the output. Also provide a termination message with your name.

Answer 37

Question 38

write a Java application that illustrates the use of the following logical or conditional operators:

Note: Logical or conditional operators

```
&&
||
!
```

Provide appropriate text in the output. Also provide a termination message with your name.

Answer 38

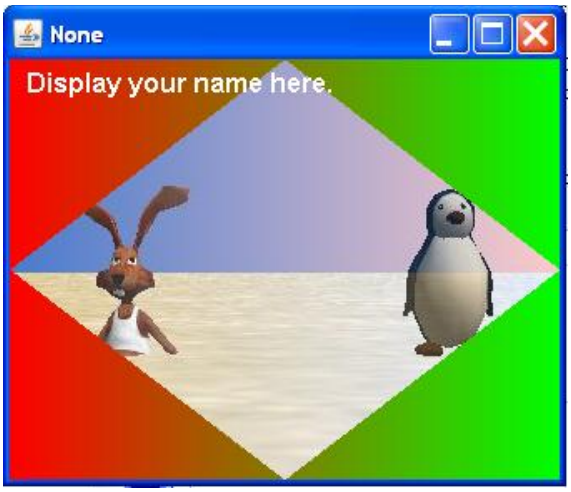
Listings

- Listing 1 . Listing for Question 13.
- Listing 2 . Listing for Question 22.
- Listing 3 . Listing for Answer 38.
- Listing 4 . Listing for Answer 37.
- Listing 5 . Listing for Answer 36.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 38

Listing 3 . Listing for Answer 38.

Listing 3 . Listing for Answer 38.

```
/*File SampProg09.java from module 22
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that illustrates the use of the following
logical or conditional operators:

&&  ||  !

Provide appropriate text in the output.  Also provide
a termination message with your name.
**************************************************/
class SampProg09 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("true and true is "
      + (true && true) );
    System.out.println("true and false is "
      + (true && false) );

    System.out.println("true or true is "
      + (true || true) );
    System.out.println("true or false is "
      + (true || false) );
    System.out.println("false or false is "
      + (false || false) );

    System.out.println("not true is " + (! true) );
    System.out.println("not false is " + (! false) );

    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//End SampProg09 class.  Note no semicolon required
```

Back to Question 38

Answer 37

Listing 4 . Listing for Answer 37.

```
/*File SampProg08.java from module 22
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that illustrates the use of the following
relational operators:

< > <= >= == !=

Provide appropriate text in the output.  Also provide
a termination message with your name.
**************************************************/
class SampProg08 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("The relational 6<5 is "
                                          + (6<5 ) );
    System.out.println("The relational 6>5 is "
                                          + (6>5 ) );
    System.out.println("The relational 5>=5 is "
                                         + (5>=5 ) );
    System.out.println("The relational 5<=5 is "
                                         + (5<=5 ) );
    System.out.println("The relational 6==5 is "
                                            + (6==5 )
);
    System.out.println("The relational 6!=5 is "
                                            + (6!=5 )
);
    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//End SampProg08 class.  Note no semicolon required
```

Back to Question 37

Answer 36

Listing 5 . Listing for Answer 36.

```
/*File SampProg07.java from module 22
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that clearly illustrates the difference
between
the prefix and the postfix versions of the increment
operator.

Provide a termination message that displays your name.
**************************************************************/
class SampProg07{
  static public void main(String[] args){
    int x = 3;
    int y = 3;
    int z = 10;
    System.out.println("Prefix version gives  "
                                          + (z + ++x));
    System.out.println("Postfix version gives "
                                          + (z + y++));
    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//end class SampProg07
```

Back to Question 36

Answer 35

Java supports the following list of *shortcut* assignment operators. These operators allow you to perform an assignment and another operation with a single operator.

Note:

```
+=
-=
*=
```

```
/=
%=
&=
|=
^=
<<=
>>=
>>>=
```

For example, the two statements that follow perform the same operation.

- **x += y;**
- **x = x + y;**

The behavior of each of the *shortcut* assignment operators follows this same pattern.

Back to Question 35

Answer 34

True.

Back to Question 34

Answer 33

The assignment operator is a *binary* operator.

Back to Question 33

Answer 32

True.

Back to Question 32

Answer 31

The answer for the bitwise *exclusive or* is:

- 11 1 xor 1 produces 0
- 10 1 xor 0 produces 1
- 01 0 xor 1 produces 1
- 00 0 xor 0 produces 0

Back to Question 31

Answer 30

The answer for the bitwise *inclusive or* is:

- 11 1 or 1 produces 1
- 10 1 or 0 produces 1
- 01 0 or 1 produces 1
- 00 0 or 0 produces 0

Back to Question 30

Answer 29

The answer for the bitwise ***and*** is:

- 11 1 and 1 produces 1
- 10 1 and 0 produces 0
- 01 0 and 1 produces 0
- 00 0 and 0 produces 0

Back to Question 29

Answer 28

True: Bits shifted off the right end are lost.

Back to Question 28

Answer 27

False: In Java, the *signed* right shift operation populates the vacated bits with the sign bit, while the left shift and the *unsigned* right shift populate the vacated bits with zeros.

Back to Question 27

Answer 26

The following table shows the seven bitwise operators supported by Java.

Note: Bitwise operators

Operator	Typical Use	Operation
>>	OpLeft >> Dist	Shift bits of OpLeft right by Dist bits (signed)
<<	OpLeft << Dist	Shift bits of OpLeft left by Dist bits
>>>	OpLeft >>> Dist	Shift bits of OpLeft right by Dist bits (unsigned)
&	OpLeft & OpRight	Bitwise and of the two operands
|	OpLeft | OpRight	Bitwise

Back to Question 26

Answer 25

The bitwise ***and*** operator and the bitwise ***inclusive or*** operator are shown below.

Note: Two bitwise operators

```
& bitwise and
| bitwise inclusive or
```

Back to Question 25

Answer 24

An important characteristic of the behavior of the ***logical and*** operator and the ***logical or*** operator in Java is that the expressions containing them are evaluated from *left to right.* The evaluation of the expression is. terminated as soon as the result of evaluating the expression can be determined. For example, in the expression given in Question 24 , if the variable **a** is less than the variable **b** , there is no need to evaluate the right operand of the || operator to determine the value of the entire expression. Therefore, evaluation will terminate as soon as it is determined that **a** is less than **b** .

Back to Question 24

Answer 23

The following three *logical* or *conditional* operators are supported by Java.

Note: The logical or conditional operators

```
Operator  Typical Use       Returns true if
&&        Left && Right     Left and Right are both true
||        Left || Right     Either Left or Right is true
!         ! Right           Right is false
```

Back to Question 23

Answer 22

This program produces the following output:

Note:

```
The relational 6<5 is false
The relational 6>5 is true
```

Back to Question 22

Answer 21

Java supports the following set of *relational* operators:

Note: Relational operators

```
Operator    Returns true if
   >        Left operand is greater than right operand
   >=       Left operand is greater than or equal
            to right operand
   <        Left operand is less than right operand
   <=       Left operand is less than or equal
            to right operand
   ==       Left operand is equal to right operand
   !=       Left operand is not equal to right operand
```

Back to Question 21

Answer 20

Relational operators return the **boolean** type in Java.

Back to Question 20

Answer 19

Java supports the following four *unary* arithmetic operators.

Note: Unary arithmetic operators

```
Operator    Description
   +        Indicates a positive value
   -        Negates, or changes algebraic sign
   ++       Adds one to the operand,
            both prefix and postfix
   --       Subtracts one from operand,
            both prefix and postfix
```

Back to Question 19

Answer 18

The operator coerces the value of the right operand to a string representation for use in the expression only. If the right operand is a variable, the value stored in the variable is not modified in any way.

Back to Question 18

Answer 17

The plus operator *(+)* is also used to concatenate strings as in the following code fragment:

Note: String concatenation

```
"MyVariable has a value of  "
     + MyVariable + " in this program."
```

Back to Question 17

Answer 16

Java support various arithmetic operators on floating point and integer numbers. The following table lists five of the *binary* arithmetic operators supported by Java.

Note: Binary arithmetic operators

```
Operator    Description

   +        Adds its operands
   -        Subtracts the right operand from the left
            operand
   *        Multiplies the operands
   /        Divides the left operand by the right
            operand
   %        Remainder of dividing the left operand by
            the right operand
```

Back to Question 16

Answer 15

As a result of performing the specified action, an operator can be said to return a value *(or evaluate to a value)* of a given type. The type depends on the operator and the type of the operands. To *evaluate to a value* means that after the action is performed, the operator and its operands are effectively replaced in the expression by the value that is returned.

Back to Question 15

Answer 14

False: *Binary* operators use *infix* notation, which means that the operator appears between its operands.

Back to Question 14

Answer 13

The output from this Java application follows:

- x = 5
- X = 5
- x + X++ = 10
- X = 6
- y = 5
- Y = 5
- y + ++Y = 11
- Y = 6

Back to Question 13

Answer 12

The increment operator may be used with both *prefix* and *postfix* notation. Basically, the increment operator causes the value of the variable to which it is applied to be increased by one.

With *prefix* notation, the operand appears to the right of the operator *(++X)* , while with *postfix* notation, the operand appears to the left of the operator *(X++)* .

The difference in behavior has to do with the point in the sequence of operations that the increment actually occurs.

With the *prefix* version, the variable is incremented before it is used to evaluate the larger overall expression in which it appears. With the *postfix* version, the variable is used to evaluate the larger overall expression and then the variable is incremented.

Back to Question 12

Answer 11

The symbol for the increment operator is two plus signs with no space between them *(++)*.

Back to Question 11

Answer 10

False: The increment operator is a *unary* operator.

Back to Question 10

Answer 9

The cast operator is used to purposely convert from one type to another.

Back to Question 9

Answer 8

The plus sign *(+)* is automatically overloaded in Java. The plus sign can be used to perform arithmetic addition. It can also be used to concatenate strings. However, the plus sign does more than concatenate strings. It also performs a conversion to **String** type. When the plus sign is used to concatenate strings and one operand is a string, the other operand is automatically converted to a character string before being concatenated with the existing string.

Back to Question 8

Answer 7

The operators listed in order are:

- =
- !=
- +
- (char)

where the cast operator is being used to cast to the type **char** .

Back to Question 7

Answer 6

Java does not support operator overloading by programmers.

Back to Question 6

Answer 5

For those languages that support it *(such as C++)* operator overloading means that the programmer can redefine the behavior of an operator with respect to objects of a new type defined by that program.

Back to Question 5

Answer 4

Both. As a *binary* operator, the minus sign causes its right operand to be subtracted from its left operand. As a *unary* operator, the minus sign causes the algebraic sign of the right operand to be changed.

Back to Question 4

Answer 3

An operator that operates on two operands is called a *binary* operator.

Back to Question 3

Answer 2

An operator that operates on only one operand is called a *unary* operator.

Back to Question 2

Answer 1

An operator performs an action on one or two operands.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0210r Review: Operators
- File: Jb0210r.htm
- Originally published: 1997
- Published at cnx.org: 11/23/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0220: Java OOP: Statements and Expressions
Java programs are composed of statements, and statements are constructed from expressions. This module takes a very brief look at Java statements and expressions.

Revised: Mon Mar 28 14:22:16 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

Java programs are composed of statements, and statements are constructed from expressions. This module takes a very brief look at Java statements and expressions.

Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, etc. This module concentrates on expressions and statements.

Expressions

The hierarchy

Java programs are composed of statements, and statements are constructed from expressions.

An expression is a specific combination of operators and operands, that evaluates to a single value. The operands can be variables, constants, or method calls.

A method call evaluates to the value returned by the method.

Named constants

Java supports named constants that are implemented through the use of the **final** keyword.

The syntax for creating a named constant in Java is as follows:

Note: Named constants

```
final float PI = 3.14159;
```

While this is not a constant type, it does produce a value that can be referenced in the program and which cannot be modified.

The **final** keyword prevents the value of **PI** from being modified in this case . You will learn later that there are some other uses for the **final** keyword in Java as well.

Operator precedence

The order in which the operations are performed determines the result. You can control the order of evaluation by the use of matching parentheses.

If you don't provide such parentheses, the order will be determined by the precedence of the operators *(you should find and review a table of Java operator precedence)* with the operations having higher precedence being evaluated first.

Statements

According to The Java Tutorials , *"A statement forms a complete unit of execution."*

A statement is constructed by combining one or more expressions into a compound expression and terminating that expression with a semicolon.

Further reading

As of November 2012, a good tutorial on this topic is available on the Oracle website titled Expressions, Statements, and Blocks .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0220: Java OOP: Statements and Expressions
- File: Jb0220.htm
- Originally published: 1997
- Published at cnx.org: 11/24/12

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0220r Review
This module contains review questions and answers keyed to the module titled Jb0220: Java OOP: Statements and Expressions.

Revised: Mon Mar 28 14:27:27 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0220: Java OOP: Statements and Expressions .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

A Java program is composed of a series of what?

Answer 1

Question 2

Statements in Java are constructed from what?

Answer 2

Question 3

Describe an expression in Java.

Answer 3

Question 4

What does a method call evaluate to in Java?

Answer 4

Question 5

True or false? Java supports named constants. If false, explain why.

Answer 5

Question 6

Provide a code fragment that illustrates the syntax for creating a named constant in Java.

Answer 6

Question 7

True or false? Java supports a **constant** type. If false, explain why.

Answer 7

Question 8

What is the common method of controlling the order of evaluation of expressions in Java?

Answer 8

Question 9

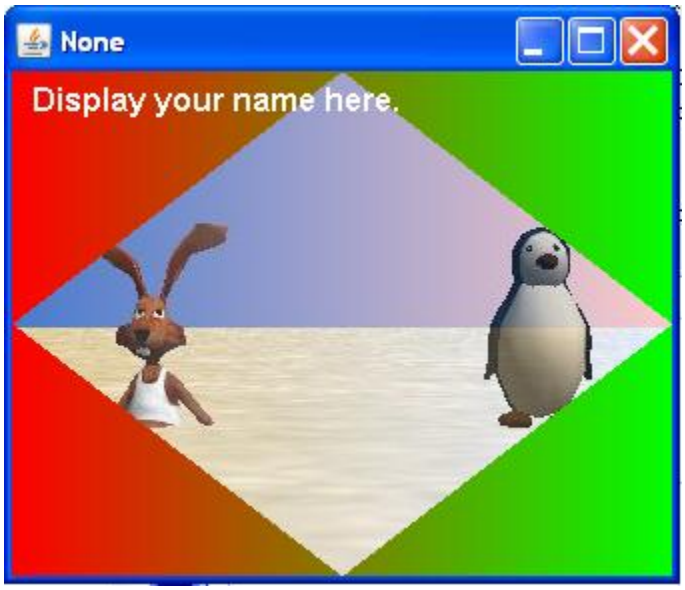
If you don't use matching parentheses to control the order of evaluation of expressions, what is it that controls the order of evaluation?

Answer 9

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 9

If you don't provide matching parentheses to control the order of evaluation, the order will be determined by the precedence of the operators with the

operations having higher precedence being evaluated first. For example, multiply and divide have higher precedence than add and subtract.

Back to Question 9

Answer 8

You can control the order of evaluation by the use of matching parentheses.

Back to Question 8

Answer 7

False. Java does not support a constant type. However, in Java, it is possible to achieve the same result by declaring and initializing a variable and making it **final** .

Back to Question 7

Answer 6

The syntax for creating a named constant in Java is shown below.

Note: A named constant in Java

```
final float PI = 3.14159;
```

Back to Question 6

Answer 5

True. Java supports named constants that are constructed using variable declarations with the **final** keyword.

Back to Question 5

Answer 4

A method call evaluates to the value returned by the method.

Back to Question 4

Answer 3

An expression is a specific combination of operators and operands that evaluates to a particular value. The operands can be variables, constants, or method calls.

Back to Question 3

Answer 2

Statements in Java re constructed from expressions.

Back to Question 2

Answer 1

A Java program is composed of a series of statements.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0220r Review: Statements and Expressions
- File: Jb0220r.htm
- Originally published: 1997
- Published at cnx.org: 11/24/12
- Revised: 12/04/14

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0230: Java OOP: Flow of Control
Java supports several different statements designed to alter or control the logical flow of the program. This module explores those statements.

Revised: Mon Mar 28 15:04:52 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

Java supports several different statements designed to alter or control the logical flow of the program. This module explores those statements.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

Listings

Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, etc. This module concentrates on *flow-of-control* .

Flow of control

What is flow of control?

Java supports several different kinds of statements designed to alter or control the logical flow of the program.

The ability to alter the logical flow of the program is often referred to as *Flow of Control* .

Statements that support flow of control

Figure 1 lists the statements supported by Java for controlling the logical flow of the program.

Figure 1 . Statements that support flow of control.

```
Statement            Type

if-else              selection
switch-case          selection
for                  loop
for-each             loop
while                loop
do-while             loop
try-catch-finally    exception handling
throw                exception handling
break                miscellaneous
continue             miscellaneous
label:               miscellaneous
return               miscellaneous
goto                 reserved by Java but not supported
```

The while statement

We've seen the **while** statement in earlier modules. Several of the programs in earlier modules contained a **while** statement designed to control the logical flow of the program.

Syntax of a while statement

The general syntax of a **while** statement is shown in Figure 2 .

Figure 2 . Syntax of a while statement.

```
while (conditional expression)
  statement or compound statement;
```

Behavior of a while statement

The **three pillars** of procedural programming are

- sequence
- selection
- loop

The **while** statement is commonly used to create a loop structure, often referred to as a *while loop* .

Once the **while** statement is encountered in the sequence of code, the program will continue to execute the statement or compound statement shown in Figure 2 for as long as the conditional expression evaluates to true. *(Note that a compound statement is created by enclosing two or more statements inside a pair of matching curly brackets, thus creating a block of code as the body of the **while** statement or loop.)*

Sample Java *while* statement

The **while** statement shown in Listing 1 was extracted from a Java program in an earlier module.

Listing 1 . Sample Java while statement.

```
while( (ch1 = System.in.read() ) != '#')
  ch2 = ch1;
```

The *in* variable of the *System* class

The **System** class defines a *class* variable named **in** . Because it is a *class* variable, it can be accessed using the name of the **System** class without the requirement to instantiate an object of the **System** class.

What the *in* variable contains

The **in** variable refers to an instance of a class that provides a **read** method that returns a character from the standard input device *(typically the keyboard)* .

Therefore, the expression **System.in.read()** in Listing 1 constitutes a call to the **read** method of the object referred to by the **in** variable of the **System** class.

A *while* loop is an entry condition loop

The **while** statement is used to form an *entry condition* loop. The significance of an entry condition loop is that the conditional expression is tested before the statements in the loop are executed. If it tests false initially, the statements in the loop are never executed.

The **while** loop shown in Listing 1 will continue reading characters from the keyboard for as long as the character entered is not the # character. *(Recall the not equal (!=) operator from an earlier module.)*

The if-else statement

The general syntax of an **if-else** statement is shown in Figure 3 .

Figure 3 . Syntax of an if-else statement.

```
if(conditional expression)
  statement or compound statement;
else //optional
  statement or compound statement; //optional
```

The **if-else** statement is the most basic of the statements used to control the logical flow of a Java program. It is used to satisfy the *selection* pillar mentioned earlier .

This statement will execute a specified block of code if some particular condition is true, and optionally, will execute a different block of code if the condition is not true.

The **else** clause shown in Figure 3 is optional. If it is not provided and the condition is not true, control simply passes to the next statement following the **If** statement with none of the code in the body of the **if** statement being executed. If the condition is true, the code in the body of the **if** statement is executed.

If the **else** clause is provided and the condition is true, the code in the body of the **if** clause is executed and the code in the body of the **else** clause is ignored.

If the **else** clause is provided and the condition is false, the code in the body of the **if** clause is ignored and the code in the body of the **else** clause is executed.

In all cases, control passes to the next statement following the **if-else** statement when the code in the **if-else** statement has finished executing. In other words, this is not a loop structure.

The switch-case statement

The **switch-case** statement is another implementation of the *selection* pillar mentioned earlier . The general syntax of a **switch-case** statement is shown in Figure 4 .

Figure 4 . Syntax of a switch-case statement.

Figure 4 . Syntax of a switch-case statement.

```
switch(expression){
  case constant:
    //sequence of optional statements
    break; //optional
  case constant:
    //sequence of optional statements
    break; //optional
.
.
.
  default //optional
    //sequence of optional statements
}
```

The type of the *expression*

According to the book, *Java Language Reference* , by Mark Grand, the expression shown in the first line in Figure 4 must be of type **byte** , **char** , **short** , or **int** .

The behavior of the switch-case statement

The expression is tested against a series of *case* constants of the same type as the expression. If a match is found, the sequence of optional statements associated with that *case* is executed.

Execution of statements continues until the optional **break** is encountered. When **break** is encountered, execution of the switch statement is terminated and control passes to the next statement following the switch statement.

If there is no **break** statement, all of the statements following the matching case will be executed including those in cases further down the page.

The optional default keyword

If no match is found and the optional default keyword along with a sequence of optional statements has been provided, those statements will be executed.

Labeled break

Java also supports labeled break statements. This capability can be used to cause Java to exhibit different behavior when switch statements are nested. This will be explained more fully in a later section on labeled break statements.

The for loop

The **for** statement is another implementation of the *loop* pillar mentioned earlier .

Actions of a *for loop*

The operation of a loop normally involves three actions in addition to executing the code in the body of the loop:

- Initialize a control variable.
- Test the control variable in a conditional expression.
- Update the control variable.

Grouping the actions

Java provides the **for** loop construct that groups these three actions in one place.

The syntax of a for loop

A **for** loop consists of three clauses separated by semicolons as shown in Figure 5 .

Figure 5 . Syntax of a for loop.

```
for (first clause; second clause; third clause)
  single or compound statement
```

Contents of the clauses

The first and third clauses can contain one or more expressions, separated by the *comma operator* .

The *comma operator*

The comma operator guarantees that its left operand will be executed before its right operand.

(While the comma operator has other uses in C++, this is the only use of the comma operator in Java.)

Behavior and purpose of the first clause

The expressions in the first clause are executed only once, at the beginning of the loop. Any legal expression(s) may be contained in the first clause, but typically the first clause is used for initialization.

Declaring and initializing variables in the first clause

Variables can be declared and initialized in the first clause, and this has an interesting ramification regarding scope that will be discussed later.

Behavior of the second clause

The second clause consists of a single expression that must evaluate to a **boolean** type with a value of true or false. The expression in the second clause must eventually evaluate to false to cause the loop to terminate.

Typically relational expressions or relational and conditional expressions are used in the second clause.

When the test is performed

The value of the second clause is tested when the statement first begins execution, and at the beginning of each iteration thereafter. Therefore, just like the **while** loop, the **for** loop is an *entry condition loop* .

When the third clause is executed

Although the third clause appears physically at the top of the loop, it isn't executed until the statements in the body of the loop have completed execution.

This is an important point since this clause is typically used to update the control variable, and perhaps other variables as well.

What the third clause can contain

Multiple expressions can appear in the third clause, separated by the comma operator. Again, those expressions will be executed from left to right. If variables are updated in the third clause and used in the body of the loop, it is important to understand that they do not get updated until the execution of the body is completed.

Declaring a variable in a *for* loop

As mentioned earlier, it is allowable to declare variables in the first clause of a **for** loop.

You can declare a variable with a given name outside *(prior to)* the **for** loop, or you can declare it inside the **for** loop, but not both.

If you declare it outside the **for** loop, you can access it either outside or inside the loop.

If you declare it inside the loop, you can access it only inside the loop. In other words, the scope of variables declared inside a **for** loop is limited to the loop.

This is illustrated in following sequence of four simple programs.

This program won't compile

The Java program shown in Listing 2 refuses to compile with a complaint that a variable named **cnt** has already been declared in the method when the attempt is made to declare it in the **for** loop.

Listing 2 . A program that won't compile.

```
/*File for1.java Copyright 1997, R.G.Baldwin
This program will not compile because the variable
named cnt is declared twice.
**************************************************/
class for1 { //define the controlling class
  public static void main(String[] args){ //main method
    int cnt = 5; //declare local method variable
    System.out.println(
          "Value of method var named cnt is " + cnt);

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
              "Value of loop var named cnt is " + cnt);

    System.out.println(
            "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

The program shown in Listing 3 also won't compile, but for a different reason.

Listing 3 . Another program that won't compile.

```
/*File for2.java Copyright 1997, R.G.Baldwin
This program will not compile because the variable
declared inside the for loop is not accessible
outside the loop.
**************************************************/
class for2 { //define the controlling class
  public static void main(String[] args){ //main method

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " +
cnt);

    System.out.println(
                "Value of method var named cnt is " +
cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

The declaration of the variable named **cnt** , outside the **for** loop, was removed from Listing 3 and the declaration inside the loop was allowed to remain. This eliminated the problem of attempting to declare the variable twice.

However, this program refused to compile because an attempt was made to access the variable named **cnt** outside the **for** loop. This was not allowed because the variable was declared inside the **for** loop and the scope of the variable was limited to the loop.

This program will compile

The Java program shown in Listing 4 will compile and run because the variable named **cnt** that is declared inside the **for** loop is accessed only inside the **for** loop. No reference to a variable with the same name appears outside the loop.

Listing 4 . A program that will compile.

Listing 4 . A program that will compile.

```
/*File for3.java Copyright 1997, R.G.Baldwin
This program will compile because the variable declared
inside the for loop is accessed only inside the loop.
**************************************************************/
class for3 { //define the controlling class
  public static void main(String[] args){ //main method

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " + cnt);
  }//end main
}//End controlling class.
```

This program will also compile

Similarly, the program shown in Listing 5 will compile and run because the variable named **cnt** was declared outside the **for** loop and was not declared inside the **for** loop. This made it possible to access that variable both inside and outside the loop.

Listing 5 . Another program that will compile.

Listing 5 . Another program that will compile.

```
/*File for4.java Copyright 1997, R.G.Baldwin
This program will compile and run because the variable
named cnt is declared outside the for loop and is not
declared inside the for loop.
**************************************************/
class for4 { //define the controlling class
  public static void main(String[] args){ //main method
    int cnt = 5; //declare local method variable
    System.out.println(
              "Value of method var named cnt is " + cnt);

    for(cnt = 0; cnt < 2; cnt++)
      System.out.println(
                 "Value of loop var named cnt is " + cnt);

    System.out.println(
                "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

Empty clauses in a *for* loop

The first and third clauses in a **for** loop can be left empty but the semicolons must be there as placeholders.

One author suggests that even the middle clause can be empty, but it isn't obvious to this author how the loop would ever terminate if there is no conditional expression to be evaluated. Perhaps the loop could be terminated by using a break inside the loop, but in that case, you might just as well use a **while** loop.

The for-each loop

There is another form of loop structure that is often referred to as a **for-each** loop. In order to appreciate the benefits of this loop structure, you need to be familiar with Java collections and iterators, both of which are beyond the scope of this module.

As near as I can tell, there is nothing that you can do with the **for-each** loop that you cannot also do with the conventional **for** loop described above. Therefore, I rarely use it. You can find a description of the **for-each** loop on this Oracle website .

I don't plan to discuss it further in this module. However, before you go for a job interview, you should probably do some online research and learn about it because an interviewer could use a question about the **for-each** loop to trip you up in the Q and A portion of the interview.

The do-while loop

The **do-while** loop is another implementation of the *loop* pillar mentioned earlier . However, it differs from the **while** loop and the **for** loop in one important respect; it is an *exit-condition* loop.

An exit-condition loop

Java provides an *exit-condition* loop having the syntax shown in Figure 6 .

Figure 6 . Syntax of a do-while loop.

```
do {
    statements
              } while (conditional expression);
```

Behavior

The statements in the body of the loop continue to be executed for as long as the conditional expression evaluates to true. An exit-condition loop guarantees that the body of the loop will be executed at least one time, even if the conditional expression evaluates to false the first time it is tested.

The break and continue statements

General behavior

Although some authors suggest that the **break** and **continue** statements provide an alternative to the infamous **goto** statement of earlier programming languages, it appears that the behaviors of the **labeled break** and **labeled continue** statements are much more restrictive than a general **goto** .

Unlabeled break and continue

The **break** and **continue** statements are supported in both labeled and unlabeled form.

First consider the behavior of break and continue in their unlabeled configuration.

Use of a *break* statement

The **break** statement can be used in a switch statement or in a loop. When encountered in a switch statement, break causes control to be passed to the next statement outside the innermost enclosing switch statement.

When break is encountered in a loop, it causes control to be passed to the next statement outside the innermost enclosing loop.

As you will see later, labeled break statements can be used to pass control to the next statement following switch or loop statements beyond the innermost switch or loop statement when those statements are nested.

Use of a continue statement

The continue statement cannot be used in a switch statement, but can be used inside a loop.

When an unlabeled continue statement is encountered, it causes the current iteration of the current loop to be terminated and the next iteration to begin.

A labeled continue statement can cause control to be passed to the next iteration of an outer enclosing loop in a nested loop situation.

An example of the use of an unlabeled switch statement is given in the next section.

Labeled break and continue statements

This section discusses the use of labeled break and continue statements.

Labeled break Statements

One way to describe the behavior of a labeled break in Java is to say: "Break all the way out of the labeled statement."

Syntax of a labeled statement

To begin with, the syntax of a labeled statement is a label followed by a colon ahead of the statement as shown in Figure 7 .

Figure 7 . Syntax of a labeled statement.

```
myLabel:  myStatement;
```

The label can be any legal Java identifier.

Behavior of labeled break

The behavior of a labeled break can best be illustrated using nested switch statements. For a comparison of labeled and unlabeled switch statements, consider the program shown in Listing 6 named **switch1** , which does not use a labeled break. Even though this program has a labeled statement, that statement is not referenced by a **break** . Therefore, the label is of no consequence.

Listing 6 . The program named switch1.java.

```
/*File switch1.java
This is a Java application which serves as a baseline
comparison for switch2.java which uses a labeled break.
Note that the program uses nested switch statements.

The program displays the following output:

Match and break from here
Case 6 in outer switch
Default in outer switch
Beyond switch statements

**********************************************************/
class switch1 { //define the controlling class
  public static void main(String[] args){ //main method

  //Note that the following labeled switch statement is
  // not referenced by a labeled break in this program.
  // It will be referenced in the next program.
```

Listing 6 . The program named switch1.java.

```
  outerSwitch: switch(5){//labeled outer switch statement
    case 5: //execute the following switch statement
    //Note that the code for this case is not followed
    // by break. Therefore, execution will fall through
    // the case 6 and the default.
      switch(1){ //inner switch statement
        case 1: System.out.println(
                              "Match and break from here");
                break; //break with no label
        case 2: System.out.println(
                             "No match for this constant");
                break;
      }//end inner switch statement

    case 6: System.out.println("Case 6 in outer switch");
    default: System.out.println(
                               "Default in outer switch");
  }//end outer switch statement

  System.out.println("Beyond switch statements");
  }//end main
}//End switch1 class.
```

After reviewing **switch1.java** , consider the same program named **switch2.java** shown in Listing 7 , which was modified to use a labeled break.

The outputs from both programs are shown in the comments at the beginning of the program. By examining the second program, and comparing the output from the second program with the first program, you should be able to see how the use of the labeled break statement causes control to break all the way out of the labeled switch statement.

Listing 7 . The program named switch2.java.

```
/*File switch2.java
This is a Java application which uses a labeled break.
Note that the program uses nested switch statements.
```

Listing 7 . The program named switch2.java.

```
See switch1.java for a comparison program which does not
use a labeled break.

The program displays the following output:

Match and break from here
Beyond switch statements
**********************************************************/
class switch2 { //define the controlling class
  public static void main(String[] args){ //main method

  outerSwitch: switch(5){//labeled outer switch statement
    case 5: //execute the following switch statement
    //Note that the code for this case is not followed by
    // break. Therefore, except for the labeled break at
    // case 1, execution would fall through the case 6 and
    // the default as demonstrated in the program named
    // switch1. However, the use of the labeled break
    // causes control to break all the way out of the
    // labeled switch bypassing case 6 and the default.
      switch(1){ //inner switch statement
        case 1: System.out.println(
                              "Match and break from here");
                break outerSwitch; //break with label
        case 2: System.out.println(
                             "No match for this constant");
                break;
      }//end inner switch statement

    case 6: System.out.println(
                                  "Case 6 in outer switch");
    default: System.out.println("Default in outer switch");
  }//end outer switch statement

  System.out.println("Beyond switch statements");
  }//end main
}//End switch1 class.
```

The modified program in Listing 7 uses a labeled break statement in the code group for *case 1* whereas the original program in Listing 6 has an unlabeled break in that position.

By comparing the output from this program with the output from the previous program, you can see that execution of the labeled break statement caused control to break all the way out of the labeled switch statement completely bypassing *case 6* and default.

As you can see from examining the output, the labeled break statement causes the program to break all the way out of the switch statement which bears a matching label.

A similar situation exists when a labeled break is used in nested loops with one of the enclosing outer loops being labeled. Control will break out of the enclosing loop to which the labeled break refers. It will be left as an exercise for the student to demonstrate this behavior to his or her satisfaction.

Labeled continue statements

Now consider use of the labeled continue statement. A **continue** statement can only be used in a loop; it cannot be used in a switch. The behavior of a labeled continue statement can be described as follows: "Terminate the current iteration and continue with the next iteration of the loop to which the label refers."

Again, it will be left as an exercise for the student to demonstrate this behavior to his or her satisfaction.

The return statement

Use of the return statement

Java supports the use of the **return** statement to terminate a method and *(optionally)* return a value to the calling method.

The return type

The type of value returned must match the type of the declared return value for the method.

The void return type

If the return value is declared as **void** , you can use the syntax shown in Figure 8 to terminate the method. *(You can also simply allow the method to run out of statements to execute.)*

Figure 8 . An empty return statement.

```
return;
```

Returning a value

If the method returns a value, follow the word return with an expression *(or constant)* that evaluates to the value being returned as shown in Figure 9 .

Figure 9 . Returning a value from a method.

```
return x+y;
```

Return by value only

You are allowed to return only by *value* . In the case of primitive types, this returns a copy of the returned item. In the case of objects, returning by value returns a copy of the object's reference.

What you can do with a copy the object's reference

Having a copy of the reference is just as good as having the original reference. A copy of the reference gives you access to the object.

When Java objects are destroyed

All objects in Java are stored in dynamic memory and that memory is not overwritten until all references to that memory cease to exist.

Java uses a garbage collector running on a background thread to reclaim memory from objects that have become *eligible for garbage collection* .

An object becomes eligible for garbage collection when there are no longer any variables, array elements, or similar storage locations containing a reference to the object. In other

words, it becomes eligible when there is no way for the program code to find a reference to the object.

Exception handling

Exception handling is a process that modifies the flow of control of a program, so it merits being mentioned in this module. However, it is a fairly complex topic, which will be discussed in detail in future modules.

Suffice it at this point to say that whenever an exception is detected, control is transferred to exception handler code if such code has been provided. Otherwise, the program will terminate. Thus, the exception handling system merits being mentioned in discussions regarding flow of control.

Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library
- Java OOP: Objects and Encapsulation

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0230: Java OOP: Flow of Control
- File: Jb0230.htm
- Originally published: 1997
- Published at cnx.org: 11/24/12

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0230r Review
This module contains review questions and answers keyed to the module titled Jb0230: Java OOP: Flow of Control.

Revised: Mon Mar 28 15:41:09 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0230: Java OOP: Flow of Control .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

List and describe eight of the statements used in Java programs to alter or control the logical flow of the program.

Answer 1

Question 2

Provide pseudo-code that illustrates the general syntax of a **while** statement.

Answer 2

Question 3

True or false? During the execution of a **while** statement, the program will continue to execute the statement or compound statement for as long as the conditional expression evaluates to true, or until a **break** , **continue** ,or **return** statement is encountered. If false, explain why.

Answer 3

Question 4

True or false? A **while** loop is an *entry condition* loop. If false, explain why.

Answer 4

Question 5

What is the significance of an *entry condition* loop?

Answer 5

Question 6

Provide pseudo-code illustrating the general syntax of the **if-else** statement.

Answer 6

Question 7

Provide pseudo-code illustrating the general syntax of the **switch-case** statement.

Answer 7

Question 8

Describe the behavior of a **switch-case** statement. Provide a pseudo-code fragment that illustrates your description of the behavior. Do not include a description of labeled break statements.

Answer 8

Question 9

What are the three actions normally involved in the operation of a loop *(in addition to executing the code in the body of the loop)* ?

Answer 9

Question 10

True or false? A **for** loop header consists of three clauses separated by colons. If false, explain why.

Answer 10

Question 11

Provide pseudo-code illustrating the general syntax of a **for** loop

Answer 11

Question 12

True or false? In a **for** loop, the first and third clauses within the parentheses can contain one or more expressions, separated by the comma operator. If False, explain why.

Answer 12

Question 13

What is the guarantee made by the *comma operator* ?

Answer 13

Question 14

True or false? The expressions within the first clause in the parentheses in a **for** loop are executed only once during each iteration of the loop. If false, explain why.

Answer 14

Question 15

While any legal expression(s) may be contained in the first clause within the parentheses of a **for** loop, the first clause has a specific purpose. What is

that purpose?

Answer 15

Question 16

True or false? Variables can be declared and initialized within the first clause in the parentheses of a for loop. If false, explain why.

Answer 16

Question 17

True or false? The second clause in the parentheses of a **for** loop consists of a single expression which must eventually evaluate to true to cause the loop to terminate. If false, explain why.

Answer 17

Question 18

True or false? A **for** loop is an *exit condition* loop. If false, explain why.

Answer 18

Question 19

True or false? Because a **for** loop is an *entry condition* loop, the third clause inside the parentheses is executed at the beginning of each iteration. If false, explain why.

Answer 19

Question 20

True or false? A return statement is used to terminate a method and *(optionally)* return a value to the calling method. If False, explain why.

Answer 20

Question 21

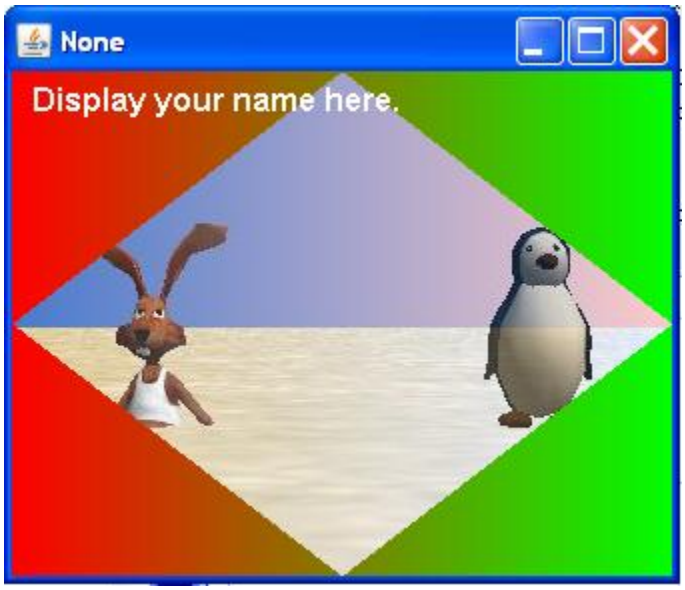
True or false? Exception handling modifies the flow of control of a Java program. If false, explain why.

Answer 21

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 21

True.

Back to Question 21

Answer 20

True.

Back to Question 20

Answer 19

False. Although the third clause appears physically at the top of the loop, it isn't executed until the statements in the body of the loop have completed execution. This is an important point since this clause is typically used to update the control variable, and perhaps other variables as well. If variables are updated in the third clause and used in the body of the loop, it is important to understand that they do not get updated until the execution of the body is completed.

Back to Question 19

Answer 18

False. The value of the second clause is tested when the statement first begins execution, and at the beginning of each iteration thereafter. Therefore, the **for** loop is an *entry condition* loop.

Back to Question 18

Answer 17

False. The second clause consists of a single expression which must eventually evaluate to false *(not true)* to cause the loop to terminate.

Back to Question 17

Answer 16

True.

Back to Question 16

Answer 15

Typically the first clause is used for initialization. The intended purpose of the first clause is initialization.

Back to Question 15

Answer 14

False. The expressions in the first clause are executed only once, at the beginning of the loop, regardless of the number of iterations.

Back to Question 14

Answer 13

The *comma operator* guarantees that its left operand will be executed before its right operand.

Back to Question 13

Answer 12

True.

Back to Question 12

Answer 11

The general syntax of a **for** loop follows:

Note: Syntax of a for loop

```
for (first clause; second clause; third clause)
  single or compound statement
```

Back to Question 11

Answer 10

False: A **for** loop header consists of three clauses separated by semicolons, not colons.

Back to Question 10

Answer 9

The operation of a loop normally involves the following three actions in addition to executing the code in the body of the loop:

- Initialize a control variable.
- Test the control variable in a conditional expression.
- Update the control variable.

Back to Question 9

Answer 8

The pseudo-code fragment follows:

Note: Syntax of a switch-case statement

```
switch(expression){
  case constant:
    sequence of optional statements
    break; //optional
  case constant:
    sequence of optional statements
    break; //optional
.
.
.
  default //optional
    sequence of optional statements
}
```

An expression is tested against a series of unique integer constants. If a match is found, the sequence of optional statements associated with the matching constant is executed. Execution of statements continues until an optional **break** is encountered. When **break** is encountered, execution of the **switch** statement is terminated and control is passed to the next statement following the **switch** statement.

If no match is found and the optional **default** keyword along with a sequence of optional statements has been provided, those statements will be executed.

Back to Question 8

Answer 7

The general syntax of the **switch-case** statement follows:

Note: Syntax of a switch-case statement

```
switch(expression){
  case constant:
    sequence of optional statements
    break; //optional
  case constant:
    sequence of optional statements
    break; //optional
.
.
.
  default //optional
    sequence of optional statements
}
```

Back to Question 7

Answer 6

The general syntax of the if-else statement is:

Note: Syntax of an if-else statement

```
if(conditional expression)
  statement or compound statement;
else //optional
  statement or compound statement; //optional
```

Back to Question 6

Answer 5

The significance of an *entry condition* loop is that the conditional expression is tested before the statements in the loop are executed. If it tests false initially, the statements in the loop will not be executed.

Back to Question 5

Answer 4

True.

Back to Question 4

Answer 3

True. Note however that including a **return** statement inside a **while** statement is probably considered poor programming practice.

Back to Question 3

Answer 2

The general syntax of a **while** statement follows :

Note: Syntax of a while statement

```
while (conditional expression)
   statement or compound statement;
```

Back to Question 2

Answer 1

The following table lists the statements supported by Java for controlling the logical flow of the program.

Note: Flow of control statements

```
Statement             Typeif-else
selection
switch-case           selection
for                   loop
for-each              loop
while                 loop
do-while              loop
try-catch-finally     exception handling
throw                 exception handling
break                 miscellaneous
continue              miscellaneous
label:                miscellaneous
return                miscellaneous
goto                  reserved by Java but not
supported
```

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0230r Review: Flow of Control
- File: Jb0230r.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12

-end-

Jb0240: Java OOP: Arrays and Strings
This module takes a preliminary look at arrays and strings. More in-depth discussions will be provided in future modules.

Revised: Mon Mar 28 16:20:59 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module takes a preliminary look at arrays and strings. More in-depth discussions will be provided in future modules. For example, you will find a more in-depth discussions of array objects in the following modules:

- Java OOP: Array Objects, Part 1
- Java OOP: Array Objects, Part 2
- Java OOP: Array Objects, Part 3

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, arrays, strings, etc. This module concentrates on arrays and strings.

Array and String types

Java provides a type for both arrays and strings from which objects of the specific type can be instantiated. Once instantiated, the methods belonging to those types can be called by way of the object.

Arrays

Arrays and Strings

Java has a true array type and a true **String** type with protective features to prevent your program from writing outside the memory bounds of the array object or the **String** object.

Arrays and strings are true objects.

Declaring an array

You must declare an array before you can use it. *(More properly, you must declare a reference variable to hold a reference to the array object.)* In declaring the array, you must provide two important pieces of information:

- the name of a variable to hold a reference to the array object
- the type of data to be stored in the elements of the array object

Different declaration formats

A reference variable capable of holding a reference to an array object can be declared using either format shown in Figure 1 . *(I personally prefer the first option because I believe it is more indicative of the purpose of the declaration. However, both options produce the same result -- a reference variable capable of storing a reference to an array object.)*

Figure 1 . Formats for declaring a reference variable for an array object.

```
int[] myArray;
int myArray[];
```

Declaration does not allocate memory

As with other objects, the declaration of the reference variable does not allocate memory to contain the array data. Rather it simply allocates memory to contain a reference to the array.

Allocating memory for the array object

Memory to contain the array object must be allocated from dynamic memory using statements such as those shown in Figure 2 .

Figure 2 . Allocating memory for the array object.

```
int[] myArrayX = new int[15];
int myArrayY[] = new int[25];

int[] myArrayZ = {3,4,5};
```

The statements in Figure 2 simultaneously declare the reference variable and cause memory to be allocated to contain the array.

Also note that the last statement in Figure 2 is different from the first two statements. This syntax not only sets aside the memory for the array object, the elements in the array are initialized by evaluating the expressions shown in the coma-separated list inside the curly brackets.

On the other hand, the array elements in the first two statements in Figure 2 are automatically initialized with the default value for the type.

Declaration and allocation can be separated

It is not necessary to combine these two processes. You can execute one statement to declare the reference variable and another statement to cause the array object to be instantiated some time later in the program as shown in Figure 3 .

Figure 3 . Declaration and instantiation can be separated.

```
int[] myArray;
. . .
myArray = new int[25];
```

Causing memory to be set aside to contain the array object is commonly referred to as instantiating the array object *(creating an instance of the array object)* .

If you prefer to declare the reference variable and instantiate the array object at different points in your program, you can use the syntax shown in Figure 3 . This pattern is very similar to the declaration and instantiation of all objects.

General syntax for combining declaration and instantiation

The general syntax for declaring and instantiating an array object is shown in Figure 4 .

Figure 4 . General syntax for combining declaration and instantiation.

```
typeOfElements[] nameOfRefVariable =
           new typeOfElements[sizeOfArray]
```

Accessing array elements

Having instantiated an array object, you can access the elements of the array using indexing syntax that is similar to many other programming languages. An example is shown in Figure 5 .

Figure 5 . An example of array indexing syntax.

```
myArray[5] = 6;

myVar = myArray[5];
```

The value of the first index

Array indices always begin with 0.

The length property of an array

The code fragment in Figure 6 illustrates another interesting aspect of arrays. *(Note the use of **length** in the conditional clause of the **for** loop.)*

Figure 6 . The use of the length property in the conditional clause of a for loop.

```
for(int cnt = 0; cnt < myArray.length; cnt++)
  myArray[cnt] = cnt;
```

All array objects have a **length** property that can be accessed to determine the number of elements in the array. *(The number of elements cannot change once the array object is instantiated.)*

Types of data that you can store in an array object

Array elements can contain any Java data type including primitive values and references to ordinary objects or references to other array objects.

Constructing multi-dimensional arrays

All array objects contains a one-dimensional array structure. You can create multi-dimensional arrays by causing the elements in one array object to contain references to other array objects. In effect, you can create a tree structure of array objects that behaves like a multi-dimensional array.

Odd-shaped multi-dimensional arrays

The program **array01** shown in Listing 1 illustrates an interesting aspect of Java arrays. Java can produce multi-dimensional arrays that can be thought of as an array of arrays. However, the secondary arrays need not all be of the same size.

In the program shown in Listing 1 , a two-dimensional array of integers is declared and instantiated with the primary size *(size of the first dimension)* being three. The sizes of the secondary dimensions *(sizes of each of the sub-arrays)* is 2, 3, and 4 respectively.

Can declare the size of secondary dimension later

When declaring a *"two-dimensional"* array, it is not necessary to declare the size of the secondary dimension when the primary array is instantiated. Declaration of the size of each sub-array can be deferred until later as illustrated in this program.

Accessing an array out-of-bounds

This program also illustrates the result of attempting to access an element that is out-of-bounds. Java protects you from such programming errors.

ArrayIndexOutOfBoundsException

An exception occurs if you attempt to access out-of-bounds, as shown in the program in in Listing 1 .

In this case, the exception was simply allowed to cause the program to terminate. The exception could have been caught and processed by an exception handler, a concept that will be explored in a future module.

The program named array01

The entire program is shown in Listing 1 . The output from the program is shown in the comments at the top of the listing.

Listing 1 . The program named array01.

```
/*File array01.java Copyright 1997, R.G.Baldwin
Illustrates creation and manipulation of two-dimensional
array with the sub arrays being of different lengths.

Also illustrates detection of exception when an attempt is
made to store a value out of the array bounds.

This program produces the following output:

00
012
0246
Attempt to access array out of bounds
java.lang.ArrayIndexOutOfBoundsException:
     at array01.main(array01.java: 47)

**************************************************************/
class array01 { //define the controlling class
  public static void main(String[] args){ //main method
```

Listing 1 . The program named array01.

```
    //Declare a two-dimensional array with a size of 3 on
    // the primary dimension but with different sizes on
    // the secondary dimension.
    //Secondary size not specified initially
    int[][] myArray = new int[3][];
    myArray[0] = new int[2];//secondary size is 2
    myArray[1] = new int[3];//secondary size is 3
    myArray[2] = new int[4];//secondary size is 4

    //Fill the array with data
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        myArray[i][j] = i * j;
      }//end inner loop
    }//end outer loop

    //Display data in the array
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        System.out.print(myArray[i][j]);
      }//end inner loop
      System.out.println();
    }//end outer loop

    //Attempt to access an out-of-bounds array element
    System.out.println(
                   "Attempt to access array out of bounds");
    myArray[4][0] = 7;
    //The above statement produces an ArrayIndexOutOfBounds
    // exception.

  }//end main
}//End array01 class.
```

Assigning one array to another array -- be careful

Java allows you to assign one array to another. You must be aware, however, that when you do this, you are simply making another copy of the reference to the same data in memory.

Then you simply have two references to the same data in memory, which is often not a good idea. This is illustrated in the program named **array02** shown in Listing 2 .

Listing 2 . The program named array02 .

```
/*File array02.java Copyright 1997, R.G.Baldwin
Illustrates that when you assign one array to another
array, you end up with two references to the same array.

The output from running this program is:

firstArray contents
0 1 2
secondArray contents
0 1 2
Change a value in firstArray and display both again
firstArray contents
0 10 2
secondArray contents
0 10 2
**********************************************************/
class array02 { //define the controlling class
  int[] firstArray;
  int[] secondArray;

  array02() {//constructor
    firstArray = new int[3];
    for(int cnt = 0; cnt < 3; cnt++) firstArray[cnt] = cnt;

    secondArray = new int[3];
    secondArray = firstArray;
  }//end constructor

  public static void main(String[] args){//main method
    array02 obj = new array02();
    System.out.println( "firstArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.firstArray[cnt] + " " );
    System.out.println();

    System.out.println( "secondArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.secondArray[cnt] + " " );

    System.out.println();
    System.out.println(
```

Listing 2 . The program named array02 .

```
      "Change value in firstArray and display both again");
    obj.firstArray[1] = 10;

    System.out.println( "firstArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.firstArray[cnt] + " " );
    System.out.println();

    System.out.println( "secondArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.secondArray[cnt] + " " );

    System.out.println();
  }//end main
}//End array02 class.
```

Arrays of Objects

An array of objects really isn't an array of objects

There is another subtle issue that you need to come to grips with before we leave our discussion of arrays. In particular, when you create an array of objects, it really isn't an array of objects.

Rather, it is an array of object references *(or null)* . When you assign primitive values to the elements in an array object, the actual primitive values are stored in the elements of the array.

However, when you assign objects to the elements in an array , the actual objects aren't actually stored in the array elements. Rather, the objects are stored somewhere else in memory. The elements in the array contain references to those objects.

All the elements in an array of objects need not be of the same actual type

The fact that the array is simply an array of reference variables has some interesting ramifications. For example, it isn't necessary that all the elements in the array be of the same type, provided the reference variables are of a type that will allow them to refer to all the different types of objects.

For example, if you declare the array to contain references of type **Object** , those references can refer to any type of object *(including array objects)* because a reference of type **Object** can be used to refer to any object.

You can do similar things using *interface* types. I will discuss interface types in a future module.

Often need to downcast to use an Object reference

If you store all of your references as type **Object** , you will often need to downcast the references to the true type before you can use them to access the instance variables and instance methods of the objects.

Doing the downcast no great challenge as long as you can decide what type to downcast them to.

The Vector class

There is a class named **Vector** that takes advantage of this capability. An object of type **Vector** is a self-expanding array of reference variables of type **Object** . You can use an object of type **Vector** to manage a group of objects of any type, either all of the same type, or mixed.

(Note that you cannot store primitive values in elements of a non-primitive or reference type. If you need to do that, you will need to wrap your primitive values in an object of a wrapper class as discussed in an earlier module.)

A sample program using the Date class

The sample program, named **array03** and shown in Listing 3 isn't quite that complicated. This program behaves as follows:

- Declare a reference variable to an array of type **Date** . *(The actual type of the variable is Date[].)*
- Instantiate a three-element array of reference variables of type **Date** .
- Display the contents of the array elements and confirm that they are all null as they should be. *(When created using this syntax, new array elements contain the default value, which is null for reference types.)*
- Instantiate three objects of type **Date** and store the references to those objects in the three elements of the array.
- Access the references from the array and use them to display the contents of the individual **Date** objects.

As you might expect from the name of the class, each object contains information about the date.

Listing 3 . The program named Array03 .

```
/*File array03.java Copyright 1997, R.G.Baldwin

Illustrates use of arrays with objects.

Illustrates that "an array of objects" is not really an
array of objects, but rather is an array of references
to objects.  The objects are not stored in the array,
but rather are stored somewhere else in memory and the
references in the array elements refer to them.

The output from running this program is:

myArrayOfRefs contains
null
null
null

myArrayOfRefs contains
Sat Dec 20 16:56:34 CST 1997
Sat Dec 20 16:56:34 CST 1997
Sat Dec 20 16:56:34 CST 1997
**********************************************************/
import java.util.*;

class array03 { //define the controlling class
  Date[] myArrayOfRefs; //Declare reference to the array

  array03() {//constructor
    //Instantiate the array of three reference variables
    // of type Date.  They will be initialized to null.
    myArrayOfRefs = new Date[3];

    //Display the contents of the array.
    System.out.println( "myArrayOfRefs contains" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.println(this.myArrayOfRefs[cnt]);
    System.out.println();

    //Instantiate three objects and assign references to
    // those three objects to the three reference
    // variables in the array.
```

Listing 3 . The program named Array03 .

```
    for(int cnt = 0; cnt < 3; cnt++)
      myArrayOfRefs[cnt] = new Date();

  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args){//main method
    array03 obj = new array03();
    System.out.println( "myArrayOfRefs contains" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.println(obj.myArrayOfRefs[cnt]);
    System.out.println();
  }//end main
}//End array03 class.
```

Strings

What is a string?

A string is commonly considered to be a sequence of characters stored in memory and accessible as a unit.

Java implements strings using the **String** class and the **StringBuffer** class.

What is a string literal?

Java considers a series of characters surrounded by quotation marks as shown in Figure 7 to be a string literal.

Figure 7 . A string literal.

```
"This is a string literal in Java."
```

This is just an introduction to strings

A major section of a future module will be devoted to the topic of strings, so this discussion will be brief.

String objects cannot be modified

String objects cannot be changed once they have been created. *(They are said to be immutable.)* If you have that need, use the **StringBuffer** class instead.

StringBuffer objects can be used to create and manipulate character data as the program executes.

String Concatenation

Java supports string concatenation using the overloaded + operator as shown in Figure 8 .

Figure 8 . String concatenation.

```
"My variable has a value of " + myVar
   + " at this point in the program."
```

Coercion of an operand to type String

The overloaded + operator is used to concatenate strings. If either operand is type **String** , the other operand is coerced into type **String** and the two strings are concatenated.

Therefore, in addition to concatenating the strings, Java also converts values of other types, such as **myVar** in Figure 8 , to character-string format in the process.

Arrays of String References

Declaring and instantiating a String array

The statement in Figure 9 declares and instantiates an array of references to five **String** objects.

Figure 9 . Declaring and instantiating a String array.

```
String[] myArrayOfStringReferences = new String[5];
```

No string data at this point

Note however, that this array doesn't contain the actual **String** objects. Rather, it simply sets aside memory for storage of five references of type **String** . *(The array elements are automatically initialized to null.)* No memory has been set aside to store the characters that make up the individual **String** objects. You must allocate the memory for the actual **String** objects separately using code similar to the code shown in Figure 10 .

Figure 10 . Allocating memory to contain the String objects.

```
myArrayOfStringReferences[0] = new String(
    "This is the first string.");
myArrayOfStringReferences[1] = new String(
    "This is the second string.");
```

The new operator is not required for String class

Although it was used in Figure 10 , the **new** operator is not required to instantiate an object of type **String** . I will discuss the ability of Java to instantiate objects of type **String** without the requirement to use the **new** operator in a future module.

Run the programs

I encourage you to copy the code from Listing 1 , Listing 2 , and Listing 3 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library
- Java OOP: Objects and Encapsulation

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0240: Java OOP: Arrays and Strings
- File: Jb0240.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0240r Review
This module contains review questions and answers keyed to the module titled Jb0240: Java OOP: Arrays and Strings.

Revised: Mon Mar 28 16:31:25 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0240: Java OOP: Arrays and Strings .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or false? Arrays and Strings are true objects. If false, explain why.

Answer 1

Question 2

True or false? It is easy to write outsides the bounds of a **String** or an array. If false, explain why.

Answer 2

Question 3

You must declare a variable capable of holding a reference to an array object before you can use it. In declaring the variable, you must provide two important pieces of information. What are they?

Answer 3

Question 4

Provide code fragments that illustrate the two different syntaxes that can be used to declare a variable capable of holding a reference to an array object that will store data of type int.

Answer 4

Question 5

True or false? When you declare a variable capable of holding a reference to an array object, the memory required to contain the array object is automatically allocated. If false, explain why and show how memory can be allocated.

Answer 5

Question 6

True or false? It is required that you simultaneously declare the name of the variable and cause memory to be allocated to contain the array object in a single statement. If false, explain why and show code fragments to illustrate your answer.

Answer 6

Question 7

True or false? Array indices always begin with 1. If false, explain why.

Answer 7

Question 8

What is the name of the property of arrays that can be accessed to determine the number of elements in the array? Provide a sample code fragment that illustrates the use of this property.

Answer 8

Question 9

What types of data can be stored in array objects?

Answer 9

Question 10

True or false? Just as in other languages, when you create a multi-dimensional array, the secondary arrays must all be of the same size. If false, explain your answer. Then provide a code fragment that illustrates your answer or refer to a sample program in Jb0240: Java OOP: Arrays and Strings that illustrates your answer.

Answer 10

Question 11

True or false? Just as in other languages, when declaring a two-dimensional array, it is necessary to declare the size of the secondary dimension when the array is declared. If false, explain your answer. Then provide a code fragment that illustrates your answer or refer to a sample program in Jb0240: Java OOP: Arrays and Strings that illustrates your answer.

Answer 11

Question 12

True or false? Java allows you to assign one array to another. Explain what happens when you do this. Then provide a code fragment that illustrates your answer or refer to a sample program in Jb0240: Java OOP: Arrays and Strings that illustrates your answer.

Answer 12

Question 13

Give a brief description of the concept of a string and list the names of two classes used to implement strings?

Answer 13

Question 14

What is the syntax that is used to create a literal string? Provide a code fragment to illustrate your answer.

Answer 14

Question 15

Explain the difference between objects of types **String** and **StringBuffer** .

Answer 15

Question 16

Provide a code fragment that illustrates how to concatenate strings.

Answer 16

Question 17

Provide a code fragment that declares and instantiates an array object capable of storing references to two **String** objects. Explain what happens when this code fragment is executed. Then show a code fragment that will allocate memory for the actual **String** objects.

Answer 17

Question 18

Write a Java application that illustrates the creation and manipulation of a two-dimensional array with the sub arrays being of different lengths. Also

cause your application to illustrate that an attempt to access an array element out of bounds results in an exception being thrown. Catch and process the exception. Display a termination message with your name.

Answer 18

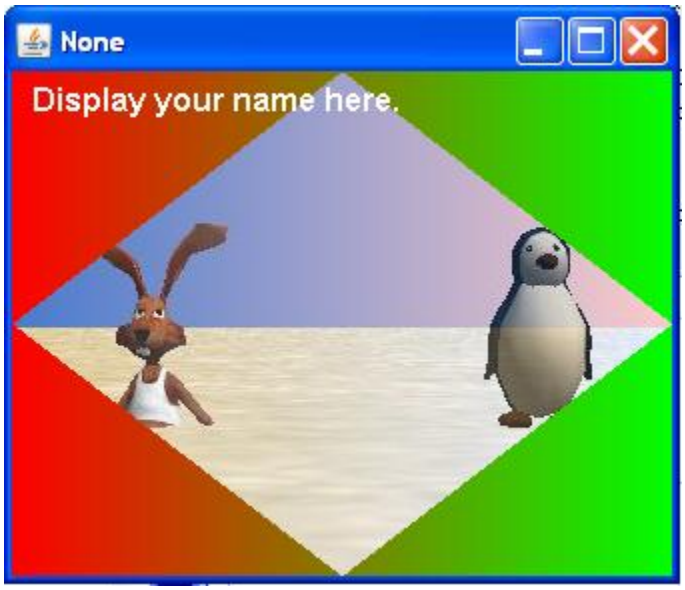
Listings

- Listing 1 . Listing for Answer 18.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 18

Listing 1 . Listing for Answer 18.

```
class SampProg10 { //define the controlling
class
  public static void main(String[] args){
//define main
    //Declare a two-dimensional array with a
size of 3 on
    // the primary dimension but with
different sizes on
    // the secondary dimension.
```

Listing 1 . Listing for Answer 18.

```
    //Secondary size not specified
    int[][] myArray = new int[3][];
    myArray[0] = new int[2];//secondary size
is 2
    myArray[1] = new int[3];//secondary size
is 3
    myArray[2] = new int[4];//secondary size
is 4

    //Fill the array with data
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length;
j++){
        myArray[i][j] = i * j;
      }//end inner loop
    }//end outer loop

    //Display data in the array
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length;
j++){
        System.out.print(myArray[i][j]);
      }//end inner loop
      System.out.println();
    }//end outer loop

    //Attempt to access an out-of-bounds array
element
    try{
      System.out.println(
                  "Attempt to access array out
of bounds");
      myArray[4][0] = 7;
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println(e);
```

Listing 1 . Listing for Answer 18.

```
    }//end catch

    System.out.println("Terminating, Dick
Baldwin");

  }//end main
}//End SampProg10 class.  Note no semicolon
required
```

Back to Question 18

Answer 17

The following statement declares and instantiates an array object capable of storing references to two **String** objects.

Note:

```
String[] myArrayOfStringReferences = new
String[2];
```

Note however, that this array object doesn't contain the actual string data. Rather, it simply sets aside memory for storage of two references to **String** objects. No memory has been set aside to store the characters that make up the individual strings. You must allocate the memory for the actual **String** objects separately using code similar to the following.

Note:

```
myArrayOfStringReferences[0] = new String(
                    "This is the first string.");
myArrayOfStringReferences[1] = new String(
                   "This is the second string.");
```

Back to Question 17

Answer 16

Java supports string concatenation using the overloaded + operator as shown in the following code fragment:

Note:

```
"My variable has a value of " + myVar +
" at this point in the program."
```

Back to Question 16

Answer 15

String objects cannot be modified once they have been created.
StringBuffer objects can be modified

Back to Question 15

Answer 14

The Java compiler considers a series of characters surrounded by quotation marks to be a literal string, as in the following code fragment:

Note:

```
"This is a literal string in Java."
```

Back to Question 14

Answer 13

A string is commonly considered to be a sequence of characters stored in memory and accessible as a unit. Java implements strings using the **String** class and the **StringBuffer** class.

Back to Question 13

Answer 12

Java allows you to assign one array to another. When you do this, you are simply making another copy of the reference to the same data in memory. Then you have two references to the same data in memory. This is illustrated in the program named **array02.java** in Jb0240: Java OOP: Arrays and Strings .

Back to Question 12

Answer 11

False. When declaring a two-dimensional array, it is not necessary to declare the size of the secondary dimension when the array is declared. Declaration of the size of each sub-array can be deferred until later as illustrated in the program named **array01.java** in Jb0240: Java OOP: Arrays and Strings .

Back to Question 11

Answer 10

False. Java can be used to produce multi-dimensional arrays that can be viewed as an array of arrays. However, the secondary arrays need not all be of the same size. See the program named **array01.java** in Jb0240: Java OOP: Arrays and Strings .

Back to Question 10

Answer 9

Array objects can contain any Java data type including primitive values, references to ordinary objects, and references to other array objects.

Back to Question 9

Answer 8

All array objects have a **length** property that can be accessed to determine the number of elements in the array as shown below.

Note:

```
for(int cnt = 0; cnt < myArray.length; cnt++)
  myArray[cnt] = cnt;
```

Back to Question 8

Answer 7

False. Array indices always begin with 0.

Back to Question 7

Answer 6

False. While it is possible to simultaneously declare the name of the variable and cause memory to be allocated to contain the array object, it is not necessary to combine these two processes. You can execute one statement to declare the variable and another statement to cause the memory for the array object to be allocated as shown below.

Note:

```
int[] myArray;
.
.
.
myArray = new int[25];
```

Back to Question 6

Answer 5

False. As with other objects. the declaration of the variable does not allocate memory to contain the array object. Rather it simply allocates memory to contain a reference to the array object. Memory to contain the array object must be allocated from dynamic memory using statements such as the following.

Note:

```
int[] myArray = new int[15];
int myArray[] = new int[25];
int[] myArray = {1,2,3,4,5}
```

Back to Question 5

Answer 4

Note:

```
int[] myArray;
```

```
int myArray[];
```

Back to Question 4

Answer 3

In declaring the variable, you must provide two important pieces of information:

- the name of the variable
- the type of the variable, which indicates the type of data to be stored in the array

Back to Question 3

Answer 2

False. Java has a true array type and a true **String** type with protective features to prevent your program from writing outside the memory bounds of the array or the **String** .

Back to Question 2

Answer 1

True.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0240r Review: Arrays and Strings
- File: Jb0240r.htm
- Originally published: 1997
- Published at cnx.org: 11/26/12

-end-

Jb0250: Java OOP: Brief Introduction to Exceptions
This module provides a very brief treatment of exception handling. The topic is discussed in detail in the module titled Java OOP: Exception Handling by Richard Baldwin

Revised: Tue Mar 29 09:51:41 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module provides a very brief treatment of exception handling. The topic is discussed in detail in the module titled Java OOP: Exception Handling . The topic is included in this *Programming Fundamentals* section simply to introduce you to the concept.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following link to easily find and view the listing while you are reading about it.

Listings

- Listing 1 . The program named simple1.

Discussion

What is an exception?

According to The Java Tutorials , *"An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions."*

A very common example of an exception given in textbooks is code that attempts to divide by zero *(this is easy to demonstrate)* .

Throwing an exception

Common terminology states that when this happens, the system *throws an exception* . If a thrown exception is not *caught* , a runtime error may occur.

Purpose of exception handling

The purpose of exception handling is to make it possible for the program to either attempt to recover from the problem, or at worst shut down the program in a graceful manner, whenever an exception occurs.

Java supports exception handling

Java, C++, and some other programming languages support exception handling in similar ways.

In Java, the exception can be thrown either by the system or by code created by the programmer. There is a fairly long list of exceptions that will be thrown automatically by the Java runtime system.

Checked exceptions cannot be ignored

Included in that long list of automatic exceptions is a subset known as "checked" exceptions. Checked exceptions cannot be ignored by the programmer. A method must either specify *(declare)* or catch all "checked" exceptions that can be thrown in order for the program to compile.

An example of specifying an exception

I explain the difference between specifying and catching an exception in Java OOP: Exception Handling . For now, suffice it to say that the code that begins with the word "throws" in Listing 1 specifies *(declares)* an exception that can be thrown by the code inside the **main** method.

If this specification is not made, the program will not compile.

Listing 1 . The program named simple1.

```
/*File simple1.java Copyright 1997, R.G.Baldwin
************************************************************/
class simple1 { //define the controlling class
  public static void main(String[] args)
                                  throws java.io.IOException {
    int ch1, ch2 = '0';

    System.out.println(
      "Enter some text, terminate with #");

    //Get and save individual bytes
    while( (ch1 = System.in.read() ) != '#') ch2 = ch1;

    //Display the character immediately before the #
    System.out.println("The char before the # was "
      + (char)ch2);
  }//end main
}//End simple1 class.
```

The program in Listing 1 does not throw any exceptions directly nor does it attempt to catch any exceptions. However, it can throw exceptions indirectly through its call to **System.in.read** .

Because **IOException** is a checked exception, the **main** method must either specify it or catch it . Otherwise the program won't compile. In this case, the **main** method specifies the exception as opposed to catching it.

Very brief treatment

As mentioned earlier, this is a very brief treatment of a fairly complex topic that is discussed in much more detail in the module titled Java OOP: Exception Handling . The topic was included at this point simply to introduce you to the concept of exceptions.

Run the program

II encourage you to copy the code from Listing 1 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library
- Java OOP: Objects and Encapsulation

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0250: Java OOP: Brief Introduction to Exceptions
- File: Jb0250.htm
- Originally published: 1997
- Published at cnx.org: 11/26/12

-end-

Jb0260: Java OOP: Command-Line Arguments
Although the use of command-line arguments is rare is this time of Graphical User Interfaces (GUI), they are still useful for testing and debugging code. This module explains the use of command-line arguments in Java.

Revised: Tue Mar 29 10:02:05 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

Although the use of command-line arguments is rare is this time of Graphical User Interfaces *(GUI)* , they are still useful for testing and debugging code. This module explains the use of command-line arguments in Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following link to easily find and view the listing while you are reading about it.

Listings

Discussion

Familiar example from DOS

Java programs can be written to accept command-line-arguments.

DOS users will be familiar with commands such as the following:

Note: Familiar DOS command

```
copy fileA fileB
```

In this case, **copy** is the name of the program to be executed, while **fileA** and **fileB** are command-line arguments.

Java syntax for command-line arguments

The Java syntax for supporting command-line arguments is shown below *(note the formal argument list for the **main** method)* .

Note: Java syntax for command-line arguments

```
public static void main(String[] args){
. . .
}//end main method
```

In Java, the formal argument list for the **main** method must appear in the method signature whether or not the program is written to support the use of command-line arguments. If the argument isn't used, it is simply ignored.

Where the arguments are stored

The parameter **args** contains a reference to a one-dimensional array object of type **String** .

Each of the elements in the array *(including the element at index zero)* contains a reference to an object of type **String** . Each object of type String encapsulates one command-line argument.

The number of arguments entered by the user

Recall from an earlier module on arrays that the number of elements in a Java array can be obtained from the **length** property of the array. Therefore, the number of arguments entered by the user is equal to the value of the **length** property. If the user didn't enter any arguments, the value will be zero.

Command-line arguments are separated by the space character. If you need to enter an argument that contains a space, surround the entire argument with quotation mark characters as in *"My command line argument"* .

The first command-line argument is encapsulated in the **String** object referred to by the contents of the array element at index 0, the second argument is referred to by the element at index 1, etc.

Sample Java program

The sample program in Listing 1 illustrates the use of command-line arguments.

Listing 1 . Illustration of command-line arguments.

Listing 1 . Illustration of command-line arguments.

```
/*File cmdlin01.java Copyright 1997, R.G.Baldwin
This Java application illustrates the use of Java
command-line arguments.

When this program is run from the command line as
follows:

java cmdlin01 My command line arguments

the program produces the following output:

My
command
line
arguments
**************************************************/
class cmdlin01 { //define the controlling class
  public static void main(String[] args){ //main method
    for(int i=0; i < args.length; i++)
    System.out.println( args[i] );
  }//end main
}//End cmdlin01 class.
```

The output from running this program for a specific input is shown in the comments at the beginning of the program.

Run the program

I encourage you to copy the code Listing 1 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library
- Java OOP: Objects and Encapsulation

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0260: Java OOP: Command-Line Arguments
- File: Jb0260.htm
- Originally published: 1997
- Published at cnx.org: 11/27/12

-end-

Jb0260r Review
This module contains review questions and answers keyed to the module titled Jb0260: Java OOP: Command-Line Arguments.

Revised: Tue Mar 29 10:09:37 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0260: Java OOP: Command-Line Arguments .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

Provide a common example of a command-line statement that illustrates the use of command-line-arguments.

Answer 1

Question 2

Describe the purpose of command-line-arguments.

Answer 2

Question 3

True or false? In Java, syntax provisions must be made in the method signature for the **main** method to accommodate command-line-arguments even if the remainder of the program is not designed to make use of them. If False, explain why.

Answer 3

Question 4

Provide the method signature for the **main** method in a Java application that is designed to accommodate the use of command-line-arguments. Identify the part of the method signature that applies to command-line-arguments and explain how it works.

Answer 4

Question 5

Explain how a Java application can determine the number of command-line-arguments actually entered by the user.

Answer 5

Question 6

Write a program that illustrates the handling of command-line arguments in Java.

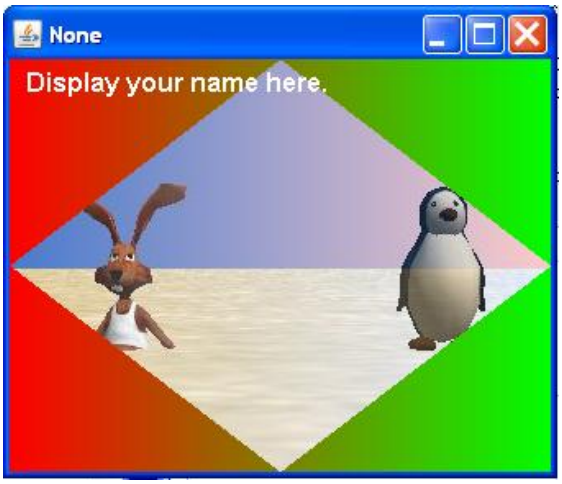
Answer 6

Listings

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 6

Listing 1 . Handling command-line arguments in Java.

Listing 1 . Handling command-line arguments in Java.

```
/*File SampProg11.java from module 32
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that illustrates the handling of command-line
arguments in Java.

Provide a termination message that displays your name.
**************************************************************/
class SampProg11 { //define the controlling class
  public static void main(String[] args){ //define main
    for(int i=0; i < args.length; i++)
      System.out.println( args[i] );
    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//End SampProg11 class.
```

Back to Question 6

Answer 5

The number of command-line arguments is equal to the number of elements in the array of references to **String** objects referred to by **args** . The number of elements is indicated by the value of the **length** property of the array. If the value is zero, the user didn't enter any command-line arguments.

Back to Question 5

Answer 4

The Java syntax for command-line arguments is shown below.

Note: Java syntax for command-line arguments.

```
public static void main(String[] args){
```

```
. . .
}//end main method
```

Each of the elements in the array object referred to by **args** *(including the element at position zero)* contains a reference to a **String** object that encapsulates one of the command-line arguments.

Back to Question 4

Answer 3

True.

Back to Question 3

Answer 2

Command-line-arguments are used in many programming and computing environments to provide information to the program at startup that it will need to fulfill its mission during that particular invocation.

Back to Question 2

Answer 1

DOS users will be familiar with commands such as the following:

Note: Command-line arguments in DOS

```
copy fileA fileB
```

In this case, **copy** is the name of the program to be executed, while **fileA** and **fileB** are command-line arguments.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0260r Review: Command-Line Arguments
- File: Jb0260r.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12

-end-

Jb0270: Java OOP: Packages
This module explains the concept of packages and provides a sample program that illustrates the concept.

Revised: Tue Mar 29 10:38:11 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module explains the concept of packages and provides a sample program that illustrates the concept.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

- Listing 1 . File: Package00.java.
- Listing 2 . File Package01.java.
- Listing 3 . File Package02.java.
- Listing 4 . File: CompileAndRun.bat.

Introduction

Before you can understand much about packages, you will need to understand the *classpath environment variable* , so that is where I will begin the discussion.

After learning about the classpath environment variable, you will learn how to create your own packages.

Classpath environment variable

The purpose of the *classpath environment variable* is to tell the JVM and other Java applications *(such as the javac compiler)* where to find class files and class libraries. This includes those class files that are part of the JDK and class files that you may create yourself.

I am assuming that you already have some familiarity with the use of environment variables in your operating system. All of the discussion in this module will be based on the use of a generic form of Windows. *(By generic, I mean not tied to any particular version of Windows.)* Hopefully you will be able to translate the information to your operating system if you are using a different operating system.

Note: In a nutshell
Environment variables provide information that the operating system uses to do its job.

There are usually a fairly large number of environment variables installed on a machine at any give time. If you would like to see what kind of environment variables are currently installed on your machine, bring up a command-line prompt and enter the command **set** . This should cause the names of several environment variables, along with their settings to be displayed on your screen.

While you are at it, see if any of those items begin with **CLASSPATH=** . If so, you already have a classpath environment variable set on your machine, but it may not contain everything that you need.

I am currently using a Windows 7 operating system and no classpath environment variable is set on it. I tend to use the **-cp** switch option *(see Listing 4)* in the JDK to set the classpath on a temporary basis when I need it to be set.

Rather than trying to explain all of the ramifications regarding the classpath, I will simply refer you to an Oracle document on the topic titled Setting the class path .

I will also refer you to Java OOP: The Guzdial-Ericson Multimedia Class Library where I discuss the use of the classpath environment variable with a Java multimedia class library.

Some rules

There are some rules that you must follow when working with the classpath variable, and if you fail to do so, things simply won't work.

For example, if your class files are in a jar file, the classpath must end with the name of that jar file.

On the other hand, if the class files are not in a jar file, the classpath must end with the name of the folder that contains the class files.

Your classpath must contain a fully-qualified path name for every folder that contains class files of interest, or for every jar file of interest. The paths should begin with the letter specifying the drive and end either with the name of the jar file or the name of the folder that contains the class files. .

If you followed the default JDK installation procedure and are simply compiling and executing Java programs in the current directory you probably won't need to set the classpath. By default, the system already knows *(or can figure out)* how to allow you to compile and execute programs in the current directory and how to use the JDK classes that come as part of the JDK.

However, if you are using class libraries other than the standard Java library, are saving your class files in one or more different folders, or are ready to start creating your own packages, you will need to set the classpath so that the system can find the class files in your packages.

Developing your own packages

One of the problems with storing all of your class files in one or two folders is that you will likely experience name conflicts between class files.

Every Java program can consist of a large number of separate classes. A class file is created for each class that is defined in your program, even if they are all combined into a single source file.

It doesn't take very many programs to create a lot of class files, and it isn't long before you find yourself using the same class names over again. If you do, you will end up overwriting class files that were previously stored in the folder.

For me, it only takes two GUI programs to get in trouble because I tend to use the same class names in every program for certain standard operations such as closing a **Frame** or processing an **ActionEvent** . For the case of the **ActionEvent** , the body of the class varies from one application to the next so it doesn't make sense to turn it into a library class.

So we need a better way to organize our class files, and the Java package provides that better way.

The Java package approach allows us to store our class files in a hierarchy of folders *(or a jar file that represents that hierarchy)* while only requiring that the classpath variable point to the top of the hierarchy. The remaining hierarchy structure is encoded into our programs using package directives and import directives.

Now here is a little jewel of information that cost me about seven hours of effort to discover when I needed it badly.

When I first started developing my own packages, I spent about seven hours trying to determine why the compiler wouldn't recognize the top-level folder in my hierarchy of package folders.

I consulted numerous books by respected authors and none of them was any help at all. I finally found the following statement in the Java documentation *(when all else fails, read the documentation)* . By the way, a good portion of that wasted seven hours was spent trying to find this information in the documentation which is voluminous.

Note: The following text was extracted directly from the JDK 1.1 documentation
If you want the CLASSPATH to point to class files that belong to a package, you should specify a path name that includes the path to the directory one level above the directory having the name of your package.
For example, suppose you want the Java interpreter to be able to find classes in the package mypackage. If the path to the mypackage directory is C:\java\MyClasses\mypackage, you would set the CLASSPATH variable as follows:
set CLASSPATH=C:\java\MyClasses

If you didn't catch the significance of this, read it again. When you are creating a classpath variable to point to a folder containing classes, it must point to the folder. However, when you are creating a classpath variable to point to your package, it must point to the folder that is one level above the directory that is the top-level folder in your package.

Once I learned that I had to cause the classpath to point to the folder immediately above the first folder in the hierarchy that I was including in my package directives, everything started working.

The package directive

So, what is the purpose of a package directive, and what does it look like?

Note: Purpose of a package directive
The purpose of the package directive is to identify a particular class *(or group of classes contained in a single source file (compilation unit))* as belonging to a specific package.

This is very important information for a variety of reasons, not the least of which is the fact that the entire access control system is wrapped around the concept of a class belonging to a specific package. For example, code in one package can only access public classes in a different package.

Stated in the simplest possible terms, a package is a group of class files contained in a single folder on your hard drive.

At compile time, a class can be identified as being part of a particular package by providing a package directive at the beginning of the source code..

A package directive, if it exists, must occur at the beginning of the source code *(ignoring comments and white space)* . No text other than comments and whitespace is allowed ahead of the package directive.

If your source code file does not contain a package directive, the classes in the source code file become part of the *default package* . With the exception of the default package, all packages have names, and those names are the same as the names of the folders that contain the packages. There is a one-to-one correspondence between folder names and package names. The default package doesn't have a name.

Some examples of package directives that you will see in upcoming sample programs follow:

Note: Example package directives

```
package Combined.Java.p1;
package Combined.Java.p2;
```

Given the following as the classpath on my hypothetical machine,

CLASSPATH=.;c:\Baldwin\JavaProg

these two package directives indicate that the class files being governed by the package directives are contained in the following folders:

Note:

```
c:\Baldwin\JavaProg\Combined\Java\p1
c:\Baldwin\JavaProg\Combined\Java\p2
```

Notice how I concatenated the package directive to the classpath setting and substituted the backslash character for the period when converting from the package directive to the fully-qualified path name.

Code in one package can refer to a class in another package *(if it is otherwise accessible)* by qualifying the class name with its **package name as follows** :

Note:

```
Combined.Java.p2.Access02 obj02 =
       new Combined.Java.p2.Access02();
```

Obviously, if we had to do very much of that, it would become very burdensome due to the large amount of typing required. As you already know, the *import directive* is available to allow us to specify the package containing the class just once at the beginning of the source file and then refer only to the class name for the remainder of that source file.

The import directive

This discussion will be very brief because you have been using import directives since the very first module. Therefore, you already know what they look like and how to use them.

If you are interested in the nitty-gritty details *(such as what happens when you provide two import directives that point to two different packages containing the same class file name)* ,

you can consult the Java Language Reference by Mark Grand, or you can download the Java language specification from Oracle's Java website.

The purpose of the import directive is to help us avoid the burdensome typing requirement described in the previous section when referring to classes in a different package.

An import directive makes the definitions of classes from other packages available on the basis of their file names alone.

You can have any number of import directives in a source file. However, they must occur after the package directive *(if there is one)* and before any class or interface declarations.

There are essentially two different forms of the import directive, one with and the other without a wild card character *(*)* . These two forms are illustrated in the following box.

Note: Two forms of import directives

```
import java.awt.*
import java.awt.event.ActionEvent
```

The first import directive makes all of the class files in the **java.awt** package available for use in the code in a different package by referring only to their file names.

The second import directive makes only the class file named **ActionEvent** in the **java.awt.event** package available by referring only to the file name.

Compiling programs with package directives

So, how do you compile programs containing package directives? There are probably several ways. I am going to describe one way that I have found to be successful.

First, you must create your folder hierarchy to match the package directive that you intend to use. Remember to construct this hierarchy downward relative to the folder specified at the end of your classpath setting. If you have forgotten the critical rule in this respect, go back and review it.

Next, place source code files in each of the folders where you intend for the class files associated with those source code files to reside. *(After you have compiled and tested the program, you can remove the source code files if you wish.)*

You can compile the source code files individually if you want to, but that isn't necessary.

One of the source code files will contain the *controlling class* . The controlling class is the class that contains the **main** method that will be executed when you run the program from the command line using the JVM.

Make the directory containing that source code file be the current directory. *(If you don't know what the current directory is, go out and get yourself a **DOS For Dummies** book and read it.)*

Each of the source code files must contain a package directive that specifies the package that will contain the compiled versions of all the class definitions in that source code file. Using the instructions that I am giving you, that package directive will also describe the folder that contains the source code file.

Any of the source code files containing code that refers to classes in a different package must also contain the appropriate import directives, or you must use fully-qualified package names to refer to those classes.

Then use the **javac** program with your favorite options to compile the source code file containing the controlling class. This will cause all of the other source code files containing classes that are linked to the code in the controlling class, either directly or indirectly, to be compiled also. At least an attempt will be made to compile them all. You may experience a few compiler errors if your first attempt at compilation is anything like mine.

Once you eliminate all of the compiler errors, you can test the application by using the **java** program with your favorite options to execute the controlling class.

Once you are satisfied that everything works properly, you can copy the source code files over to an archive folder and remove them from the package folders if you want to do so.

Finally, you can also convert the entire hierarchy of package folders to a jar file if you want to, and distribute it to your client. If you don't remember how to install it relative to the classpath variable, go back and review that part of the module.

Once you have reached this point, how do you execute the program. I will show you how to execute the program from the command line in the sample program in the next section. *(Actually I will encapsulate command-line commands in a batch file and execute the batch file. That is a good way to guard against typing errors.)*

Sample program

The concept of packages can get very tedious in a hurry. Let's take a look at a sample program that is designed to pull all of this together.

This application consists of three separate source files located in three different packages. Together they illustrates the use of package and import directives, along with **javac** to build a standalone Java application consisting of classes in three separate packages.

(If you want to confirm that they are really in different packages, just make one of the classes referred to by the controlling class non-public and try to compile the program.)

In other words, in this sample program, we create our own package structure and populate it with a set of cooperating class files.

A folder named **jnk** is a child of the root folder on the M-drive.

A folder named **SampleCode** is a child of the folder named **jnk** .

A folder named **Combined** is a child of the folder named **SampleCode** .

A folder named **Java** is a child of the folder named **Combined** .

Folders named **p1** and **p2** are children of the folder named **Java** .

The file named **Package00.java** , shown in Listing 1 is stored in the folder named **Java** .

Listing 1 . File: Package00.java.

Listing 1 . File: Package00.java.

```
/*File Package00.java Copyright 1997, R.G.Baldwin
Illustrates use  of package and import directives to
build an application consisting of classes in three
separate packages.

The output from running the program follows:

Starting Package00
Instantiate obj of public classes in different packages
Constructing Package01 object in folder p1
Constructing Package02 object in folder p2
Back in main of Package00
**********************************************************/
package Combined.Java; //package directive

//Two import directives
import Combined.Java.p1.Package01;//specific form
import Combined.Java.p2.*; //wildcard form

class Package00{
  public static void main(String[] args){ //main method
    System.out.println("Starting Package00");

    System.out.println("Instantiate obj of public " +
                       "classes in different packages");
    new Package01();//Instantiate objects of two classes
    new Package02();// in different packages.

    System.out.println("Back in main of Package00");

  }//end main method
}//End Package00 class definition.
```

The file named **Package01.java** , shown in Listing 2 is stored in the folder named **p1** .

Listing 2 . File Package01.java.

```
/*File Package01.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
**************************************************/
package Combined.Java.p1;
public class Package01 {
  public Package01(){//constructor
    System.out.println(
              "Constructing Package01 object in folder p1");
  }//end constructor
}//End Package01 class definition.
```

The file named **Package02.java** , shown in Listing 3 is stored in the folder named **p2** .

Listing 3 . File Package02.java.

```
/*File Package02.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
**************************************************/
package Combined.Java.p2;
public class Package02 {
  public Package02(){//constructor
    System.out.println(
              "Constructing Package02 object in folder p2");
  }//end constructor
}//End Package02 class definition.
```

The file named **CompileAndRun** .bat, shown in Listing 4 is stored in the folder named **SampleCode** .

Listing 4 . File: CompileAndRun.bat.

```
echo off
rem This file is located in folder named M:\SampleCode,
rem which is Parent of folder Combined.

del Combined\Java\*.class
del Combined\Java\p1\*.class
del Combined\Java\p2\*.class

javac -cp M:\jnk\SampleCode Combined\Java\Package00.java

java -cp M:\jnk\SampleCode Combined.Java.Package00

pause
```

The controlling class named **Package00** is stored in the package named **Combined.Java** , as declared in Listing 1 .

The class named **Package01** is stored in the package named **Combined.Java.p1** , as declared in Listing 2 .

The class named **Package02** is stored in the package named **Combined.Java.p2** , as declared in Listing 3 .

The controlling class named **Package00** imports **Combined.Java.p1.Package01** and **Combined.Java.p2.*** , as declared in Listing 1 .

Code in the **main** method of the controlling class in Listing 1 instantiates objects of the other two classes in different packages. The constructors for those two classes announce that they are being constructed.

The two classes being instantiated are **public** . Otherwise, it would not be possible to instantiate them from outside their respective packages.

This program was tested using JDK 7 under Windows by executing the batch file named **CompileAndRun.bat** .

The classpath is set to the parent folder of the folder named **Combined** *(M:\jnk\SampleCode)* by the **-cp** switch in the file named **CompileAndRun.bat** .

The output from running the program is shown in the comments at the beginning of Listing 1 .

Run the program

I encourage you to copy the code from Listing1 through Listing 4 into a properly constructed tree of folders. Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library
- Java OOP: Objects and Encapsulation

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0270: Java OOP: Packages
- File: Jb0270.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12

-end-

Jb0280: Java OOP: String and StringBuffer
This module discusses the String and StringBuffer classes in detail.

Revised: Tue Mar 29 11:14:56 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module discusses the **String** and **StringBuffer** classes in detail.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

Introduction

A string in Java is an object. Java provides two different string classes from which objects that encapsulate string data can be instantiated:

- **String**
- **StringBuffer**

The **String** class is used for strings that are not allowed to change once an object has been instantiated *(an immutable object)* . The **StringBuffer** class is used for strings that may be modified by the program.

You can't modify a String object, but you can replace it

While the contents of a **String** object cannot be modified, a reference to a **String** object can be caused to point to a different **String** object as illustrated in the sample program shown in Listing 1 . Sometimes this makes it appear that the original **String** object is being modified.

Listing 1 . File String01.java.

Listing 1 . File String01.java.

```
/*File String01.java Copyright 1997, R.G.Baldwin
This application illustrates the fact that while a String
object cannot be modified, the reference variable can be
modified to point to a new String object which can have
the appearance of modifying the original String object.

The program was tested using JDK 1.1.3 under Win95.

The output from this program is

Display original string values
THIS STRING IS NAMED str1
This string is named str2
Replace str1 with another string
Display new string named str1
THIS STRING IS NAMED str1 This string is named str2
Terminating program

**********************************************************/

class String01{
  String str1 = "THIS STRING IS NAMED str1";
  String str2 = "This string is named str2";

  public static void main(String[] args){
    String01 thisObj = new String01();
    System.out.println("Display original string values");
    System.out.println(thisObj.str1);
    System.out.println(thisObj.str2);
    System.out.println("Replace str1 with another string");
    thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
    System.out.println("Display new string named str1");
    System.out.println(thisObj.str1);
    System.out.println("Terminating program");
  }//end main()
}//end class String01
```

It is important to note that the following statement does not modify the original object pointed to by the reference variable named **str1** .

Note:

```
thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
```

Rather, this statement creates a new object, which is concatenation of two existing objects and causes the reference variable named **str1** to point to the new object instead of the original object.

The original object then becomes eligible for garbage collection *(unless there is another reference to the object hanging around somewhere)* .

Many aspects of string manipulation can be accomplished in this manner, particularly when the methods of the **String** class are brought into play.

Why are there two string classes?

According to *The Java Tutorial* by Campione and Walrath:

Note:
"Because they are constants, Strings are typically cheaper than StringBuffers and they can be shared. So it's important to use Strings when they're appropriate."

Creating String and StringBuffer objects

The **String** and **StringBuffer** classes have numerous overloaded constructors and many different methods. I will attempt to provide a sampling of constructors and methods that will prepare you to explore other constructors and methods on your own.

The next sample program touches on some of the possibilities provided by the wealth of constructors and methods in the **String** and **StringBuffer** classes.

At this point, I will refer you to Java OOP: Java Documentation where you will find a link to online Java documentation. Among other things, the online documentation provides a list of the overloaded constructors and methods for the **String** and **StringBuffer** classes.

As of Java version 7, there are four overloaded constructors in the **StringBuffer** class and about thirteen different overloaded versions of the **append** method. There are many additional methods in the **StringBuffer** class including about twelve overloaded versions of the **insert** method.

As you can see, there are lots of constructors and lots of methods from which to choose. One of your challenges as a Java programmer will be to find the right methods of the right classes to accomplish what you want your program to accomplish.

The sample program named String02

The sample program shown in Listing 2 illustrates a variety of ways to create and initialize **String** and **StringBuffer** objects.

Listing 2 . File String02.java.

```
/*File String02.java Copyright 1997, R.G.Baldwin
Illustrates different ways to create String objects and
StringBuffer objects.

The program was tested using JDK 1.1.3 under Win95.

The output from this program is as follows.  In some cases,
manual line breaks were inserted to make the material fin
this presentation format.

Create a String the long way and display it
String named str2

Create a String the short way and display it
String named str1

Create, initialize, and display a StringBuffer using new
StringBuffer named str3

Try to create/initialize StringBuffer without
using new - not allowed

Create an empty StringBuffer of default length
Now put some data in it and display it
StringBuffer named str5
```

Listing 2 . File String02.java.

```
Create an empty StringBuffer and specify length
when it is created
Now put some data in it and display it
StringBuffer named str6

Try to create and append to StringBuffer without
using new -- not allowed

*********************************************************/

class String02{
  void d(String displayString){//method to display strings
    System.out.println(displayString);
  }//end method d()

  public static void main(String[] args){
    String02 o = new String02();//obj of controlling class

    o.d("Create a String the long way and display it");
    String str1 = new String("String named str2");
    o.d(str1 + "\n");

    o.d("Create a String the short way and display it");
    String str2 = "String named str1";
    o.d(str2 + "\n");

    o.d("Create, initialize, and display a StringBuffer " +
                                              "using new");
    StringBuffer str3 = new StringBuffer(
                                 "StringBuffer named str3");
    o.d(str3.toString()+"\n");

    o.d("Try to create/initialize StringBuffer without " +
                               "using new - not allowed\n");
    //StringBuffer str4 = "StringBuffer named str4";x

    o.d("Create an empty StringBuffer of default length");
    StringBuffer str5 = new StringBuffer();

    o.d("Now put some data in it and display it");
    //modify length as needed
    str5.append("StringBuffer named str5");
    o.d(str5.toString() + "\n");
```

Listing 2 . File String02.java.

```
    o.d("Create an empty StringBuffer and specify " +
                              "length when it is created");
    StringBuffer str6 = new StringBuffer(
                      "StringBuffer named str6".length());
    o.d("Now put some data in it and display it");
    str6.append("StringBuffer named str6");
    o.d(str6.toString() + "\n");

    o.d("Try to create and append to StringBuffer " +
                      "without using new -- not allowed");
    //StringBuffer str7;
    //str7.append("StringBuffer named str7");
  }//end main()
}//end class String02
```

Alternative String instantiation constructs

The first thing to notice is that a **String** object can be created using either of the following constructs:

Note: Alternative String instantiation constructs

```
   String str1 = new String("String named str2");

   String str2 = "String named str1";
```

The first approach uses the **new** operator to instantiate an object while the shorter version doesn't use the new operator.

Later I will discuss the fact that

- the second approach is not simply a shorthand version of the first construct, but that
- they involve two different compilation scenarios with the second construct being more efficient than the first.

Instantiating StringBuffer objects

The next thing to notice is that a similar alternative strategy does not hold for the **StringBuffer** class.

For example, it is not possible to create a **StringBuffer** object without use of the **new** operator. *(It is possible to create a reference to a **StringBuffer** object but it is later necessary to use the **new** operator to actually instantiate an object.)*

Note the following code fragments that illustrate allowable and non-allowable instantiation scenarios for **StringBuffer** objects.

Note: Instantiating StringBuffer objects

```
//allowed
StringBuffer str3 = new StringBuffer(
                      "StringBuffer named str3");

//not allowed
//StringBuffer str4 = "StringBuffer named str4";


o.d("Try to create and append to StringBuffer " +
             "without using new -- not allowed");
//StringBuffer str7;
//str7.append("StringBuffer named str7");
```

Declaration, memory allocation, and initialization

To review what you learned in an earlier module, three steps are normally involved in creating an object *(but the third step may be omitted)* .

- declaration
- memory allocation
- initialization

The following code fragment performs all three steps:

Note: Declaration, memory allocation, and initialization

```
StringBuffer str3 =
  new StringBuffer("StringBuffer named str3");
```

The code

```
StringBuffer str3
```

declares the type and name of a reference variable of the correct type for the benefit of the compiler.

The **new** operator allocates memory for the new object.

The constructor call

```
StringBuffer("StringBuffer named str3")
```

constructs and initializes the object.

Instantiating an empty StringBuffer object

The instantiation of the **StringBuffer** object shown above uses a version of the constructor that accepts a **String** object and initializes the **StringBuffer** object when it is created.

The following code fragment instantiates an empty **StringBuffer** object of a default capacity and then uses a version of the **append** method to put some data into the object. *(Note that the data is actually a **String** object -- a sequence of characters surrounded by quotation marks.)*

Note: Instantiating an empty StringBuffer object

```
//default initial length
StringBuffer str5 = new StringBuffer();

//modify length as needed
str5.append("StringBuffer named str5");
```

It is also possible to specify the capacity when you instantiate a **StringBuffer** object.

Some authors suggest that if you know the final length of such an object, it is more efficient to specify that length when the object is instantiated than to start with the default length and then require the system to increase the length "on the fly" as you manipulate the object.

This is illustrated in the following code fragment. This fragment also illustrates the use of the **length** method of the **String** class just to make things interesting. *(A simple integer value for the capacity of the **StringBuffer** object would have worked just as well.)*

Note: Instantiating a StringBuffer object of a non-default length

```
StringBuffer str6 = new StringBuffer(
            "StringBuffer named str6".length());
str6.append("StringBuffer named str6");
```

Accessor methods

The following quotation is taken directly from *The Java Tutorial* by Campione and Walrath.

Note:
"An object's instance variables are encapsulated within the object, hidden inside, safe from inspection or manipulation by other objects. With certain well-defined exceptions, the object's methods are the only means by which other objects can inspect or alter an object's instance variables. Encapsulation of an object's data protects the object from corruption by other objects and conceals an object's implementation details from outsiders. This encapsulation of data behind an object's methods is one of the cornerstones of object-oriented programming."

The above statement lays out an important consideration in good object-oriented programming.

The methods used to obtain information about an object are often referred to as *accessor methods* .

Constructors and methods of the String class

I told you in an earlier section that the **StringBuffer** class provides a large number of overloaded constructors and methods. The same holds true for the **String** class.

Once again, I will refer you to Java OOP: Java Documentation where you will find a link to online Java documentation. Among other things, the documentation provides a list of the overloaded constructors and methods for the **String** class

String objects encapsulate data

The characters in a **String** object are not directly available to other objects. However, as you can see from the documentation, there are a large number of methods that can be used to access and manipulate those characters. For example, in an earlier sample program *(Listing 2)* , I used the **length** method to access the number of characters stored in a **String** object as shown in the following code fragment.

Note:

```
StringBuffer str6 = new StringBuffer(
                    "StringBuffer named str6".length());
```

In this case, I applied the **length** method to a literal string, but it can be applied to any valid representation of an object of type **String** .

I then passed the value returned by the **length** method to the constructor for a **StringBuffer** object.

As you can determine by examining the argument lists for the various methods of the **String** class,

- some methods return data stored in the string while
- other methods return information about that data.

For example, the **length** method returns information about the data stored in the **String** object.

Methods such as **charAt** and **substring** return portions of the actual data.

Methods such **toUpperCase** can be thought of as returning the data, but returning it in a different format.

Creating String objects without calling the constructor

Methods in other classes and objects may create **String** objects without an explicit call to the constructor by the programmer. For example the **toString** method of the **Float** class receives a **float** value as an incoming parameter and returns a reference to a **String** object that represents the **float** argument.

Memory management by the StringBuffer class

If the additional characters cause the size of the **StringBuffer** to grow beyond its current capacity when characters are added, additional memory is automatically allocated.

However, memory allocation is a relatively expensive operation and you can make your code more efficient by initializing **StringBuffer** capacity to a reasonable first guess. This will minimize the number of times memory must be allocated for it.

When using the **insert** methods of the **StringBuffer** class, you specify the index *before which* you want the data inserted.

The toString method

Frequently you will need to convert an object to a **String** object because you need to pass it to a method that accepts only **String** values *(or perhaps for some other reason)* .

All classes inherit the **toString** method from the **Object** class. Many of the classes *override* this method to provide an implementation that is meaningful for objects of that class.

In addition, you may sometimes need to *override* the **toString** method for classes that you define to provide a meaningful **toString** behavior for objects of that class.

I explain the concept of overriding the **toString** method in detail in the module titled Java OOP: Polymorphism and the Object Class .

Strings and the Java compiler

In Java, you specify literal strings between double quotes as in:

Note: Literal strings

```
"I am a literal string of the String type."
```

You can use literal strings anywhere you would use a **String** object.

You can also apply **String** methods directly to a literal string as in an earlier program that calls the **length** method on a literal string as shown below.

Note: Using String methods with literal strings

```
StringBuffer str6 = new StringBuffer(
                StringBuffer named str6".length());
```

Because the compiler automatically creates a new **String** object for every literal string, you can use a literal string to initialize a **String** object *(without use of the new operator)* as in the following code fragment from a previous program :

Note:

```
String str1 = "THIS STRING IS NAMED str1";
```

The above construct is equivalent to, but more efficient than the following, which, according to *The Java Tutorial* by Campione and Walrath, ends up creating two **String** objects instead of one:

Note:

```
String str1 = new String("THIS STRING IS NAMED str1");
```

In this case, the compiler creates the first **String** object when it encounters the literal string, and the second one when it encounters **new String()** .

Concatenation and the + operator

The plus (+) operator is overloaded so that in addition to performing the normal arithmetic operations, it can also be used to concatenate strings.

This will come as no surprise to you because we have been using code such as the following since the beginning of this group of *Programming Fundamentals* modules:

Note:

```
String cat = "cat";

System.out.println("con" + cat + "enation");
```

According to Campione and Walrath, Java uses **StringBuffer** objects behind the scenes to implement concatenation. They indicate that the above code fragment compiles to:

Note:

```
String cat = "cat";
System.out.println(new StringBuffer().append("con").
                       append(cat).append("enation"));
```

Fortunately, that takes place behind the scenes and we don't have to deal directly with the syntax.

Run the programs

I encourage you to copy the code from Listing 1 and Listing 2 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library
- Java OOP: Objects and Encapsulation

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0280: Java OOP: String and StringBuffer
- File: Jb0280.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0280r Review
This module contains review questions and answers keyed to the module titled Jb0280: Java OOP: String and StringBuffer.

Revised: Tue Mar 29 11:29:05 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

Preface

This module contains review questions and answers keyed to the module titled Jb0280: Java OOP: String and StringBuffer .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

Java provides two different string classes from which string objects can be instantiated. What are they?

Answer 1

Question 2

True or false? The **StringBuffer** class is used for strings that are not allowed to change. The **String** class is used for strings that are modified by the program. If false, explain why.

Answer 2

Question 3

True or false? While the contents of a **String** object cannot be modified, a reference to a **String** object can be caused to point to a different **String** object. If false, explain why.

Answer 3

Question 4

True or false? The use of the **new** operator is required for instantiation of objects of type **String** . If false, explain your answer.

Answer 4

Question 5

True or false? The use of the **new** operator is required for instantiation of objects of type **StringBuffer** . If false, explain your answer

Answer 5

Question 6

Provide a code fragment that illustrates how to instantiate an empty **StringBuffer** object of a default length and then use a version of the **append** method to put some data into the object.

Answer 6

Question 7

Without specifying any explicit numeric values, provide a code fragment that will instantiate an empty **StringBuffer** object of the correct initial length to contain the string *"StringBuffer named str6"* and then store that string in the object.

Answer 7

Question 8

Provide a code fragment consisting of a single statement showing how to use the **Integer** wrapper class to convert a string containing digits to an integer and store it in a variable of type **int** .

Answer 8

Question 9

Explain the difference between the **capacity** method and the **length** method of the **StringBuffer** class.

Answer 9

Question 10

True or false? The following is a valid code fragment. If false, explain why.

Note:

```
StringBuffer str6 =
  new StringBuffer("StringBuffer named str6".length());
```

Answer 10

Question 11

Which of the following code fragments is the most efficient, first or second?

Note:

```
String str1 = "THIS STRING IS NAMED str1";
```

```
String str1 = new String("THIS STRING IS NAMED str1");
```

Answer 11

Question 12

Write a Java application that illustrates the fact that while a **String** object cannot be modified, the reference variable can be modified to point to a new **String** object, which can have the appearance of modifying the original **String** object.

Answer 12

Question 13

Write a Java application that illustrates different ways to create **String** objects and **StringBuffer** objects.

Answer 13

Question 14

Write a Java application that illustrates conversion from string to numeric.

Answer 14

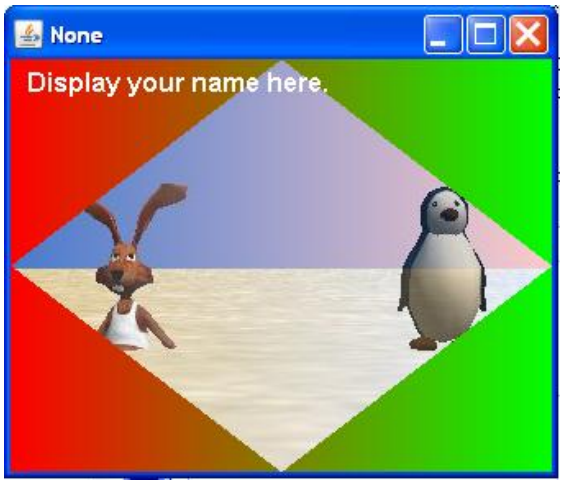
Listings

- Listing 1 . File SampProg26.java.
- Listing 2 . File SampProg25.java.
- Listing 3 . File SampProg24.java.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 14

Listing 1 . File SampProg26.java.

Listing 1 . File SampProg26.java.

```
/*File SampProg26.java from module 50
Copyright 1997, R.G.Baldwin
Without viewing the solution that follows, write a Java
application that illustrates conversion from string to
numeric, similar to the atoi() function in C.

The output from the program should be:
The value of the int variable num is 3625
============================================================
*/

class SampProg26{
  public static void main(String[] args){
    int num = new Integer("3625").intValue();
    System.out.println(
      "The value of the int variable num is " + num);
  }//end main()
}//end class SampProg26
```

Back to Question 14

Answer 13

Listing 2 . File SampProg25.java.

```
/*File SampProg25.java from module 50
Copyright 1997, R.G.Baldwin
Write a Java application that illustrates different ways to
create String objects and StringBuffer objects.

The output from this program should be (line breaks
```

Listing 2 . File SampProg25.java.

```
manually inserted to make it fit the format):

Create a String using new and display it
String named str2

Create a String without using new and display it
String named str1

Create, initialize, and display a StringBuffer using new
StringBuffer named str3

Try to create/initialize StringBuffer without using new

Create an empty StringBuffer of default length
Now put some data in it and display it
StringBuffer named str5

Create an empty StringBuffer and specify length when
created
Now put some data in it and display it
StringBuffer named str6

Try to create and append to StringBuffer without using new
**********************************************************/

class SampProg25{
  void d(String displayString){//method to display strings
    System.out.println(displayString);
  }//end method d()

  public static void main(String[] args){
    //instantiate an object to display methods
    SampProg25 o = new SampProg25();

    o.d("Create a String using new and display it");
    String str1 = new String("String named str2");
    o.d(str1 + "\n");

    o.d(
    "Create a String without using new and display it");
    String str2 = "String named str1";
    o.d(str2 + "\n");
```

Listing 2 . File SampProg25.java.

```
    o.d("Create, initialize, and display a StringBuffer "
      + "using new");
    StringBuffer str3 = new StringBuffer(
      "StringBuffer named str3");
    o.d(str3.toString()+"\n");

    o.d("Try to create/initialize StringBuffer without "
      + "using new \n");
    //StringBuffer str4 = //not allowed by compiler
    //  "StringBuffer named str4";

    o.d(
    "Create an empty StringBuffer of default length");
    //accept default initial length
    StringBuffer str5 = new StringBuffer();
    o.d("Now put some data in it and display it");
    //modify length as needed
    str5.append("StringBuffer named str5");
    o.d(str5.toString() + "\n");

    o.d("Create an empty StringBuffer and specify length "
      + "when created");
    StringBuffer str6 = new StringBuffer(
      "StringBuffer named str6".length());
    o.d("Now put some data in it and display it");
    str6.append("StringBuffer named str6");
    o.d(str6.toString() + "\n");

    o.d(
    "Try to create and append to StringBuffer without "
      + "using new");
    //StringBuffer str7;
    //str7.append("StringBuffer named str7");
  }//end main()
}//end class SampProg25
```

Back to Question 13

Answer 12

Listing 3 . File SampProg24.java.

```
/*File SampProg24.java from module 50
Copyright 1997, R.G.Baldwin
Without viewing the solution that follows, Write a Java
application that illustrates the fact that while a String
object cannot be modified, the reference variable can be
modified to point to a new String object which can have the
appearance of modifying the original String object.

The output from this program should be

Display original string values
THIS STRING IS NAMED str1
This string is named str2
Replace str1 with another string
Display new string named str1
THIS STRING IS NAMED str1 This string is named str2
Terminating program
**********************************************************/

class SampProg24{
  String str1 = "THIS STRING IS NAMED str1";
  String str2 = "This string is named str2";

  public static void main(String[] args){
    SampProg24 thisObj = new SampProg24();
    System.out.println("Display original string values");
    System.out.println(thisObj.str1);
    System.out.println(thisObj.str2);
    System.out.println(
      "Replace str1 with another string");
    thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
    System.out.println("Display new string named str1");
    System.out.println(thisObj.str1);
    System.out.println("Terminating program");
  }//end main()
}//end class SampProg24
```

Back to Question 12

Answer 11

The first code fragment is the most efficient.

Back to Question 11

Answer 10

True.

Back to Question 10

Answer 9

The **capacity** method returns the amount of space currently allocated for the **StringBuffer** object. The **length** method returns the amount of space used.

Back to Question 9

Answer 8

Note:

```
int num = new Integer("3625").intValue();
```

Back to Question 8

Answer 7

Note:

```
StringBuffer str6 =
```

```
  new StringBuffer("StringBuffer named str6".length());
str6.append("StringBuffer named str6");
```

Back to Question 7

Answer 6

Note:

```
StringBuffer str5 =
  new StringBuffer();//accept default initial length
str5.append(
      "StringBuffer named str5");//modify length as needed
```

Back to Question 6

Answer 5

True.

Back to Question 5

Answer 4

False. A String object can be instantiated using either of the following statements:

Note:

```
    String str1 = new String("String named str2");

    String str2 = "String named str1";
```

Back to Question 4

Answer 3

True.

Back to Question 3

Answer 2

False. This statement is backwards. The **String** class is used for strings that are not allowed to change. The **StringBuffer** class is used for strings that are modified by the program.

Back to Question 2

Answer 1

The two classes are:

- String
- StringBuffer

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0280r Review: String and StringBuffer
- File: Jb0280r.htm
- Originally published: 1997
- Published at cnx.org: 11/29/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jb0290: The end of Programming Fundamentals
This module signals the end of the section on Programming Fundamentals

Revised: Tue Mar 29 11:40:13 CDT 2016

This page is included in the following Books:

- *Programming Fundamentals with Java*
- *Object-Oriented Programming (OOP) with Java*

Looking ahead

You have now reached the end of this Programming Fundamentals book.

The next stop along your journey to become a Java/OOP programmer should be either the OOP Self-Assessment , or the course material for the ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library
- Java OOP: Objects and Encapsulation

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jb0290: Java OOP: The end of Programming Fundamentals
- File: Jb0290.htm
- Published: 11/29/12

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-