Java2 Certification last minute tutorial

The purpose of this tutorial

These pages are designed to help you pass the Sun Certified Java Programmers Exam. I have attempted to lay out the pages so they print out acceptably on A4 paper. It has been under active development for more than two years and incorporates feedback and corrections from many hundreds of people (thank you all). You can now purchase the pdf file of this tutorial with index. A pdf file preserves the font and layout details so you can view and print out a document as the author intended. It is commonly used to sell digital versions of books. For the zipped up pdf file go to Purchase CertKey for Java2 Pdf file

This tutorial is designed mainly for final cramming rather than in-depth study and learning. It assumes that you already know the basic principles of programming from a language such as C/C++ or Visual Basic and that you can set up your Java environment to create and run Java programs. It doesn't try to make you a good programmer, or even a good Java programmer, it just tries to get you through the exam by concentrating narrowly on the objectives.

It assumes that you know stuff like the difference between an application and an applet and that Java is case sensitive. If you don't know this type of stuff, get hold of a beginners Java tutorial and play with the language to get some understanding. You could do worse that going to the Sun web site and downloading the Sun Java Tutorial at http://www.javasoft.com/docs/books/tutorial/index.html

Another handy online Java Tutorial can be found at http://www.phrantic.com/scoop/onjava.html. Two other books for beginning Java programmers are Peter van der Lindens Just Java And Beyond and the O'Reilly Java In a Nutshell Deluxe Edition which comes with the Nutshell book plus the text on a CD ROM and the text of 4 other books on the CD ROM.

Can't afford the Technical Books?

If you find Technical books are not affordable, you you may like to check out the links I am putting in to free tutorial information on the net. These are generally web sites, but one is the excellent book by Bruce Eckel "Thinking in Java". This is freely downloadable and is an excellent general Java book. See http://www.bruceeckel.com for the links. If you have an unreliable internet link

and find that large downloads like the JDK tend to break half way through then you may benefit from a program like GoZilla, from http://www.gizmo.net/gozilla. You can find the Java Glossary and a huge amount of excellent links organised by the unique Roedy Green at http://mindprod.com/jgloss.html

For the final arbiter on the Java language check out the Java Language Specification, sometimes called the JLS. This can answer the really tricky questions and the sneaky ones like "is null a keyword".You can find the JLS at

http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

Subjects not covered by the exam

As you will see from reading the objectives the topics of the exam are limited to the basics of the Java language. In case you are concerned to find what is not on the exam, take a look at the document at.

http://www.software.u-net.com/javaexam/NotCovered.htm

The licence for this tutorial

For information on other Certification books and resources see my Java Certification FAQ at

http://www.jchq.net/faq/jcertfaq.htm

For a professional Exam simulator with plenty of study material, take a look at

JCertify

---

1 Section Title: Declarations and Access Control

1. Write code that declares constructs and initializes arrays of any base type using any of the permitted forms both for declaration and for initialization.
2. Declare classes inner classes methods instance variables static variables and automatic (method local) variables making appropriate use of all permitted modifiers (such as public final static abstract and so forth). State the significance of each of these modifiers both singly and in combination and state the effect of package relationships on declared items qualified by these modifiers.

3. For a given class determine if a default constructor will be created and if so state the prototype of that constructor .

4. State the legal return types for any method given the declarations of all related methods in this or parent classes.

2 Section Title: Flow Control and Exception Handling

1. Write code using if and switch statements and identify legal argument types for these statements.

2. Write code using all forms of loops including labeled and unlabeled use of break and continue and state the values taken by loop counter variables during and after loop execution.

3. Write code that makes proper use of exceptions and exception handling clauses (try catch finally) and declare methods and overriding methods that throw exceptions.

3 Section Title: Garbage Collection

1. State the behavior that is guaranteed by the garbage collection system and write code that explicitly makes objects eligible for collection.

4 Section Title: Language Fundamentals

1. Identify correctly constructed package declarations import statements, class declarations (of all forms including inner classes) interface declarations and implementations (for java.lang.Runnable or other interface described in the test) method declarations (including the main method that is used to start execution of a class) variable declarations and identifiers.

2. State the correspondence between index values in the argument array passed to a main method and command line arguments.

3. Identify all Java programming language keywords .

4. State the effect of using a variable or array element of any kind when no explicit assignment has been made to it .

5. State the range of all primitive data types and declare literal values for String and all primitive types using all permitted formats bases and representations.

5 Section Title: Operators and Assignments

1. Determine the result of applying any operator including assignment operators and instanceof to operands of any type class scope or accessibility or any combination of these.
2. Determine the result of applying the boolean equals(Object) method to objects of any combination of the classes java.lang.String java.lang.Boolean and java.lang.Object.
3. In an expression involving the operators & | && || and variables of known values state which operands are evaluated and the value of the expression.
4. Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.

6 Section Title: Overloading, Overriding, Runtime Type and Object Orientation

1. State the benefits of encapsulation in object oriented design and write code that implements tightly encapsulated classes and the relationships "is a" and "has a".
2. Write code to invoke overridden or overloaded methods and parental or overloaded constructors; and describe the effect of invoking these methods.
3. Write code to construct instances of any concrete class including normal top level classes inner classes static inner classes and anonymous inner classes.

7 Section Title: Threads

1. Write code to define instantiate and start new threads using both java.lang.Thread and java.lang.Runnable
2. Recognize conditions that might prevent a thread from executing .
3. Write code using synchronized wait notify and notifyAll to protect against concurrent access problems and to communicate between threads. Define the interaction between threads and between threads and object locks when executing synchronized wait notify or notifyAll.

8 **Section Title: The java.awt package**

1. Write code using component container and layout manager classes of the java.awt package to present a GUI with specified appearance and resize the behavior and distinguish the responsibilities of layout managers from those of containers.
2. Write code to implement listener classes and methods and in listener methods extract information from the event to determine the affected component mouse position nature and time of the event. State the event classname for any specified event listener interface in the java.awt.event package.

9 **Section Title: The java.lang package**

1. Write code using the following methods of the java.lang.Math class: abs ceil floor max min random round sin cos tan sqrt.
2. Describe the significance of the immutability of String objects.

10 **Section Title: The java.util package**

1. Make appropriate selection of collection classes/interfaces to suit specified behavior requirements.

11 **Section Title: The Java.io package)**

1. Write code that uses objects of the file class to navigate a file system.
2. Write code that uses objects of the classes InputStreamReader and OutputStreamWriter to translate between Unicode and either platform default or ISO 8859-1 character encodings.
3. Distinguish between conditions under which platform default encoding conversion should be used and conditions under which a specific conversion should be used
4. Select valid constructor arguments for subclasses from a list of classes in the java.io.package.
5. Write appropriate code to read, write and update files using FileInputStream, FileOutputStream, and RandomAccessFile objects.

Last Modified 5 May 2001

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

Recommended book on this topic

Just Java and Beyond by Peter van der Linden

If you are new to Java this is an excellent place to start. If you read all the way through this tutorial you will see I quote Peters writing in several places. He is more than quoteable though, he manages to explain the language to beginners without over simplifying topics. If you are already familiar with Java a Certification specific book might be more appropriate.

Buy from Amazon.com or from Amazon.co.uk

---

1) Declarations and Access Control

Objective 1)

Write code that declares, constructs and initializes arrays of any base type using any of the permitted forms, both for declaration and for initialization.

Arrays

Arrays in Java are similar in syntax to arrays in other languages such as C/C++ and Visual Basic. However, Java removes the feature of C/C++ whereby you can bypass the [] style accessing of elements and get under the hood using pointers. This capability in C/C++ , although powerful, makes it easy to write buggy software. Because Java does not support this direct manipulation of pointers, this source of bugs is removed.

An array is a type of object that contains values called elements. This gives you a convenient bag or holder for a group of values that can be moved around a program, and allows you to access and change values as you need them. Unlike variables which are accessed by a name, elements are accessed by numbers starting from zero. Because of this you can "walk" through an array, accessing each element in turn.

Every element of an array must be of the same type The type of the elements of an array is decided when the array is declared. If you need a way of storing a group of elements of different types, you can use the collection classes which are a new feature in the Java2 exam, and are discussed in section 10.

Declaration without allocation

The declaration of an array does not allocate any storage, it just announces the intention of creating an array. A significant difference to the way C/C++ declares an array is that no size is specified with the identifier. Thus the following will cause a compile time error

```
int num[5];
```

The size of an array is given when it is actually created with the *new* operator thus

```
int num[];
num = new int[5];
```

Simultaneous declaration and creation

This can be compressed into one line as

```
int num[] = new int[5];
```

Also the square brackets can be placed either after the data type or after the name of the array. Thus both of the following are legal

```
int[] num;

int num[];
```

You can read these as either

- An integer array named num
- An integer type in an array called num.

You might also regard it as enough choice to cause confusion

Java vs C/C++ arrays

Java arrays know how big they are, and the language provides protection from accidentally walking off the end of them.

This is particularly handy if you are from a Visual Basic background and are not used to constantly counting from 0. It also helps to avoid one of the more insidious bugs in C/C++ programs where you walk off the end of an array and are pointing to some arbitrary area of memory.

Thus the following will cause a run time error,
ArrayIndexOutOfBoundsException

```
int[] num= new int[5];
for(int i =0; i<6; i++){
        num[i]=i*2;
        }
```

The standard idiom for walking through a Java array is to use the *length* member of the array thus

```
int[] num= new int[5];
for(int i =0; i<num.length; i++){
        num[i]=i*2;
```

```
}
```

Arrays know their own size

Just in case you skipped the C/C++ comparison, arrays in Java always know how big they are, and this is represented in the *length* field. Thus you can dynamically populate an array with the following code

```
int myarray[]=new int[10];
for(int j=0; j<myarray.length;j++){
myarray[j]=j;
}
```

Note that arrays have a *length* field not a *length()* method. When you start to use *Strings* you will use the string, *length* method, as in
s.length();

With an array the length is a field (or property) not a method.

Java vs Visual Basic Arrays

Arrays in Java always start from zero. Visual Basic arrays may start from 1 if the *Option base* statement is used. There is no Java equivalent of the Visual Basic *redim preserve* command whereby you change the size of an array without deleting the contents. You can of course create a new array with a new size and copy the current elements to that array.

An array declaration can have multiple sets of square brackets. Java does not formally support multi dimensional arrays, however it does support arrays of arrays, also known as nested arrays.

The important difference between multi dimensional arrays, as in C/C++ and nested arrays, is that each array does not have to be of the same length. If you think of an array as a matrix, the matrix does not have to be a rectangle. According to the Java Language Specification

(http://java.sun.com/docs/books/jls/html/10.doc.html#27805)

"The number of bracket pairs indicates the depth of array nesting."

In other languages this would correspond to the dimensions of an array. Thus you could set up the squares on a map with an array of 2 dimensions thus

```
int i[][];
```

The first dimension could be X and second Y coordinates.

Combined declaration and initialization

Instead of looping through an array to perform initialisation, an array can be created and initialised all in one statement. This is particularly suitable for small arrays. The following will create an array of integers and populate it with the numbers 0 through 4

```
int k[]=new int[] {0,1,2,3,4};
```

Note that at no point do you need to specify the number of elements in the array. You might get exam questions that ask if the following is correct.

```
int k=new int[5] {0,1,2,3,4} //Wrong, will not compile!
```

You can populate and create arrays simultaneously with any data type, thus you can create an array of strings thus

```
String s[]=new String[] {"Zero","One","Two","Three","Four"};
```

The elements of an array can be addressed just as you would in C/C++ thus

```
String s[]=new String[] {"Zero","One","Two","Three","Four"};
System.out.println(s[0]);
```

This will output the string Zero.

Default values of arrays

The elements of arrays are always set to default values wherever the array is created

Unlike other variables that act differently between class level creation and local method level creation, Java arrays are always set to default values. Thus an array of integers will all be set to zero, an array of boolean values will always be set to false.

Exercise 1)

Create a class with a method that simultaneously creates and initialises a String array. Initialise the array with four names, then print out the first name in the array.

Exercise 2)

Create a class that creates a 5 element array of Strings called Fruit at class level but do not initialise with any values. Create a method called amethod. In amethod initialise the first four elements with the names of fruit. Create another method called modify and change the contents of the first element of the Fruit array to contain the string "bicycle". Within the modify method create a for loop that prints out every element of the Fruit array.

Suggested solution to exercise 1)

```
public class Bevere{

public static void main(String argv[]){
        Bevere b = new Bevere();
        b.Claines();
        }

        public void Claines(){
        String[] names= new String[]{"Peter","John","Balhar","Raj"};
        System.out.println(names[0]);
        }
}
```

Note: The syntax for simultaneous creation and initialisation is not obvious and is worth practising. I asked for the first name to be printed out to ensure you did not request names[1].

Suggested solution to exercise 2)

```
public class Barbourne{
String Fruit[]= new String[5];
public static void main(String argv[]){
        Barbourne b = new Barbourne();
        b.amethod();
        b.modify();
        }
        public void amethod(){
                Fruit[0]="Apple";
                Fruit[1]="Orange";
                Fruit[2]="Bannana";
                Fruit[3]="Mango";

        }

        public void modify(){
                Fruit[0]="Bicycle";
                for(int i=0; i< Fruit.length; i++){
                        System.out.println(Fruit[i]);
                }

        }
}
```

Note: that when the loop executes the output for the final elements is null

Quiz

Questions

Question 1)

How can you re-size an array in a single statement whilst keeping the original contents?

Question 2)

You want to find out the value of the last element of an array. You write the following code. What will happen when you compile and run it?

```
public class MyAr{
```

```
public static void main(String argv[]){
        int[] i = new int[5];
        System.out.println(i[5]);
        }
}
```

Question 3)

You want to loop through an array and stop when you come to the last element. Being a good java programmer, and forgetting everything you ever knew about C/C++ you know that arrays contain information about their size. Which of the following can you use?

```
1)myarray.length();
2)myarray.length;
3)myarray.size
4)myarray.size();
```

Question 4)

Your boss is so pleased that you have written HelloWorld he she has given you a raise. She now puts you on an assignment to create a game of TicTacToe (or noughts and crosses as it was when I were a wee boy). You decide you need a multi dimensioned array to do this. Which of the following will do the job?

```
1) int i=new int[3][3];
2) int[] i=new int[3][3];
3) int[][] i=new int[3][3];
4) int i[3][3]=new int[][];
```

Question 5)

You want to find a more elegant way to populate your array than looping through with a *for* statement. Which of the following will do this?

```
   1)
myArray{

   [1]="One";

   [2]="Two";

   [3]="Three";

   end with



   2)String s[5]=new String[] {"Zero","One","Two","Three","Four"};
   3)String s[]=new String[] {"Zero","One","Two","Three","Four"};
   4)String s[]=new String[]={"Zero","One","Two","Three","Four"};
```

Answers

Answer 1)

You cannot "resize" and array. You need to create a new temporary array of a different size and populate it with the contents of the original. Java provides resizable containers with classes such as Vector or one of the members of the collection classes.

Answer 2)

You will get a runtime error as you attempt to walk off the end of the array. Because arrays are indexed from 0 the final element will be i[4], not i[5]

Answer 3)

2) myarray.length;

Answer 4)

```
3) int[][] i=new int[3][3];
```

Answer 5)

```
3)String s[]=new String[] {"Zero","One","Two","Three","Four"};
```

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/java/data/arrays.html
Richard Baldwin covers this topic at
http://www.Geocities.com/Athens/Acropolis/3797/Java028.htm
Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec1.html#obj1
Bruce Eckel Thinking In Java
http://codeguru.earthweb.com/java/tij/tij0053.shtml
http://codeguru.earthweb.com/java/tij/tij0087.shtml

Last updated
10 July 2000

most recent copy at http://www.jchq.net

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

1) Declarations and Access Control

Objective 2

Declare classes, inner classes, methods, instance variables static, variables and automatic (method local) variables, making appropriate use of all permitted modifiers (such as public final static abstract and so forth). State the significance of each of these modifiers both singly and in combination and state the effect of package relationships on declared items qualified by these modifiers.

Comment on the objective

I find it a little disturbing that the objective uses the words *"and so forth"*.
I suspect this means you should also be aware of

- native
- transient
- synchronized
- volatile

Comparing C++/VB classes with Java

Because Java was designed to be easy for C++ programmers to learn there are many similarities between the way the two languages deal with classes. Both C++ and Java have inheritance, polymorphism, and data hiding using visibility modifiers. Some of the ways in which they differ are to do with making Java an easier language to learn and use.

The C++ language implements multiple inheritance and thus a class can have more than one parent (or base) class. Java allows only single inheritance and thus can only ever have a single parent. To overcome this limitation Java has a feature called interfaces. The language designers decided that interfaces would give some of the benefits of multiple inheritance without the drawbacks. All Java classes are descendants of the great ancestor class called *Object*.

Objects in Visual Basic are somewhat of a bolt on afterthought to the language. Visual Basic is sometimes called an Object Based language rather than Object Oriented. It is almost as if the language designers decided that classes are cool and with VB version 4 decided that they would create a new type of module, call it a class and use the dot notation to make it more like C++. The crucial element missing from the VB concept of class is that of inheritance. With VB5 Microsoft delivered the concept of interfaces which acts similarly to the Java concept of an interface. Some of the main similarities between VB classes and Java classes is the use of references and the keyword *new* word.

The role of classes in Java

Classes are the heart of Java, all Java code occurs within a class. There is no concept of free standing code and even the most simple HelloWorld application involves the creation of a class. To indicate that a class is a descendent of another class the *extends* keyword is used. If the *extends* keyword is not used the class will be a descended of the base class Object, which gives it some basic functionality including the ability to print out its name and some of the capability required in threads.

The simplest of class

The minimum requirements to define a class are the keyword *class*, the class name and the opening and closing braces. Thus

```
class classname {}
```

is a syntactically correct, if not particularly useful class (surprisingly I have found myself defining classes like this, when creating examples to illustrate inheritance).

Normally a class will also include an access specifier before the keyword class and of course, a body between the braces. Thus this is a more sensible template for a class.

```
public class classname{
//Class body goes here
}
```

Creating a simple *HelloWorld* class

Here is a simple HelloWorld program that will output the string "hello world" to the console.

```
public class HelloWorld{
public static void main(String argv[]){
        System.out.println("Hello world");
        }

}//End class definition
```

The keyword *public* is a visibility modifier that indicates this class should be visible to any other class. Only one outer class per file can be declared public. Inner classes will be covered elsewhere. If you declare more than one class in a file to be public, a compile time error will occur. Note that Java is case sensitive in every respect. The file that contains this class must be called HelloWorld.Java. Of course this is somewhat of an anomaly on Microsoft platforms that preserve, yet ignore the case of letters in a file name.

The keyword *class* indicates that a class is about to be defined and *HelloWorld* is the name of that class. The curly braces indicate the start of the class. Note that the closing brace that ends the class definition does not involve any closing semi colon. The comment

```
//End class definition
```

uses the style of single line comments that is available in C/C++. Java also understands the multi-line /* */ form of comments.

The magic of the *main* name

Giving a method the following signature has a certain significance (or magic) as it indicates to Java that this is where the program should begin its run, (similar to *main* in the C language).

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

This line indicates that a method called *main* is being defined that takes arguments (or parameters) of an array of Strings. This method is public, i.e. visible from anywhere that can see this class. The *static* keyword indicates that this method can be run without creating an instance of the class. If that means nothing to you, don't worry about it for the moment as *static* methods will be covered at length elsewhere. The keyword *void* indicates the data type returned from this method when it is called. The use of *void* indicates that no value will be returned.

The parameters of the *main* method

```
String argv[]
```

Indicate that the method takes an array of type *String*. The square brackets indicate an array. Note that the data type *String* starts with an upper case S. This is important as Java is thoroughly case sensitive. Without this exact signature the Java Virtual Machine will not recognise the method as the place to start execution of the program.

Creating an instance of a class

The *HelloWorld* application as described above is handy to illustrate the most basic of applications that you can create, but it misses out on one of the most crucial elements of using classes, the use of the key word

new

Which indicates the creation of a new instance of a class. In the *HelloWorld* application this was not necessary as the only method that was called was *System.out.println*, which is a static method and does

not require the creation of a class using the new keyword. Static methods can only access static variables, of which only one instance can exist per class. The *HelloWorld* application can be slightly modified to illustrate the creation of a new instance of a class.

```
public class HelloWorld2{
        public static void main(String argv[]){
                HelloWorld2 hw = new HelloWorld2();
                hw.amethod();
        }

        public void amethod(){
                System.out.println("Hello world");
        }

}
```

This code creates a new instance of itself with the line

```
                HelloWorld2 hw = new HelloWorld2();
```

This syntax of creating a new instance of a class is basic to the use of classes. Note how the name of the class appears twice. The first time indicates the data type of the reference to the class. This need not be the same as the actual type of the class as indicated after the use of the *new* keyword. The name of this instance of the class is *hw*. This is simply a name chosen for a variable. There is a naming convention that an instance of a class starts with a lower case letter, whereas the definition of a class starts with an upper case letter.

The empty parenthesis for the name of the class *HelloWorld()* indicate that the class is being created without any parameters to its constructor. If you were creating an instance of a class that was initialized with a value or a string such as the label of a button the parenthesis would contain one or more initializing values.

Creating Methods

As illustrated in the last example *HelloWorld2*, a method in Java is similar to a function in C/C++ and a function or sub in Visual Basic. The method called *amethod* in that example is the method called *amethod* in this example is declared as

public

To indicate it can be accessed from anywhere. It has a return type of

void

indicating no value will be returned. And it has empty parenthesis, indicating that it takes no parameters.

The same method might have been defined in these alternative ways

```
private void amethod(String s)
```

```
private void amethod(int i, String s)

protected void amethod(int i)
```

These examples are to illustrate some other typical signatures of methods. The use of the keywords *private* and *protected* will be covered elsewhere.

The difference between Java methods and methods in a non OO language such as C is that the methods belong to a class. This means they are called using the dot notation indicating the instance of the class that the code belongs to. (Static methods are an exception to this but don't worry about that at the moment).

Thus in HelloWorld2 amethod was called thus

```
HelloWorld hw = new HelloWorld()
hw.amethod();
```

If other instances of the *HelloWorld* class had been created the method could have been called from each instance of the class. Each instance of the class would have access to its own variables. Thus the following would involve calling the amethod code from different instances of the class.

```
HelloWorld hw = new HelloWorld();
HelloWorld hw2 = new HelloWorld();
hw.amethod();
hw2.amethod();
```

The two instances of the class *hw* and *hw2* might have access to different variables.

Automatic variables

Automatic variables are method variables. They come into scope when the method code starts to execute and cease to exist once the method goes out of scope. As they are only visible within the method they are typically useful for temporary manipulation of data. If you want a value to persist between calls to a method then a variable needs to be created at class level.

An automatic variable will "shadow" a class level variable.

Thus the following code will print out 99 and not 10.

```
public class Shad{
public int iShad=10;
public static void main(String argv[]){
                Shad s = new Shad();
                s.amethod();
        }//End of main
        public void amethod(){
                int iShad=99;
                System.out.println(iShad);
```

```
        }//End of amethod
}
```

Modifiers and encapsulation

The visibility modifiers are part of the encapsulation mechanism for Java. Encapsulation allows separation of the interface from the implementation of methods.

The visibility modifiers are a key part of the encapsulation mechanism for java. Encapsulation allows separation of the interface from the implementation of methods. The benefit of this is that the details of the code inside a class can be changed without it affecting other objects that use it. This is a key concept of the Object Oriented paradaigm (had to use that word somewhere eventually).

Encapsulation generally takes form of methods to retrieve and update the values of *private* class variables. These methods are known as a *accessor* and *mutator* methods. The accessor (or get) method retrieves the value and the mutator changes (or sets) the value. The naming convention for these methods are *setFoo* to change a variable and *getFoo* to obtain the contents of a variable. An aside note: the use of *get* and *set* in the naming of these methods is more significant than just programmer convenience and is an important part of the Javabeans system. Javabeans are not covered in the programmer exam however.

Take the example where you had a variable used to store the age of a student.

You might store it simply with a public integer variable

int iAge;

later when your application is delivered you find that some of your students have a recorded age of more than 200 years and some have an age of less than zero. You are asked to put in code to check for these error conditions. So wherever your programs change the age value, you write if statements that check for the range.

```
if(iAge > 70){
        //do something
        }
if (iAge <3){
        //do something
}
```

In the process of doing this you miss some code that used the iAge variable and you get called back because you have a 19 year old student who is on your records has being 190 years old.

The Object Oriented approach to this problem using encapsulation, is to create methods that access a private field containing the age value, with names like setAge and getAge. The setAge method might take an integer paramete and update the private value for Age and the getAge method would take no parameter but return the value from the private age field.

```
public void setAge(int iStudentAge){
```

```
        iAge = iStudentAge;
}

public int getAge(){
        return iAge;
}
```

At first this seems a little pointless as the code seems to be a long way around something that could be done with simple variable manipulation. However when they come back to you with the requirement to do more and more validation on the iAge field you can do it all in these methods without affecting existing code that uses this information.

By this approach the implementation of code, (the actual lines of program code), can be changed whilst the way it looks to the outside world (the interface) remains the same.

Private

Private variables are only visible from within the same class as they are created.in. This means they are NOT visible within sub classes. This allows a variable to be insulated from being modified by any methods except those in the current class. As described in modifiers and encapsulation, this is useful in separating the interface from the implementation.

```
class Base{
private int iEnc=10;
public void setEnc(int iEncVal){
     if(iEncVal < 1000){
         iEnc=iEncVal;
          }else
          System.out.println("Enc value must be less than 1000");
         //Or Perhaps thow an exception
        }//End if
}

public class Enc{
public static void main(String argv[]){
                Base b = new Base();
                b.setEnc(1001);
        }//End of main
}
```

Public

The *public* modifier can be applied to a variable (field) or a class. It is the first modifier you are likely to come across in learning Java. If you recall the code for the *HelloWorld*.Java program the class was declared as

```
public class HelloWorld
```

This is because the Java Virtual Machine only looks in a class declared as public for the magic *main* startup method

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

A public class has global scope, and an instance can be created from anywhere within or outside of a program. Only one non inner class in any file can be defined with the *public* keyword. If you define more than one non inner class in a file with the keyword public the compiler will generate an error.

Using the public modifier with a variable makes it available from anywhere. It is used as follows,

```
public int myint =10;
```

If you want to create a variable that can be modified from anywhere you can declare it as public. You can then access it using the dot notation similar to that used when calling a method.

```
class Base {
        public int iNoEnc=77;
}
public class NoEnc{
public static void main(String argv[]){
                Base b = new Base();
                b.iNoEnc=2;
                System.out.println(b.iNoEnc);
        }//End of main
}
```

Note that this is not the generally suggested way as it allows no separation between the interface and implementation of code. If you decided to change the data type of *iNoEnc*, you would have to change the implementation of every part of the external code that modifies it.

Protected

The *protected* modifier is a slight oddity. A *protected* variable is visible within a class, and in sub classes, the same package but not elsewhere. The qualification that it is visible from the same package can give more visibility than you might suspect. Any class in the same directory is considered to be in the default package, and thus protected classes will be visible. This means that a protected variable is more visible than a variable defined with no access modifier.

A variable defined with no access modifier is said to have default visibility. Default visibility means a variable can be seen within the class, and from elsewhere within the same package, but not from sub-classes that are not in the same package.

Static

Static is not directly a visibility modifier, although in practice it does have this effect. The modifier *static* can be applied to an inner class, a method and a variable. Marking a variable as *static* indicates that only one copy will exist per class. This is in contrast with normal items where for instance with an integer

variable a copy belongs to each instance of a class. Thus in the following example of a non *static integer* three instances of the *integer* iMyVal will exist and each instance can contain a different value.

```
class MyClass{
        public int iMyVal=0;
}
public class NonStat{
public static void main(String argv[]){
        MyClass m1 = new MyClass();
        m1.iMyVal=1;
        MyClass m2 = new MyClass();
        m2.iMyVal=2;
        MyClass m3 = new MyClass();
        m3.iMyVal=99;
        //This will output 1 as each instance of the class
        //has its own copy of the value iMyVal
        System.out.println(m1.iMyVal);
        }//End of main

}
```

The following example shows what happens when you have multiple instances of a class containing a static integer.

```
class MyClass{
        public static  int iMyVal=0;

}//End of MyClass
public class Stat{
public static void main(String argv[]){
           MyClass m1 = new MyClass();
           m1.iMyVal=0;
           MyClass m2 = new MyClass();
           m2.iMyVal=1;
           MyClass m3 = new MyClass();
           m2.iMyVal=99;
           //Because iMyVal is static, there is only one
           //copy of it no matter how many instances
           //of the class are created /This code  will
           //output a value of 99
           System.out.println(m1.iMyVal);
        }//End of main

}
```

Bear in mind that you cannot access non static variables from within a static method. Thus the following will cause a compile time error

```
public class St{
int i;
public static void main(String argv[]){
        i = i + 2;//Will cause compile time error
        }
}
```

A static method cannot be overriden to be non static in a child class

A static method cannot be overriden to be non static in a child class. Also a non static (normal) method cannot be overriden to be static in a child class. There is no similar rule with reference to overloading. The following code will cause an error as it attempts to override the class amethod to be non-static.

```
class Base{
        public static void amethod(){
        }
}

public class Grimley extends Base{
        public void amethod(){}//Causes a compile time error
}
```

The IBM Jikes compiler produces the following error

```
Found 1 semantic error compiling "Grimley.java":

     6.         public void amethod(){}

                            <------->

*** Error: The instance method "void amethod();"

cannot override the static method "void amethod();"

declared in type "Base"
```

Native

The native modifier is used only for methods and indicates that the body of the code is written in a language other than Java such as C and C++. Native methods are often written for platform specific purposes such as accessing some item of hardware that the Java Virtual Machine is not aware of. Another reason is where greater performance is required.

A native method ends with a semicolon rather than a code block. Thus the following would call an

external routine, written perhaps in C++

```
public native fastcalc();
```

Abstract

It is easy to overlook the *abstract* modifier and miss out on some of its implications. It is the sort of modifier that the examiners like to ask tricky questions about.

The *abstract* modifier can be applied to classes and methods. When applied to a method it indicates that it will have no body (ie no curly brace part) and the code can only be run when implemented in a child class. However there are some restrictions on when and where you can have *abstract* methods and rules on classes that contain them. A class must be declared as abstract if it has one or more abstract methods or if it inherits abstract methods for which it does not provide an implementation. The other circumstance when a class must be declared abstract is if it implements an interface but does not provide implementations for every method of the interface. This is a fairly unusual circumstance however.

If a class has any abstract methods it must be declared abstract itself.

Do not be distracted into thinking that an *abstract* class cannot have non *abstract* methods. Any class that descends from an *abstract* class must implement the *abstract* methods of the base class or declare them as *abstract* itself. These rules tend to beg the question why would you want to create *abstract* methods?

Abstract methods are mainly of benefit to class designers. They offer a class designer a way to create a prototype for methods that ought to be implemented, but the actual implementation is left to people who use the classes later on. Here is an example of an *abstract* a class with an abstract method. Again note that the class itself is declared *abstract*, otherwise a compile time error would have occurred.

The following class is abstract and will compile correctly and print out the string

```
public abstract class abstr{
public static void main(String argv[]){
        System.out.println("hello in the abstract");
        }
        public abstract int amethod();
}
```

Final

The final modifier can be applied to classes, methods and variables. It has similar meanings related to inheritance that make it fairly easy to remember. A final class may never be subclassed. Another way to think of this is that a *final* class cannot be a parent class. Any methods in a *final* class are automatically

final. This can be useful if you do not want other programmers to "mess with your code". Another benefit is that of efficiency as the compiler has less work to do with a final method. This is covered well in Volume 1 of Core Java.

The *final* modifier indicates that a method cannot be overriden. Thus if you create a method in a sub class with exactly the same signature you will get a compile time error.

The following code illustrates the use of the *final* modifier with a class. This code will print out the string "amethod"

```
final class Base{

public void amethod(){
        System.out.println("amethod");
        }
}

public class Fin{
public static void main(String argv[]){
        Base b = new Base();
        b.amethod();
        }
}
```

A final variable cannot have it's value changed and must be set at creation time. This is similar to the idea of a constant in other languages.

Synchronized

The *synchronized* keyword is used to prevent more than one thread from accessing a block of code at a time. See section 7 on threads to understand more on how this works.

Transient

The *transient* keyword is one of the less frequently used modifiers. It indicates that a variable should not be written out when a class is serialized.

Volatile

You probably will not get a question on the *volatile* keyword. The worst you will get it is recognising that it actually is a Java keyword. According to Barry Boone

"it tells the compiler a variable may change asynchronously due to threads"

Accept that it is part of the language and then get on worrying about something else

Using modifiers in combination

The visibility modifiers cannot be used in combination, thus a variable cannot be both *private* and *public*, *public* and *protected* or *protected* and *private*. You can of course have combinations of the visibility modifiers and the modifiers mentioned in my *so forth* list

- native
- transient
- synchronized
- volatile

Thus you can have a public static native method.

Where modifiers can be used

Modifier	Method	Variable	class
public	yes	yes	yes
private	yes	yes	yes (nested)
protected	yes	yes	yes(nested)
abstract	yes	no	yes
final	yes	yes	yes
transient	no	yes	no
native	yes	no	no
volatile	no	yes	no

Exercise 1)

Create a file called Whitley.java. In this file define a class called Base with an abstract method called lamprey with an int return type. In this file create a class called Whitley that extends the base class. Give the Whitley class a method called lamprey and code that prints out the string "lamprey"..

Create a native method for the class called mynative. Now compile and run the code.

Exercise 2)

Create a public class called Malvern. Create a private inner class called Great that has a public void method called show. Make this method print out the string "Show". Give the class Malvern a public method called go that creates an instance of Great and calls its show method.. In the main method of Malvern create an instance of itself. Make the instance of itself call its go method. Compile and run the code.

Suggested solution to Exercise 1)

```
abstract class Base{
abstract int lamprey();
}

public class Whitley extends Base{
public static void main(String argv[]){

        }

public int lamprey(){
        System.out.println("lamprey");
        return 99;
        }
native public void mynative();
}
```

Suggested solution to Exercise 2)

```
public class Malvern{
public static void main(String argv[]){
        Malvern m = new Malvern();
        m.go();
        }
        public void go(){
        Great g = new Great();
        g.show();
        }

        private class Great{
                public void show(){
                        System.out.println("Show");
                }
        }
}
```

Quiz

Questions

Question 1)

What will happen when you attempt to compile and run this code?

```
abstract class Base{
        abstract public void myfunc();
        public void another(){
        System.out.println("Another method");
        }
}

public class Abs extends Base{
        public static void main(String argv[]){
        Abs a = new Abs();
        a.amethod();
        }
        public void myfunc(){
                System.out.println("My func");
                }

        public void amethod(){
        myfunc();
         }
}
```

1) The code will compile and run, printing out the words "My Func"
2) The compiler will complain that the Base class has non abstract methods
3) The code will compile but complain at run time that the Base class has non abstract methods
4) The compiler will complain that the method myfunc in the base class has no body, nobody at all to looove it

---

Question 2)

What will happen when you attempt to compile and run this code?

```
public class MyMain{
public static void main(String argv){
        System.out.println("Hello cruel world");
        }
}
```

1) The compiler will complain that main is a reserved word and cannot be used for a class
2) The code will compile and when run will print out "Hello cruel world"
3) The code will compile but will complain at run time that no constructor is defined
4) The code will compile but will complain at run time that main is not correctly defined

---

Question 3)

Which of the following are Java modifiers?

1) public
2) private
3) friendly
4) transient

Question 4)

What will happen when you attempt to compile and run this code?

```
class Base{
        abstract public void myfunc();
        public void another(){
        System.out.println("Another method");
        }
}

public class Abs extends Base{
        public static void main(String argv[]){
        Abs a = new Abs();
        a.amethod();
        }
        public void myfunc(){
                System.out.println("My func");
                }
        public void amethod(){
        myfunc();
        }

}
```

1) The code will compile and run, printing out the words "My Func"
2) The compiler will complain that the Base class is not declared as abstract.
3) The code will compile but complain at run time that the Base class has non abstract methods
4) The compiler will complain that the method myfunc in the base class has no body, nobody at all to

looove it

Question 5)

Why might you define a method as native?

1) To get to access hardware that Java does not know about
2) To define a new data type such as an unsigned integer
3) To write optimised code for performance in a language such as C/C++
4) To overcome the limitation of the private scope of a method

---

Question 6)

What will happen when you attempt to compile and run this code?

```
class Base{
public final void amethod(){
        System.out.println("amethod");
        }
}

public class Fin extends Base{
public static void main(String argv[]){
        Base b = new Base();
        b.amethod();
        }
}
```

1) Compile time errror indicating that a class with any final methods must be declared final itself
2) Compile time error indicating that you cannot inherit from a class with final methods
3) Run time error indicating that Base is not defined as final
4) Success in compilation and output of "amethod" at run time.

Question 7)

What will happen when you attempt to compile and run this code?

```
public class Mod{
public static void main(String argv[]){
}
        public static native void amethod();
```

```
}
```

1) Error at compilation: native method cannot be static
2) Error at compilation native method must return value
3) Compilation but error at run time unless you have made code containing native amethod available
4) Compilation and execution without error

Question 8)

What will happen when you attempt to compile and run this code?

```
private class Base{}
public class Vis{
transient int  iVal;
public static void main(String elephant[]){
        }
}
```

1) Compile time error: Base cannot be private
2) Compile time error indicating that an integer cannot be transient
3) Compile time error transient not a data type
4) Compile time error malformed main method

Question 9)

What happens when you attempt to compile and run these two files in the same directory?

```
//File P1.java
package MyPackage;
class P1{
void afancymethod(){
        System.out.println("What a fancy method");
        }
}
//File P2.java
public class P2 extends P1{
afancymethod();
}
```

1) Both compile and P2 outputs "What a fancy method" when run
2) Neither will compile

3) Both compile but P2 has an error at run time
4) P1 compiles cleanly but P2 has an error at compile time

Question 10)

Which of the following are legal declarations?

1) public protected amethod(int i)
2) public void amethod(int i)
3) public void amethod(void)
4) void public amethod(int i)

Answers

Answer 1)

1) The code will compile and run, printing out the words "My Func"

An abstract class can have non abstract methods, but any class that extends it must implement all of the abstract methods.

Answer 2)

4) The code will compile but will complain at run time that main is not correctly defined

The signature of main has a parameter of String rather than string array

Answer 3)

1) public
2) private
4) transient

Although some texts use the word friendly when referring to visibility it is not a Java reserved word. Note that the exam will almost certainly contain questions that ask you to identify Java keywords from a list

Answer 4)

2) The compiler will complain that the Base class is not declared as abstract.

The actual error message using my JDK 1.1 compiler was

```
Abs.java:1: class Base must be declared abstract.
```

```
It does not define void myfunc

() from class Base.

class Base{

^

1 error
```

Answer 5)

1) To get to access hardware that Java does not know about
3) To write optimised code for performance in a language such as C/C++

Although the creation of "Pure Java" code is highly desirable, particularly to allow for platform independence, it should not be a religion, and there are times when native code is required.

Answer 6)

4) Success in compilation and output of "amethod" at run time.

This code calls the version of amethod in the Base class. If you were to attempt to implement an overriden version of amethod in Fin you would get a compile time error.

Answer 7)

4) Compilation and execution without error

There is no call to the native method and so no error occurs at run time

Answer 8)

1) Compile time error: Base cannot be private

A top level class such as base cannot be declared to be private.

Answer 9)

4) P1 compiles cleanly but P2 has an error at compile time

Even though P2 is in the same directory as P1, because P1 was declared with the package statement it is not visible from P2

Answer 10)

2) public void amethod(int i)

If you thought that option 3 was legal with a parameter argument of *void* you may have to empty some of the C/C++ out of your head.
Option 4) is not legal because method return type must come immediatly before the method name.

Other sources on this topic

This topic is covered in the Sun Tutorial at
Class modifiers
http://java.sun.com/docs/books/tutorial/reflect/class/getModifiers.html
Controlling access to members of a class
http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html

Richard Baldwin Covers this topic at
http://www.Geocities.com/Athens/Acropolis/3797/Java040.htm

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec1.html#obj2

Bruce Eckel Thinking in Java
http://codeguru.earthweb.com/java/tij/tij0056.shtml

Last updated
30 October 2001

most recent copy at http://www.jchq.net

End of document

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

1) Declarations and Access Control

Objective 3

For a given class, determine if a default constructor will be created and if so state the prototype of that constructor.

Note on this objective

This is a neat small objective that concentrates on an easily overlooked aspect of the Java language

What is a constructor?

You need to understand the concept of a constructor to understand this objective. Briefly, it is special type of method that runs automatically when an class is instantiated. Constructors are often used to initialise values in the class. Constructors have the same name as the class and no return value. You may get questions in the exam that have methods with the same name as the class but a return type, such as int or string. Be careful and ensure that any method you assume is a constructor does not have a return type.

Here is an example of a class with a constructor that prints out the string "Greetings from Crowle" when an instance of the class is created.

```
public class Crowle{
        public static void main(String argv[]){
        Crowle c = new Crowle();
        }
        Crowle(){
        System.out.println("Greetings from Crowle");
```

```
        }
}
```

When does java supply the default constructor?

If you do not specifically define any constructors, the compiler inserts an invisible zero parameter constructor "behind the scenes". Often this is of only theoretical importance, but the important qualification is that you only get a default zero parameter constructor if you do not create any of your own.

If you create constructors of your own,
Java does not supply the default zero parameter constructor

As soon as you create any constructors of your own you loose the default no parameter constructor. If you then try to create an instance of the class without passing any parameters (i.e. invoking the class with a zero parameter constructor), you will get an error. Thus as soon as you create any constructors for a class you need to create a zero parameter constructor. This is one of the reasons that code generators like Borland/Inprise JBuilder create a zero parameter constructor when they generate a class skeleton.

The following example illustrates code that will not compile. When the compiler checks to create the instance of the Base class called *c* it inserts a call to the zero parameter constructor. Because *Base* has an *integer* constructor the zero parameter constructor is not available and a compile time error occurs. This can be fixed by creating a "do nothing" zero parameter constructor in the class *Base*.

```
//Warning: will not compile.
class Base{
Base(int i){
        System.out.println("single int constructor");
        }
}

public class Cons {
        public static void main(String argv[]){
        Base c = new Base();
        }
}
//This will compile
class Base{
Base(int i){
        System.out.println("single int constructor");
        }
Base(){}
}
```

```
public class Cons {
        public static void main(String argv[]){
        Base c = new Base();
        }
}
```

The prototype of the default constructor

The objective asks you to be aware of the prototype of the default constructor. Naturally it must have no parameters. The most obvious default is to have no scope specifier, but you could define the constructor as public or protected.

Constructors cannot be native, abstract, static, synchronized or final.

That piece of information was derived directly from a compiler error message. It seems that the quality of the error messages is improving with the new releases of Java. I have heard that the new IBM Java compilers have good error reporting. You might be well advised to have more than one version of the Java compiler available to check your code and hunt down errors.

Exercises

Exercise 1

Create a class called Salwarpe with a method called hello that prints out the "Hello". In the main method of the class create an instance of itself called s1 and call the hello method from that instance. Compile and run the program so you can see the output.

Exercise 2

Still using the Salwarpe.java file comment out the line that creates the s1 instance and calls its hello method. Create a public constructor for the class that takes an integer parameter. and prints out the value of the integer. Create an instance of the class called s2 passing the value of 99 to the constructor. Compile and run the program so you can see it print out the output

Exercise 3

Uncomment the line that creates the s1 instance and modify the program so it will compile and run printing out both Hello and 99.

Suggested Solution for Exercise 1

```
public class Salwarpe {
        public static void main(String argv[]){
        Salwarpe s1 = new Salwarpe();
        s1.hello();
        }
        public void hello(){
        System.out.println("Hello");
        }
}
```

Suggested Solution for Exercise 2

```
public class Salwarpe {
        public static void main(String argv[]){
        //Salwarpe s1 = new Salwarpe();
        //s1.hello();
        Salwarpe s2 = new Salwarpe(99);
        }
        public void hello(){
        System.out.println("Hello");
        }
        public Salwarpe(int i){
        System.out.println(i);
        }
}
```

Suggested Solution for Exercise 3

```
public class Salwarpe {
        public static void main(String argv[]){
        Salwarpe s1 = new Salwarpe();
        s1.hello();
        Salwarpe s2 = new Salwarpe(99);
        }
        public void hello(){
        System.out.println("Hello");
        }
        public Salwarpe(int i){
        System.out.println(i);
        }
        public Salwarpe(){}
```

```
}
```

Note how you must create a zero parameter constructor for this final exercise. Once you have created any constructors for a class, Java will not provide the "behind the scenes" zero parameter constructor that was available in exercise 1.

Quiz

Questions

Question 1)

Given the following class definition

```
class Base{
        Base(int i){}
}

class DefCon extends Base{
DefCon(int i){
     //XX
     }
}
```

Which of the following lines would be legal individually if added at the line marked //XX?

1) super();
2) this();
3) this(99);
4)super(99);

Question 2)

Given the following class

```
public class Crowle{
```

```
        public static void main(String argv[]){
        Crowle c = new Crowle();
        }
        Crowle(){
        System.out.println("Greetings from Crowle");
        }
}
```

What is the datatype returned by the constructor?

1) null
2) integer
3) String
4) no datatype is returned

Question 3)

What will happen when you attempt to compile and run the following code?

```
public class Crowle{
        public static void main(String argv[]){
        Crowle c = new Crowle();
        }
void Crowle(){
        System.out.println("Greetings from Crowle");
        }
}
```

1) Compilation and output of the string "Greetings from Crowle"
2) Compile time error, constructors may not have a return type
3) Compilation and output of string "void"
4) Compilation and no output at runtime

Question 4)

What will happen when you attempt to compile and run the following class?

```
class Base{
        Base(int i){
        System.out.println("Base");
        }



}
```

```
class Severn extends Base{
public static void main(String argv[]){
        Severn s = new Severn();
        }
        void Severn(){
        System.out.println("Severn");
        }
}
```

1) Compilation and output of the string "Severn" at runtime
2) Compile time error
3) Compilation and no output at runtime
4) Compilation and output of the string "Base"

Question 5)

Which of the following statements are true?

1) The default constructor has a return type of void
2) The default constructor takes a parameter of void
3) The default constructor takes no parameters
4) The default constructor is not created if the class has any constructors of its own.

Answers

Answer to Question 1)

4)super(99);

Because the class Base has a constructor defined the compiler will not insert the default zero argument constructor. Therefore calls to *super()* will cause an error. A call to *this()* is an attempt to call a non existant zero argument constructor in the current class. The call to *this(99)* causes a circular reference and will cause a compile time error.

Answer to Question 2)

4) no datatype is returned

It should be fairly obvious that no datatype is returned as by definition constructors do not have

datatypes.

Answer to Question 3)

4) Compilation and no output at runtime

Because the method Crowle has a return type it is not a constructor. Therefore the class will compile and at runtime the method Crowle is not called.

Answer to Question 4)

2) Compile time error

An error occurs when the class Severn attempts to call the zero parameter constructor in the class *Base*

Answer to Question 5)

3) The default constructor takes no parameters
4) The default constructor is not created if the class has any constructors of its own.

Option 1 is fairly obviously wrong as constructors never have a return type. Option 2 is very dubious as well as Java does not offer void as a type for a method or constructor.

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/java/javaOO/constructors.html
Richard Baldwin Covers this topic at
http://www.Geocities.com/Athens/Acropolis/3797/Java042.htm#default constructor
Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec1.html#obj3
Bruce Eckel Thinking In Java
http://codeguru.earthweb.com/java/tij/tij0050.shtml#Heading143

Last updated
26 Dec 1999

most recent copy at www.jchq.net

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

1)Declarations and Access Control

Objective 4

State the legal return types for any method given the declarations of all related methods in this or parent classes.

Note on this objective

This seems to be a rather obscurely phrased objective. It appears to be asking you to understand the difference between overloading and overriding.

To appreciate this objective you need a basic understanding of overloading and overriding of methods. This is covered in

Section 6: Overloading Overriding Runtime Type and Object Orientation

Methods in the same class

By related methods I assume that the objective means a method with the same name. If two or more methods in the same class have the same name, the method is said to be *overloaded*. You can have two methods in a class with the same name but they must have different parameter types and order.

It is the parameter order and types that distinguish between any two versions of overloaded method. The return type does not contribute towards distinguishing between methods.

The following code will generate an error at compile time, the compiler sees *amethod* as an attempt to

define the same method twice. It causes an error that will say something like

```
method  redefined with different return type: void amethod(int)
was int amethod(int)
```

```
class Same{
public static void main(String argv[]){
       Over o = new Over();
        int iBase=0;
        o.amethod(iBase);
        }
        //These two cause a compile time error
        public void amethod(int iOver){
                System.out.println("Over.amethod");
        }
       public int amethod(int iOver){
                System.out.println("Over int return method");
                return 0;
        }
}
```

The return data type does not contribute towards distinguishing between one method and another.

Methods in a sub class

You can overload a method in a sub class. All that it requires is that the new version has a different parameter order and type. The parameter names are not taken into account or the return type.

If you are going to override a method, ie completely replace its functionality in a sub class, the overriding version of the method must have exactly the same signature as the version in the base class it is replacing. This includes the return type. If you create a method in a sub class with the same name and signature but a different return type you will get the same error message as in the previous example. i.e.

```
method  redefined with different return type: void amethod(int)
was int amethod(int)
```

The compiler sees it as a faulty attempt to overload the method rather than override it.

Quiz

Questions

Question 1)

Given the following class definition

```
public class Upton{
public static void main(String argv[]){
        }
        public void amethod(int i){}
        //Here
}
```

Which of the following would be legal to place after the comment //Here ?
1) public int amethod(int z){}
2) public int amethod(int i,int j){return 99;}
3) protected void amethod(long l){ }

4) private void anothermethod(){}

Question 2)

Given the following class definition

```
class Base{
        public void amethod(){
        System.out.println("Base");
        }
}
public class Hay extends Base{
public static void main(String argv[]){
        Hay h = new Hay();
        h.amethod();
        }
}
```

Which of the following methods in class Hay would compile and cause the program to print

out the string "Hay"

1) public int amethod(){ System.out.println("Hay");}
2) public void amethod(long l){ System.out.println("Hay");}
3) public void amethod(){ System.out.println("Hay");}
4) public void amethod(void){ System.out.println("Hay");}

Answers

Answer to Question 1)

2) public int amethod(int i, int j) {return 99;}
3) protected void amethod (long l){}
4) private void anothermethod(){}

Option 1 will not compile on two counts. One is the obvious one that it claims to return an integer. The other is that it is effectivly an attempt to redefine a method within the same class. The change of name of the parameter from i to z has no effect and a method cannot be overriden within the same class.

Answer to Question 2)

3) public void amethod(){ System.out.println("Hay");}

Option 3 represents an overriding of the method in the base class, so any zero parameter calls will invoke this version.

Option 1 wll return an error indicating you are attempting to redefine a method with a different return type. Although option 2 will compile the call to amethod() invoke the Base class method and the string "Base" will be output.Option 4 was designed to catch out those with a head full of C/C++, there is no such thing as a void method parameter in Java.

Other sources on this subject

Jyothi Krishnan
http://www.geocities.com/SiliconValley/Network/3693/obj_sec1.html#obj4

In that link Jyothi suggests you go to objective 19 which you can find at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec6.html#obj19

Last updated
10 Nov 2000

most recent copy at http://www.jchq.net

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

2) Flow control and exception Handling

Objective 1)

Write code using *if* and *switch* statements and identify legal argument types for these statements.

If/else statements

If/else constructs in Java are just as you might expect from other languages. *switch/case* statements have a few peculiarities.

The syntax for the *if/else* statement is

```
if(boolean condition){
        //the boolean was true so do this
        }else {
        //do something else
}
```

Java does not have a "*then*" keyword like the one in Visual Basic.

The curly braces are a general indicator in Java of a compound statement that allows you to execute multiple lines of code as a result of some test. This is known as a *block* of code. The *else* portion is always optional.

One idiosyncrasy of the Java *if* statement is that it must take a boolean value. You cannot use the C/C++ convention of any non zero value to represent true and 0 for false.

Thus in Java the following will simply not compile

```
int k =-1;
        if(k){//Will not compile!
        System.out.println("do something");
}
```

because you must explicitly make the test of k return a boolean value, as in the following example

```
if(k == -1){
    System.out.println("do something"); //Compiles OK!
```

```
    }
```

As in C/C++ you can miss out the curly brackets thus

```
boolean k=true;
if(k)
System.out.println("do something");
```

This is sometimes considered bad style, because if you modify the code later to include additional statements they will be outside of the conditional block. Thus

```
 if(k)
        System.out.println("do something");
        System.out.println("also do this");
```

The second output will always execute.

Switch statements

Peter van der Lindens opinion of the switch statement is summed up when he says

"death to the switch statement"

Thus this is a subject you should pay extra attention to. The argument to a switch statement must be a *byte*, *char*, *short* or *int*. You might get an exam question that uses a *float* or *long* as the argument to a *switch* statement.. A fairly common question seems to be about the use of the *break* statement in the process of falling through a *switch* statement. Here is an example of this type of question.

```
int k=10;
switch(k){
        case 10:
                System.out.println("ten");
        case 20:
                System.out.println("twenty");
        }
```

Common sense would indicate that after executing the instructions following a *case* statement, and having come across another *case* statement the compiler would then finish falling through the *switch* statement. However, for reasons best known to the designers of the language *case* statements only stop falling through when they come across a *break* statement. As a result, in the above example both the strings ten and twenty will be sent to the output.

Another little peculiarity that can come up on questions is the placing of the default statement.

The default clause does not need to come at the end of a *case* statement

The conventional place for the default statement is at the end of of case options. Thus normally code will be written as follows

```
int k=10;
switch(k){
        case 10:
                System.out.println("ten");
```

```
                break;
        case 20:
                System.out.println("twenty");
                break;
        default:
                System.out.println("This is the default output");
}
```

This approach mirrors the way most people think. Once you have tried the other possibilities, you perform the default output. However, it is syntactically correct, if not advisable, to code a switch statement with the default at the top

```
int k=10;
    switch(k){
    default: //Put the default at the bottom, not here
        System.out.println("This is the default output");
        break;
    case 10:
        System.out.println("ten");
        break;
    case 20:
        System.out.println("twenty");
        break;
}
```

Legal arguments for *if* and *switch* statements

As mentioned previously an *if* statement can only take a *boolean* type and a *switch* can only take a *byte*, *char*, *short* or *int.*

The ternary ? operator

Some programmers claim that the ternary operator is useful. I do not consider it so. It is not specifically mentioned in the objectives so please let me know if it does come up in the exam.

Other flow control statements

Although the published objectives only mention the *if/else* and *case* statements the exam may also cover the *do/while* and the *while* loop.

Quiz

Question 1)

What will happen when you attempt to compile and run the following code?

```
public class MyIf{
boolean b;
public static void main(String argv[]){
 MyIf mi = new MyIf();
```

```
}

MyIf(){
        if(b){
            System.out.println("The value of b was true");
            }
            else{
            System.out.println("The value of b was false");
            }
    }
}
```

1) Compile time error variable b was not initialised
2) Compile time error the parameter to the *if* operator must evaluate to a *boolean*
3) Compile time error, cannot simultaneously create and assign value for boolean value
4) Compilation and run with output of false

Question 2)

What will happen when you attempt to compile and run this code?

```
public class MyIf{
public static void main(String argv[]){
     MyIf mi = new MyIf();
    }
MyIf(){
    boolean b = false;
    if(b=false){
       System.out.println("The value of b is"+b);
      }
    }
}

1) Run time error, a boolean cannot be appended using the + operator
2) Compile time error the parameter to the if operator must evaluate
  to a boolean
3) Compile time error, cannot simultaneously create and assign value for boolean
  value
4) Compilation and run with no output
```

Question 3)

What will happen when you attempt to compile and run this code?

```
public class MySwitch{
public static void main(String argv[]){
    MySwitch ms= new MySwitch();
    ms.amethod();
    }
public void amethod(){
    char k=10;
```

```
        switch(k){
        default:
           System.out.println("This is the default output");
          break;
        case 10:
            System.out.println("ten");
            break;
        case 20:
            System.out.println("twenty");
       break;
       }
    }
}
```

1) None of these options
2) Compile time errror target of switch must be an integral type
3) Compile and run with output "This is the default output"
4) Compile and run with output "ten"

Question 4)

What will happen when you attempt to compile and run the following code?

```
public class MySwitch{
public static void main(String argv[]){
    MySwitch ms= new MySwitch();
    ms.amethod();
    }
public void amethod(){
    int k=10;
       switch(k){
       default: //Put the default at the bottom, not here
           System.out.println("This is the default output");
            break;
         case 10:
            System.out.println("ten");
         case 20:
            System.out.println("twenty");
        break;
       }
    }
}
```

1) None of these options
2) Compile time errror target of switch must be an integral type
3) Compile and run with output "This is the default output"
4) Compile and run with output "ten"

Question 5)

Which of the following could be used as the parameter for a switch statement?

1) byte b=1;
2) int i=1;

3) boolean b=false;
4) char c='c';

Answers

Answer 1)

4) Compilation and run with output of false

Because the boolean b was created at the class level it did not need to be explicitly initialised and instead took the default value of a boolean which is false. An if statement must evaluate to a boolean value and thus b meets this criterion.

Answer 2)

4) Compilation and run with no output

Because b is a boolean there was no error caused by the if statement. If b was of any other data type an error would have occured as you attempted to perform an assignment instead of a comparison. The expression

```
if(b=false)
```

would normally represent a programmer error. Often the programmer would have ment to say

```
if (b==false)
```

If the type of b had been anything but boolean a compile time error would have resulted. The requirement for the if expression is that it return a boolean and because

```
(b=false )
```

does return a boolean it is acceptable (if useless).

Answer 3)

4) Compile and run with output "ten"

Answer 4)

1) None of these options

Because of the lack of a break statement after the

```
break 10;
```

statement the actual output will be

"ten" followed by "twenty"

Answer 5)

1) byte b=1;
2) int i=1;
4) char c='c';

A switch statement can take a parameter of byte, char, short or int.

Other sources on this topic

The Sun tutorial
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/while.html

Richard Baldwin Covers this topic at
http://www.Geocities.com/Athens/Acropolis/3797/Java026.htm#the if-else statement

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec2.html#obj5

Bruce Eckel, Thinking in Java
http://codeguru.earthweb.com/java/tij/tij0045.shtml

Last updated
24 Feb 2000

most recent version at www.jchq.net

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

2) Flow Control and Exception Handling

Objective 2)

Write code using all forms of loops including labeled and unlabeled use of break and continue and state the values taken by loop counter variables during and after loop execution.

The *for* statement

The most common method of looping is to use the *for* statement. Like C++ and unlike C, the variable that controls the looping can be created and initialised from within the *for* statement. Thus

```
public class MyLoop{
      public static void main(String argv[]){
      MyLoop ml = new MyLoop();
      ml.amethod();
      }
       public void amethod(){
                for(int K=0;K<5l;K++){
                     System.out.println("Outer "+K);
                     for(int L=0;L<5;L++)
                     {System.out.println("Inner "+L);}
                   }
          }
}
```

This code will loop 5 times around the inner loop for every time around the outer loop. Thus the output will read

```
Outer 0;
Inner 0
Inner 1
Inner 2
Inner 3
inner 4
Outer 1;
Inner 0
```

```
Inner 2
```

etc etc

The *for* statement is the equivalent of a *for/next* loop in Visual Basic. You may consider the syntax to be

```
for(initialization; conditional expression;increment)
```

The conditional expression must be a boolean test in a similar way to an *if* statement. In the code example above the for statement was followed by a code block marked by curly braces. In the same way that an *if* statement does not demand a block you can have a *for* statement that simply drives the following line thus

```
        for(int i=0;i<5;i++)
                System.out.println(i);
```

Note that in neither versions do you terminate the for line with a semi colon. If you do, the *for* loop will just spin around until the condition is met and then the program will continue in a "straight line". You do not have to create the initialisation variable (in this case) within the *for* loop, but if you do it means the variable will go out of scope as soon as you exit the loop. This can be considered an advantage in terms of keeping the scope of variables as small as possible.

The *while* loops and *do* loops, nothing unexpected

The *while* and *do* loops perform much as you would expect from the equivalent in other languages.

Thus a *while* will perform zero or more times according to a test and the *do* will perform one or more times. For a *while* loop the syntax is

```
while(condition){
        bodyOfLoop;
}
```

The condition is a boolean test just like with an *if* statement. Again you cannot use the C/C++ convention of 0 for *true* or any other value for *false*

So you might create a *while* loop as follows

```
while(i<4){
        i++;
        System.out.println("Loop value is :"i);
}
```

Note that if the variable *i* was 4 or more when you reached the while statement would not result in any output. By contrast a do loop will always execute once.

Thus with the following code you will always get at least one set of output whatever the value of the variable *i* on entering the loop.

```
do{
        System.out.println("value of : "+i);
        } while (i <4);
```

Many programmers try to use the *for* loop instead of *do while* loop as it can concentrate the creation initialisation, test and incrementation of a counter all on one line.

The *goto* statement, science or religion?

The designers of Java decided that they agreed with programming guru Edsger Dijkstra who wrote a famous article titled "Goto considered harmful". Because indiscriminate use of *goto* statements can result in hard to maintain spaghetti code it has fallen out of use and considered bad programming style. There are situations when it would be useful and to help in those situations Java offers the labeled and unlabeled versions of the *break* and *continue* keywords.

Break and *continue*

These statements allow you to conditionally break out of loops. They do not however, allow you to simply jump to another part of the program. The exam is likely to include questions covering this subject in the form of a set of nested loops. You have to work out what numbers will be printed out before the loops finish due to the action of the *break* statement.

Here is an example of the sort of irritating question you are likely to get in the exam

```
public class Br{
public static void main(String argv[]){
        Br b = new Br();
        b.amethod();
        }
        public void amethod(){
        for(int i=0;i <3;i ++){
        System.out.println("i"+i+"\n");
        outer://<==Point of this example
                if(i>2){
                break outer;//<==Point of this example
                }//End of if
               for(int j=0; j <4 && i<3; j++){
                       System.out.println("j"+j);
                       }//End of for
         }//End of for
     }//end of Br method
}
```

You then have to pick out which combination of letters are output by the code. By the way the code "\n" means to output a blank line.

Jump to a label

It is often desirable to jump from an inner loop to an outer loop according to some condition. You can do this with the use of the labeled *break* and *continue* statement.

A label is simply a non key word with a colon placed after it. By placing the name of the label after *break* or *continue* your code will jump to the label. This is handy for making part of a loop conditional. You could of course do this with an *if* statement but a *break* statement can be convenient. You cannot jump to another loop or method but exiting the current loop is often very useful.

The break statement abandons processing of the current loop entirely, the continue statement only abandons the currently processing time around the loop.

Take the following example

```
public class LabLoop{
                public static void main(String argv[]){
                LabLoop ml = new LabLoop();
  ml.amethod();
   }
        public void amethod(){
```

```
                outer:
                for(int i=0;i<2;i++){
                                for(int j=0;j<3;j++) {
                                if(j>1)
                                        //Try this with break instead of continue
                                        continue outer;
                                        System.out.println("i "+ i + " j "+j);
                                    }
                        }//End of outer for
                        System.out.println("Continuing");
                 }
}
```

This version gives the following output

```
i 0 j 0
i 0 j 1
i 1 j 0
i 1 j 1
```

```
Continuing
```

If you were to substitute the *continue* command with *break*, the *i* counter would stop at zero as the processing of the outer loop would be abandoned instead of simply continuing to the next increment.

Quiz

Question 1)

What will happen when you attempt to compile and run the following code in a method?

```
        for(int i=0;i<5;){
                System.out.println(i);
                i++;
                continue;
                }
```

1) Compile time error, malformed for statement
2) Compile time error continue within for loop
3) runtime error continue statement not reached
4) compile and run with output 0 to 4

Question 2)

What will happen when you attempt to compile and run the following code?

```
public class LabLoop{
                public static void main(String argv[]){
                LabLoop ml = new LabLoop();
  ml.amethod();
                mainmethod:
                System.out.println("Continuing");
                }
        public void amethod(){
                outer:
                for(int i=0;i<2;i++){
                       for(int j=0;j<3;j++){
                           if(j>1)
                              break mainmethod;
                              System.out.println("i "+ i + " j "+j);
                    }
        }//End of outer for
  }
}
```

1)

```
i 0 j 0
i 0 j 1
Continuing
```

2)

```
i 0 j 0
i 0 j 1
i 1 j 0
i 1 j 1
Continuing
```

3)

Compile time error

4)

```
i 0 j 0
i 0 j 1
i 1 j 0
i 1 j 1
i 2 j 1
Continuing
```

Question 3)

What will happen when you attempt to compile and run the following code?

```
public void amethod(){
                outer:
                for(int i=0;i<2;i++){
                        for(int j=0;j<2;j++){
```

```
                                System.out.println("i="+i + " j= "+j);
                                        if(i >0)
                                             break outer;
                                   }

                        }
                         System.out.println("Continuing with i set to ="+i);
                }
```

1) Compile time error

2)

```
i=0 j= 0
i=0 j= 1
i=1 j= 0
```

3)

```
i=0 j= 0
i=0 j= 1
i=1 j= 0
i=2 j= 0
```

4)

```
i=0 j= 0
i=0 j= 1
```

Question 4)

What will happen when you attempt to compile and run the following code?

```
        int i=0;
        while(i>0){
                System.out.println("Value of i: "+i);
        }
                do{
                        System.out.println(i);
                        } while (i <2);
                }
```

1)

```
Value of i: 0
followed by
0
1
2
```

2)

```
0
1
2
```

3)

```
Value of i: 0
Followed by continuous output of 0
```

4) Continuous output of 0

Answers

Answer 1)

4) compile and run with output 0 to 4

This is a strange but perfectly legal version of the for statement

Answer 2)

3) Compile time error

You cannot arbitrarily jump to another method, this would bring back all the evils manifest in the *goto* statement

Answer 3)

1) Compile time error

This is not really a question about *break* and *continue*. This code will not compile because the variable is no longer visible outside the *for* loop. Thus the final *System.out.println* statement will cause a compile time error.

Answer 4)

4) Continuous output of 0

There is no increment of any value and a while loop will not execute at all if its test is not true the on the first time around

---

Other sources on this subject

The Sun Tutorial

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/while.html

Jyothi Krishnan
http://www.geocities.com/SiliconValley/Network/3693/obj_sec2.html#obj6

Richard Baldwin
http://www.Geocities.com/Athens/Acropolis/3797/Java026.htm#flow of control

Bruce Eckel Thinking in Java
http://codeguru.earthweb.com/java/tij/tij0045.shtml#Heading131

Last updated
28 Dec 1999

most recent version at www.jchq.net

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

2) Flow Control and Exception Handling

Objective 3)

Write code that makes proper use of exceptions and exception handling clauses (try catch finally) and declares methods and overriding methods that throw exceptions.

An exception condition is a when a program gets into a state that is not quite normal. Exceptions trapping is sometimes referred to as error trapping. A typical example of an exception is when a program attempts to open a file that does not exsist or you try to refer to an element of an array that does not exist.

The *try* and *catch* statements are part of the exception handling built into Java. Neither C/C++ nor Visual Basic have direct equivalents to Javas built in exceptions. C++ does support exceptions but they are optional, and Visual Basic supports *On Error/Goto* error trapping, which smacks somewhat of a throwback to an earlier less flexible era of BASIC programming.

Java exceptions are a built in part of the language. For example if you are performing I/O you must put in exception handling. You can of course put in null handling that doesn't do anything. The following is a little piece of code I have used with Borland/Inprise JBuilder to temporarily halt output to the console and wait for any key to be pressed.

```
import java.io.*;
public class Try{
    public static void main(String argv[]){
        Try t = new Try();
        t.go();
    }//End of main
public void go(){
     try{
```

```
        DataInputStream dis= new DataInputStream(System.in);
        dis.readLine();
       } catch(Exception e){
      /*Not doing anything when exception occurs*/
      } //End of try
     System.out.println("Continuing");
    }//End of go
}
```

In this case nothing is done when an error occurs, but the programmer must still acknowledge that an error might occur. If you remove the *try* and *catch* clause the code will simply not compile. The compiler knows that the I/O methods can cause exceptions and demands exception handling code.

Comparing with Visual Basic and C/C++

This is a little more rigorous than Visual Basic or C/C++ which allows you to throw together "quick and dirty" programs that pretend they are in a world where errors do not occur. Remember that the original version of DOS was called QDOS for Quick and Dirty DOS by it's creator and look how long we have lived with the legacy of that bit of hackery. By the time you have gone to the trouble of putting in a *try/catch* block and blank braces you may as well put in some real error tracking. It's not exactly bondage and discipline programming, it just persuasively encourages you to "do the right thing".

The *finally* clause

The one oddity that you are likely to be questioned on in the exam, is under what circumstances the *finally* clause of a *try/catch* block is executed. The short answer to this is that the *finally* clause is almost always executed, even when you might think it would not be. Having said that, the path of execution of the *try/catch/finally* statements is something you really need to play with to convince yourself of what happens under what circumstances.

**The finally clause of a try catch block will always execute,
even if there are any return statements in the try catch part**

One of the few occasions when the finally clause will not be executed is if there is a call to `System.exit(0);`

The exam tends not to attempt to catch you out with this exception to the rule.

The exam is more likely to give examples that include return statements in order to mislead you into thinking that the code will return without running the *finally* statement. Do not be mislead, the *finally* clause will almost always run.

The *try/catch* clause must trap errors in the order their natural order of hierarchy. Thus you cannot attempt to trap the generic catch all *Exception* before you have put in a trap for the more specific *IOException*.

The following code will not compile

```
try{
    DataInputStream dis = new DataInputStream(System.in);
    dis.read();
    }catch (Exception ioe) {}
    catch (IOException e) {//Compile time error cause}
    finally{}
```

This code will give an error message at compile time that the more specific *IOException* will never be reached.

Overriding methods that throw exceptions

An overriding method in a subclass may only throw exceptions declared in the parent class or children of the exceptions declared in the parent class. This is only true for overriding methods not overloading methods. Thus if a method has exactly the same name and arguments it can only throw exceptions declared in the parent class, or exceptions that are children of exceptions in the parent declaration. It can however throw fewer or no exceptions. Thus the following example will not compile

```
import java.io.*;
class Base{
public static void amethod()throws FileNotFoundException{}
}
public class ExcepDemo extends Base{
  //Will not compile, exception not in base version of method
  public static void amethod()throws IOException{}
}
```

If it were the method in the parent class that was throwing *IOException* and the method in the child class that was throwing *FileNotFoundException* this code would compile. Again, remember that this only applies to overridden methods, there are no similar rules to overloaded methods. Also an overriden method in a sub class may throw Exceptions.

Quiz

Question 1)

What will happen when you attempt to compile and run the following code?

```
import java.io.*;
class Base{
public static void amethod()throws FileNotFoundException{}
}

public class ExcepDemo extends Base{
public static void main(String argv[]){
       ExcepDemo e = new ExcepDemo();
}
public static void amethod(){}
protected ExcepDemo(){
         try{
          DataInputStream din = new DataInputStream(System.in);
          System.out.println("Pausing");
          din.readChar();
          System.out.println("Continuing");
          this.amethod();
          }catch(IOException ioe) {}
        }
}
```

1) Compile time error caused by protected constructor
2) Compile time error caused by *amethod* not declaring Exception
3) Runtime error caused by amethod not declaring Exception
4) Compile and run with output of "Pausing" and "Continuing" after a key is hit

---

Question 2)

What will happen when you attempt to compile and run the following code?

```
import java.io.*;
class Base{
public static void amethod()throws FileNotFoundException{}
}
public class ExcepDemo extends Base{
public static void main(String argv[]){
       ExcepDemo e = new ExcepDemo();
}

public static void amethod(int i)throws IOException{}
private ExcepDemo(){
 try{
```

```
    DataInputStream din = new DataInputStream(System.in);
    System.out.println("Pausing");
    din.readChar();
    System.out.println("Continuing");
    this.amethod();
    }catch(IOException ioe) {}
  }
}
```

1) Compile error caused by private constructor
2) Compile error caused by *amethod* declaring Exception not in base version
3) Runtime error caused by *amethod* declaring Exception not in base version
4) Compile and run with output of "Pausing" and "Continuing" after a key is hit

---

Question 3)

What will happen when you attempt to compile and run this code?

```
import java.io.*;
class Base{
public static void amethod()throws FileNotFoundException{}
}
public class ExcepDemo extends Base{
public static void main(String argv[]){
       ExcepDemo e = new ExcepDemo();
}
public static void amethod(int i)throws IOException{}
private boolean ExcepDemo(){
         try{
            DataInputStream din = new DataInputStream(System.in);
            System.out.println("Pausing");
            din.readChar();
            System.out.println("Continuing");
            this.amethod();
            return true;
            }catch(IOException ioe) {}
            finally{
            System.out.println("finally");
            }
           return false;
        }
}
```

1) Compilation and run with no output.
2) Compilation and run with output of "Pausing", "Continuing" and "finally"
3) Runtime error caused by *amethod* declaring Exception not in base version

4) Compile and run with output of "Pausing" and "Continuing" after a key is hit

---

Question 4)

What will happen when you attempt to compile and run the following code?

```
import java.io.*;
class Base{
public static void amethod()throws FileNotFoundException{}
}
public class ExcepDemo extends Base{
public static void main(String argv[]){
       ExcepDemo e = new ExcepDemo();
}
public boolean amethod(int i){
         try{
            DataInputStream din = new DataInputStream(System.in);
            System.out.println("Pausing");
            din.readChar();
            System.out.println("Continuing");
            this.amethod();
            return true;
            }catch(IOException ioe) {}
            finally{
           System.out.println("Doing finally");
           }
            return false;
          }
                ExcepDemo(){
                amethod(99);
          }
}
```

1) Compile time error amethod does not throw FileNotFoundException
2) Compile, run and output of Pausing and Continuing
3) Compile, run and output of Pausing, Continuing, Doing Finally
4) Compile time error finally clause never reached

Answers

Answer to Question 1)

4) Compile and run with output of "Pausing" and "Continuing" after a key is hit

An overriden method in a sub class must not throw Exceptions not thrown in the base class. In the case of the method amethod it throws no exceptions and will thus compile without complaint. There is no reason that a constructor cannot be protected.

Answer to Question 2)

4) Compile and run with output of "Pausing" and "Continuing" after a key is hit
In this version *amethod* has been overloaded so there are no restrictions on what Exceptions may or may not be thrown.

Answer to Question 3)

1) Compilation and run with no output.

OK, I have wandered off topic here a little. Note that the constructor now has a return value. This turns it into an ordinary method and thus it does not get called when an instance of the class is created.

Answer to Question 4)

3) Compile, run and output of Pausing, Continuing, Doing Finally

The finally clause will always be run.

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html

Richard Baldwin Covers this topic at
http://www.geocities.com/Athens/Acropolis/3797/Java030.htm
and
http://www.geocities.com/Athens/Acropolis/3797/Java056.htm

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec2.html#obj7

Bruce Eckel Thinking in Java
Chapter 9)

Last updated
28 Dec 1999

most recent version at www.jchq.net

Java2 Certification Tutorial

3)Garbage Collection

Objective 1)

State the behavior that is guaranteed by the garbage collection system and write code that explicitly makes objects eligible for collection.

Why would you want to collect the garbage?

You can be a very experienced Java programmer and yet may never had to familiarise yourself with the details of garbage collection. Even the expression garbage collection is a little bizarre. In this context it means the freeing up of memory that has been allocated and used by the program. When the memory is no longer needed it can be considered to be garbage, i.e. something that is no longer needed and is simply cluttering up the living space.

One of the great touted beauties of the Java language is that you don't have to worry about garbage collection. If you are from a Visual Basic background it may seem absurd that any system would not look after this itself. In C/C++ the programmer has to keep track of the allocation and deallocation of memory by hand. As a result "memory leaks" are a big source of hard to track bugs. This is one of the reasons that with some versions of Microsoft applications such as Word or Excel, simply starting and stopping the program several times can cause problems. As the memory leaks away eventually the whole system hangs and you need to hit the big red switch. Somewhere in those hundreds of thousands of lines of C++ code, a programmer has allocated a block of memory but forgot to ensure that it gets released.

Java and garbage

Unlike C/C++ Java automatically frees up unused references. You don't have to go through the pain of searching for allocations that are never freed and you don't need to know how to *alloc* a *sizeof* a data type to ensure platform compatibility. So why would you want to know about the details of garbage collection? Two answers spring to mind, one is to pass the exam and the other is to understand what goes on in extreme circumstances.

If you write code that creates very large numbers of objects or variables it can be useful to know when references are released.
If you read the newsgroups you will see people reporting occasions of certain Java implementations exhausting memory resources and falling over. This was not in the brochure from Sun when they launched Java.

In keeping with the philosophy of automatic garbage collection, you can suggest or encourage the JVM to perform garbage collection but you can not force it.

Let me re-state that point, you cannot force garbage collection, just suggest it.

At first glance finalisation sounds a little like the destructors in C++ used to clean up resources before an object is destroyed. The difference is that Java internal resources do not need to be cleaned up during finalisation because the garbage collector looks after memory allocation. However if you have external resources such as file information, finalisation can be used to free external resources.

Garbage collection is a tricky one to write exercises with or practice with as there is no obvious way to get code to indicate when it is available for garbage collection. You cannot write a piece of code with a syntax like

```
if(EligibleForGC(Object){ //Not real code
    System.out("Ready for Garbage");
    }
```

Because of this you just have to learn the rules. To re-state.

Once a variable is no longer referenced by anything it is available for garbage collection.

You can suggest garbage collection with *System.gc()*, but this does not guarantee when it will happen

Local variables in methods go out of scope when the method exits. At this point the methods are eligible for garbage collection. Each time the method comes into scope the local variables are re-created.

Quiz

Question 1)

Which of the following is the correct syntax for suggesting that the JVM performs garbage collection?

1) System.free();
2) System.setGarbageCollection();
3) System.out.gc();
4) System.gc();

Question 2)

What code can you write to ensure that the Integer variables are garbage collected at a particular point in this code?

```
public class Rub{
        Integer  i= new Integer(1);
        Integer  j=new Integer(2);
        Integer k=new Integer(3);
public static void main(String argv[]){
            Rub r = new Rub();
        r.amethod();
    }
    public void amethod(){
        i=0;
        j=0;
        k=0;
    }
}
```

1) System.gc();
2) System.free();
3) Set the value of each int to null
4) None of the above

Answers

Answer to Question 1)

4) System.gc();

Answer to Question 2)

4) None of the above

You can only suggest garbage collection, therefore you cannot be certain that it will run at any particular point in your code. Note that only instances of classes are subject to garbage collection not primitives.

Other sources on this topic

An article from SUN

http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj/index.html

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec3.html#obj8

Last updated
13 Jan 1999

Java2 Certification Tutorial

4) Language Fundamentals

Objective 1)

Identify correctly constructed package declarations import statements class declarations (of all forms including inner classes) interface declarations and implementations (for java.lang.Runnable or other interface described in the test) method declarations (including the main method that is used to start execution of a class) variable declarations and identifiers.

Note on this objective

This is a strangely phrased objective. It seems to be asking you to understand where, how and why you can use import and package statements and where you should place the interface statement and variable statements. The 1.1 objective seemed to make more sense in that they asked you to "distinguish legal and illegal orderings" of various statements. I have a feeling that they did not re-write every question in the bank to match the new objectives so you will get similar questions for the Java2 exam.

The package statement

The name *package* implies a collection of classes, somewhat like a library. In use a package is also a little like a directory. If you place a package statement in a file it will only be visible to other classes in the same package. You can place a comment before the package statement but nothing else. You may get exam questions that place an import statement before the package statement

```
//You can place a comment before the package statement
package MyPack;
public class MyPack{}
```

The following will cause an error

```
import java.awt.*;
//Error: Placing an import statement before the package
//statement will cause a compile time error
package MyPack;
public class MyPack{}
```

If a source file has a package statement it must come before any other statement apart from comments

The *package* statement may include the dot notation to indicate a package hierarchy. Thus the following will compile without error

```
package myprogs.MyPack;
public class MyPack{}
```

Remember that if you do not place a package statement in a source file it will be considered to have a default package which corresponds to the current directory. This has implications for visibility which is covered in *Section 1.2 Declarations and access control*.

The import statement

Import statements must come after any *package* statements and before any code. Import statements cannot come within classes, after classes are declared or anywhere else.

The *import* statement allows you to use a class directly instead of fully qualifying it with the full package name. An example of this is that the classname *java.awt.Button* is normally referred to simply as *Button*, so long as you have put in the statement at the top of the file as follows

```
import java.awt.*;
```

Note that using a class statement does not result in a performance overhead or a change in the size of the *.class* output file.

Class and inner class declarations

A file can only contain one outer *public* class. If you attempt to create a file with more than one *public* class the compiler will complain with a specific error. A file can contain multiple non public classes, but bear in mind that this will produce separate *.class* output files for each class. It does not matter where in the file the public class is placed, so long as there is only one of them in the file.

Inner classes were introduced with JDK 1.1. The idea is to allow one class to be defined within another, to be defined within a method and for the creation of anonymous inner classes. This has some interesting affects, particularly on visibility.

Here is a simple example of an inner class

```
class Outer{
         class inner{}
}
```

This results in the generation of class files with the names

```
Outer.class
Outer$Inner.class
```

The definition of the inner class is only visible within the context of an existing Outer class. Thus the following will cause a compile time error

```
class Outer{
          class Inner{}
}
class Another{
public void amethod(){
        Inner i = new Inner();
        }
}
```

So far as the class *Another* is concerned, the class Inner does not exist. It can only exist in the context of an instance of the class Outer. Thus the following code works fine because there is an instance of *this* for the outer class at the time of creation of the instance of Inner

```
class Outer{
          public void mymethod(){
                  Inner i = new Inner();
          }
          public class Inner{}
}
```

But what happens if there is no existence of *this* for the class Outer. To make sense of the rather odd syntax provided for this try to think of the keyword *new* as used in the above example as belonging to the current insistence of *this.*

Thus you could change the line that creates the instance of this to read

```
Inner i = this.new Inner();
```

Thus if you need to create an instance of *Inner* from a *static* method or somewhere else where there is no *this* object you can use *new* as a method belonging to the outer class

```
class Outer{
          public class Inner{}
}
class another{
public void amethod(){
        Outer.Inner i = new Outer().new Inner();
       }
```

```
}
```

Despite my glib explanations, I find this syntax unintuitive and forget it five minutes after learning it. It is very likely that you will get a question on this in the exam, so give it extra attention.

You can gain access to an inner class by using the syntax
Outer.Inner i = new Outer().new Inner();

One of the benefits of inner classes is that an inner class generally gets access to the fields of its enclosing (or outer) class. Unlike an outer class an inner class may be *private* or *static*. The examiners seem to like to ask simple questions that boil down to "*can an inner class be static or private*". Marking an inner class static
has some interesting effects with regards to accessing the fields of the enclosing class. The effect of marking it as *static* means there is only one instance of any variables, no matter how many instances of the outer class are created. In this situation how could the static inner class know which variables to access of its non static outer class. Of course the answer is that it could not know, and thus an *static* inner class cannot access instance variables of its enclosing class.

The methods of an static inner class can of course access any static fields of its enclosing class as there will only ever be one instance of any of those fields.

Inner classes declared within methods

Inner classes can be created within methods. This is something that GUI builders like Borland JBuilder do a great deal of when creating Event handlers.

Here is an example of such automatically generated code

```
buttonControl1.addMouseListener(new java.awt.event.MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        buttonControl1_mouseClicked(e);
      }
    });
```

Note the keyword *new* just after the first parenthesis. This indicates that an anonymous inner class is being defined within the method *addMouseListener*. This class could have been defined normally with a name which might make it easier for a human to read, but as no processing is done with it anywhere else, having a name does not help much.

If you create such code by hand, it is easy to get confused over the number and level of brackets and parentheses. Note how the whole structure ends with a semi colon, as this is actually the end of a method

call.

As you might guess an anonymous class cannot have a constructor. Think about it, a constructor is a method with no return value and the same name as the class. Duh! we are talking about classes without names. An anonymous class may extend another class or implement a single interface. This peculiar limit does not seem to be tested in the exam.

Field visibility for classes defined within a method

A class defined within a method can only access fields in the enclosing method if they have been defined as final. This is because variables defined within methods normally are considered *automatic*, ie they only exist whilst the method is executing. Fields defined within a class created within a method may outlive the enclosing method.

A class defined within a method can only access final fields of the enclosing method.

Because a final variable cannot be changed the JVM can be sure that the value will stay constant even after the outer method has ceased to execute. You are very likely to get questions on this in the exam, including questions that query the status of variables passed as a parameter to the method (yes, they too must be final)

Creating an interface

Interfaces are the way Java works around the lack of multiple inheritance. Interestingly Visual Basic uses the keyword interface and uses the concept in a manner similar to Java. The interface approach is sometimes known as programming by contract. An interface is used via the keyword "implements". Thus a class can be declared as

```
class Malvern implements Hill,Well{
   public
   }
```

Question 1)

Given the following code

```
public class FinAc{
```

```
        static int l = 4;
        private int k=2;
public static void main(String argv[]){
        FinAc a = new FinAc();
        a.amethod();
        }
     public void amethod(){
        final int i = 99;
        int j = 6;
        class CInMet{
                public void mymethod(int q){
                                //Here
                     }//end of mymethod
                }//End of CInMet
                CInMet c = new CInMet();
                c.mymethod(i);
     }//End of amthod
}
```

Which of the following variables are visible on the line marked with the comment //Here?

1) l
2) k
3) i
4) j

Question 2)

Which of the following will compile correctly?

1)

```
//A Comment
import java.awt.*;
class Base{};
```

2)

```
import java.awt.*;
package Spot;
class Base();
```

3)

```
//Another comment
package myprogs.MyPack;
public class MyPack{}
```

4)

```
class Base{}
import java.awt.*;
public class Tiny{}
```

Question 3)

Which of the following statements are true?

1) An inner class may be defined as static
2) An inner class may NOT be define as private
3) An anonymous class may have only one constructor
4) An inner class may extend another class

Question 4)

From code that has no current *this* reference how can you create an instance of an inner class?

1) Outer.Inner i = new Outer().new Inner();
2) Without a this reference an inner class cannot be created
3) Outer.Inner i = Outer().new new Inner();
4) Outer i = Outer.new().Inner();

Answers

Answer 1)

1) l
2) k
3) i
A class defined within a method can only see final fields from its enclosing method. However it can see the fields in its enclosing class including private fields. The field j was not defined as final.

Answer 2)

1)

```
//A Comment
import java.awt.*;
class Base{};
```

3)

```
//Another comment
package myprogs.MyPack;
public class MyPack{}
```

Any package statement must be the first item in a file (apart from comments). An import statement must come after any package statement and before any code.

Answer 3)

1) An inner class may be defined as static
4) An inner class may extend another class

How could an anonymous class have a constructor? Inner classes may be defined as private.

Answer 4)

1) Outer.Inner i = new Outer().new Inner();

Other sources on this topic

The Sun Tutorial
http://java.sun.com/docs/books/tutorial/java/more/nested.html

Richard Baldwin
http://www.Geocities.com/Athens/7077/Java094.htm
and also
http://www.Geocities.com/Athens/7077/Java095.htm

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec4.html#obj9

A tutorial on packages
http://v2ma09.gsfc.nasa.gov/JavaPackages.html

The Java Language Specification on interfaces
http://java.sun.com/docs/books/jls/html/9.doc.html#238680

Last updated
12 November 2000

most recent version at http://www.jchq.net

Java2 Certification Tutorial

4) Language Fundamentals

Objective 2)

State the correspondence between index values in the argument array passed to a main method and command line arguments.

Note: This seems to be a tiny subject hardly worth an objective of its own.

This objective can catch out the more experienced C/C++ programmer because the first element of argv[] is the first string after the name of the program on the command line. Thus if a program was run as follows

```
java myprog myparm
```

the element argv[0] would contain "myparm". If you are from a C/C++ background you might expect it to contain "java". Java does not contain the Visual Basic equivalent of Option Base and arrays will always start from element zero.

Take the following example

```
public class MyParm{
public static void main(String argv[]){
        String s1 = argv[1];
        System.out.println(s1);
        }
}
```

I have placed argument one into a String just to highlight that argv is a *String* array. If you run this program with the command

```
java MyParm hello there
```

The output will be *there*, and not *MyParm* or *hello*

Quiz

Question 1)

Given the following main method in a class called Cycle and a command line of

```
java Cycle one two
```

what will be output?

```
public static void main(String bicycle[]){
        System.out.println(bicycle[0]);
}
```

1) None of these options
2) Cycle
3) one
4) two

Question 2)

How can you retrieve the values passed from the command line to the main method?

1) Use the *System.getParms()* method
2) Assign an element of the argument to a string
3) Assign an element of the argument to a char array
4) None of these options

Answers

Answer 1)

3) one

Answer 2)

2) Assign an element of the argument to a string

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/attributes/cmdLineArgs.html

Last updated
28 Dec 1999

Java2 Certification Tutorial

Your feedback is valued, please send comments to feedback@marcusgreen.co.uk

4) Language Fundamentals

Objective 3)

Identify all Java programming language keywords.

Note on this objective: You may like to approach this objective on the basis of learning the less frequently used key words and ensuring you do not carry over any "false friends" from other languages you may know, particularly C/C++. The exam places significant emphasis on recognising keywords

You will come to recognise most of the Java keywords through using the language, but there are rarely used exceptions, and reserved words that might come up in the exam.

Examples of the more rarely used words (certainly for a beginner anyway) are

- volatile
- transient
- native

Java Keywords

abstract	boolean	break	byte	case	catch
char	class	`const *`	continue	default	do
double	else	extends	final	finally	float
for	`goto *`	if	implements	import	instanceof
int	interface	long	native	new	null
package	private	protected	public	return	short

static	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

The words with asterisks are reserved and not currently used. Note that all of the keywords are in lowercase, thus *for* is a keyword but *FOR* is not. There is some debate as to if *null* is a keyword but I suggest that for the purposes of the exam you assume it is.

Quiz

Question 1)

Which of the following are Java key words?

1) double
2) Switch
3) then
4) instanceof

Question 2)

Which of the following are not Java keywords?

1)volatile
2)sizeOf
3)goto
4)try

Answers

Answer 1)

1) double
4) instanceof

Note the upper case S on switch means it is not a keyword and the word then is part of Visual Basic but

not Java

Answer 2)

2) sizeOf

This is a keyword in C/C++ for determining the size of a primitive for a particular platform. Because primitives have the same size on all platforms in Java this keyword is not needed.

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/jls/html/3.doc.html#229308
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html

Michael Thomas
http://www.michael-thomas.com/java/JCP_Keywords.htm

Last updated
28 Dec 1999

Java2 Certification Tutorial

4) Language Fundamentals

Objective4)

State the effect of using a variable or array element of any kind when no explicit assignment has been made to it.

Variables

You could learn to program in Java without really understanding the agenda behind this objective, but it does represent valuable real world knowledge. Essentially a class level variable will always be assigned a default value and a member variable (one contained within a method) will not be assigned any default value. If you attempt to access an unassigned variable an error will be generated. For example

```
class MyClass{
        public static void main(String argv[]){
                int p;
                int j = 10;
                j=p;
        }
}
```

This code will result in an error along the lines

```
"error variable p might not have been assigned"
```

This can be viewed as a welcome change from the tendency of C/C++ to give you enough rope by leaving an arbitrary value in p. If p had been defined at class level it would have been assigned its default value and no error would be generated.

```
class MyClass{
static int p;
        public static void main(String argv[]){
                int j = 10;
                j=p; System.out.println(j);
                }
}
```

The default value for an integer is 0, so this will print out a value of 0.

The default values for numeric types is zero, a boolean is false and an object reference is the only type that defaults to a null.

Key Concept

Before initialization arrays are always set to contain default values wherever they are created.

Arrays

Learning this part of the objective requires understanding a simple rule. The value of the elements of an array of any base type will **always** be initialised to a default value, wherever the array is defined. It does not matter if the array is defined at class or method level, the values will always be set to default values. You may get questions that ask you what value is contained in a particular element of an array. Unless it is an array of objects the answer will not be null (or if they are being particularly tricky NULL).

Quiz

Question 1)

Given the following code what will element *b[5]* contain?

```
public class MyVal{
public static void main(String argv[]){
        MyVal m = new MyVal();
        m.amethod();
}

public void amethod(){
boolean b[] = new boolean[5];
        }
}
```

1) 0
2) null
3) ""
4) none of these options

Question 2)

Given the following constructor what will element 1 of *mycon* contain?

```
MyCon(){
        int[] mycon= new int[5];
}
```

1) 0
2) null
3) ""
4) None of these options

Question 3)

What will happen when you attempt to compile and run the following code?

```
public class MyField{
int i=99;
public static void main(String argv[]){
        MyField m = new MyField();
        m.amethod();
       }
void amethod(){
        int i;
        System.out.println(i);
        }
}
```

1) The value 99 will be output
2) The value 0 will be output
3) Compile time error
4) Run time error

Question 4)

What will happen when you attempt to compile and run the following code?

```
public class MyField{
```

```
String s;
public static void main(String argv[]){
        MyField m = new MyField();
        m.amethod();
        }
void amethod(){
        System.out.println(s);
        }
}
```

1) Compile time error s has not been initialised
2) Runtime error s has not been initialised
3) Blank output
4) Output of null

Answers

Answer 1)

4) none of these options

Sneaky one here. Array element numbering starts at 0, therefore there is no element 5 for this array. If you were to attempt to perform

```
System.out.println(b[5])
```

You would get an exception.

Answer 2)

1) 0

A constructor acts no different to any other method for this purpose and an array of integers will be initialised to contain zeros wherever it is created.

Answer 3)

3) Compile time error

You will get a compile time error indicating that variable *i* may not have been initialised. The classs level variable *i* is a red herring, as it will be shadowed by the method level version. Method level variables do not get any default initialisation.

Answer 4)

4) Output of null

A variable created at class level will always be given a default value. The default value of an object reference is *null* and the *toString* method implicitly called via System.out.println will output *null*

Other sources on this topic

This topic is covered slightly in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/vars.htmll

Richard Baldwin Covers this topic at
http://www.geocities.com/Athens/Acropolis/3797/Java020.htm#variables

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec4.html#obj12

Last updated
28 Dec 1999

Java2 Certification Tutorial

4) Language Fundamentals

Objective 5)

State the range of all primitive data types and declare literal values for *String* and all primitive types using all permitted formats, bases and representations.

Note on this objective

This is one of the slightly annoying but fairly easy to cover objectives. You can write a large amount of Java without knowing the range of primitive types but it should not take long to memorise these details. Beware of the requirement to be able to use all formats, don't overlook the octal form

The size of integral primitives

When this objective asks for the range of primitive data types I assume it is only required as representing the number 2 raised to the appropriate power rather than the actual number this represents. In my brain there are only three integral types to learn as the size of a byte is intuitively, in my PC based experience, 8 bits.

Range of Integral Primitives

Name	Size	Range
byte	8 bit	-2^7 to 2^7-1

short	16 bit	-2^{15} to $2^{15}-1$
int	32 bit	-2^{31} to $2^{31}-1$
long	64 bit	-2^{63} to $2^{63}-1$

Declaring integral literals

There are three ways to declare an integral literal. The default, as you might expect is decimal. Here are the options

Declaring 18 as an integral literal

Decimal	18
Octal	022 (Zero not letter O)
Hexidecimal	0x12

If you compile and run this little class you will get the value 18 output each time.

```
public class Lit{
public static void main(String[] argv){
        int i = 18;
        int j = 022;//Octal version: Two eights plus two
        int k = 0x12;//Hex version: One sixteen plus two
        System.out.println(i);
        System.out.println(j);
        System.out.println(k);
        }
}
```

Roberts and Heller describe 6 ways of declaring integral literals, because unusually for Java letter X is not case sensitive, nor are the letters A through F for hexadecimal notation. I find it easier to remember the three ways and that it the letters are not case sensitive.

The size of floating point primitives

Floating point numbers are slightly strange beasts as calculations can have some unexpected results. Thus to quote Peter Van Der Linden "The exact accuracy depends on the number being represented". As compensation for this variable accuracy, you do get to play with numbers large almost beyond imagination. Thus the largest double can store a number that corresponds to 17 followed by 307 zeros. So you can even store the value of the financial paper worth of Bill Gates (until Linux reaches reaches

total world domination, then an integer may do the job nicely).

Range of floating point types

float	32 bit
double	64 bit

Bear in mind that the default type for a literal with a decimal component is a *double* and not a *float*. This is slightly confusing as you might think that the default type for a "floating point number" would be a float. You may get questions in the exam in a similar form to the following.

Will the following compile?

```
float i = 1.0;
```

Intuition would tell you that this should compile cleanly. Unfortunately the exam is not designed to test your intuition. This will cause a compile time error because it attempts to assign a double to a float type. You can fix this code as follows

```
float i = 1.0F;
```

Or even

```
float i = (float) 1.0;
```

Indicating data type with a trailing letter

As demonstrated in the previous section you can tell Java that a numeric literal is of a particular type by giving it a trailing letter. These following are available

Suffix to indicate Data type

float	F
long	L
double	D

boolean and *char*

The *boolean* and *char* primitives are a little odd. If you have a background in C/C++ pay attention particularly to the *boolean* and make sure you do not bring any "false friends" from these languages. A *boolean* can not be assigned any other value than true or false. The values true or false do not correspond to 0, -1 or any other number.

A boolean can only be true or false, it cannot be assigned a number such as -1 or 0

The *char* primitive is the only unsigned primitive in Java, and is 16 bits long. The *char* type can be used to denote a Unicode character. Unicode is an alternative to ASCII that stores characters in 2 bytes instead of the 1 byte of ASCII. This gives you 65K worth of characters, which although not enough to cover all world scripts, is an improvement of the 255 characters of ASCII. Internationalisation is a whole subject on its own, and just because you can represent characters from Chinese or Vietnamese, it does not mean that they will display correctly if you have a standard English style operating system.

A *char* literal can be created by enclosing the character in single quotes thus

```
char a = 'z';
```

Note that single quotes ' are used not double ".

This works fine in my English centered little world but as Java is a world system a *char* may contain any of the characters available in the Unicode system. This is done by using four hex digits preceded by \u, with the whole expression in single quotes.

Thus the space character can be represented as follows

```
char c = ‘\u0020’
```

If you assign a plain number to a char it can be output as a alphabetic character. Thus the following will print out the letter A (ASCII value 65) and a space.

```
public class MyChar{
public static void main(String argv[]){
        char i = 65;
        char c = '\u0020';
        System.out.println(i);
        System.out.println("This"+c+"Is a space");
        }
}
```

Declaring string literals

The *String* type is not a primitive but it is so important that in certain areas Java treats it like one. One of these features is the ability to declare String literals instead of using *new* to instantiate a copy of the class.

String literals are fairly straightforward. Make sure you remember that *String* literals are enclosed in double quotes whereas a char literal takes single quotes.

Thus

```
String name = "James Bond"
```

See Objective 9.3 and 5.2 for more on the String class.

Quiz

Question 1)

Which of the following will compile correctly?

1) float f=10f;
2) float f=10.1;
3) float f=10.1f;
4) byte b=10b;

Question 2)

Which of the following will compile correctly?

1) short myshort=99S;
2) String name='Excellent tutorial Mr Green';
3) char c=17c;
4) int z=015;

Question 3)

Which of the following will compile correctly?

1) boolean b=-1;
2) boolean b2=false;
3) int i=019;
4) char c=99;

Answers

Answer 1)

1) float f=10f;

3) float f=10.1f;

There is no such thing as a byte literal and option 2 will cause an error because the default type for a number with a decimal component is a double.

Answer 2)

4)int z=015;

The letters c and s do not exist as literal indicators and a *String* must be enclosed with double quotes, not single as in this example.

Answer 3)

2) boolean b2=false;
4) char c=99;

Option 1 should be fairly obvious as wrong, as a *boolean* can only be assigned the values *true* of *false*. Option 3 is slightly more tricky as this is the correct way to declare an octal literal but you cannot use the numeric 9 if you are in base 8 where you have numbers 0 through 7. A little tricky one there perhaps.

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

The JLS
http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html#9151

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec4.html#obj13

Bruce Eckel's Thinking in Java
Chapter 2 "Special case: Primitive Types"
Chapter 3 "Literals"

Last updated

23 Aug 2001

Java2 Certification Tutorial

5)Operators and Assignments

Objective 1)

Determine the result of applying any operator including assignment operators and instanceof to operands of any type class scope or accessibility or any combination of these.

The *instanceof* operator

The *instanceof* operator is a strange beast, in my eyes it looks like it ought to be a method rather than an operator. You could probably write an great deal of Java code without using it, but you need to know about it for the purposes of the exam. It returns a boolean value as a test of the type of class at runtime. Effectively it is used to say

Is thisclass an *instanceof* thisotherclasss

If you use it in the following trivial way it does not seem particularly useful

```
public class InOf {
        public static void main(String argv[]){
        InOf i = new InOf();
               if(i instanceof InOf){
                   System.out.println("It's an instance of InOf");
                }//End if
        }//End of main
}
```

As you might guess this code will output

```
"It's an instance of InOf"
```

However circumstances may arise where you have access to an object reference that refers to something further down the hierarchy. Thus you may have a method that takes a Component as a parameter which may actually refer to a Button, Label or whatever. In this circumstance the *instanceof* operator can be used to test the type of the object, perform a matching cast and thus call the appropriate methods. The following example illustrates this

```
import java.awt.*;
public class InOfComp {
        public static void main(String argv[]){
        }//End of main
        public void mymethod(Component c){
        if( c instanceof Button){
                Button bc = (Button) c;
                bc.setLabel("Hello");
                }
                else
        if (c instanceof Label){
                Label lc = (Label) c;
                lc.setText("Hello");
                }
        }//End of mymethod
}
```

If the runtime test and cast were not performed the appropriate methods, *setLabel* and *setText* would not be available. Note that *instanceof* tests against a class name and not against an object reference for a class.

The + operator

As you might expect the + operator will add two numbers together. Thus the following will output 10

```
int p=5;
int q=5;
System.out.println(p+q);
```

The + operator is a rare example of operator overloading in Java. C++ programmers are used to being able to overload operators to mean whatever they define. This facility is not available to the programmer in Java, but it is so useful for Strings, that the plus sign is overridden to offer concatenation. Thus the following code will compile

```
String s = "One";
String s2 = "Two"
String s3 = "";
s3 = s+s2;
System.out.println(s3);
```

This will output the string OneTwo. Note there is no space between the two joined strings.

If you are from a Visual Basic background the following syntax may not be familiar

```
s2+=s3
```

This can also be expressed in Java in a way more familiar to a Visual Basic programmer as

```
s2= s2+s3
```

Under certain circumstances Java will make an implicit call to the *toString* method. This method as it's name implies tries to convert to a *String* representation. For an integer this means toString called on the number 10 will return the string "10".

This becomes apparent in the following code

```
int p = 10;
String s = "Two";
String s2 = "";
s2 = s + p;
System.out.println(s2);
```

This will result in the output

```
Two10
```

Remember that it is only the + operator that is overloaded for Strings. You will cause an error if you try to use the divide or minus (/ -) operator on Strings.

Assigning primitive variables of different types

A boolean cannot be assigned to a variable of any other type than another boolean. For the C/C++ programmers, remember that this means a boolean cannot be assigned to -1 or 0, as a Java boolean is not substitutable for zero or non zero.

With that major exception of the boolean type the general principle to learn for this objective is that widening conversions are allowed, as they do not compromise accuracy. Narrowing conversions are not allowed as they would result in the loss of precision. By widening I mean that a variable such as a byte that occupies one *byte* (eight bits) may be assigned to a variable that occupies more bits such as an integer.

However if you try to assign an integer to a byte you will get a compile time error

```
byte b= 10;
int i = 0;
b = i;
```

Primitives may be assigned to "wider" data types, a boolean can only assigned to another boolean

As you might expect you cannot assign primitives to objects or vice versa. This includes the wrapper classes for primitives. Thus the following would be illegal

```
int j=0;
```

```
Integer k = new Integer(99);
j=k; //Illegal assignment of an object to a primitive
```

An important difference between assigning objects and primitives is that primitives are checked at compile time whereas objects are checked at runtime. This will be covered later as it can have important implications when an object is not fully resolved at compile time.

You can, of course, perform a cast to force a variable to fit into a narrower data type. This is often not advisable as you will loose precision, but if you really want enough rope, Java uses the C/C++ convention of enclosing the data type with parenthesis i.e. (), thus the following code will compile and run

```
public class Mc{
public static void main(String argv[]){
    byte b=0;
    int i = 5000;
    b = (byte) i;
    System.out.println(b);
    }
}

The output is
-120
```

Possibly not what would be required.

Assigning object references of different types

When assigning one object reference to another the general rule is that you can assign up the inheritance tree but not down. You can think of this as follows. If you assign an instance of Child to Base, Java knows what methods will be in the Child class. However a child may have additional methods to its base class. You can force the issue by using a cast operation.

Object references can be assigned up the hierarchy from Child to Base.

The following example illustrates how you can cast an object reference up the hierarchy

```
class Base{}
public class ObRef extends Base{
public static void main(String argv[]){
        ObRef o = new ObRef();
        Base b = new Base();
        b=o;//This will compile OK
        /*o=b;         This would cause an error indicating
                an explicit cast is needed to cast Base
                to ObRef */
```

```
        }
}
```

The bit shifting operators

I hate bit the whole business of bit shifting. It requires filling your brain with a non intuitive capability that an infinitesimally small number of programmers will ever use. But that's all the more reason to learn it especially for the exam as you probably won't learn it via any other means. This objective could do with a whole bunch of warning or banana skin icons, so if anyone has any good ones send them to me. The results can be surprising, particularly on negative numbers.

To understand it you have to be fairly fluent in at thinking in binary, ie knowing the value of the bit at each position i.e.

```
32, 16, 8, 4, 2, 1
```

If you are from a C/C++ background you can take slight comfort from the fact that the meaning of the right shift operator in Java is less ambiguous than in C/C++. In C/C++ the right shift could be signed or unsigned depending on the compiler implementation. If you are from a Visual Basic background, welcome to programming at a lower level.

Note that the objective only asks you to understand the results of applying these operators to *int* values. This is handy as applying the operators to a *byte* or *short*, particularly if negative, can have some very unexpected results.

Signed shifting << and >>

The left and right shift operators move the bit pattern to the left or right and leave the sign bit alone.

For positive numbers the results are fairly predictable. Thus the signed shift of the positive number gives the following results

```
int x = 14;
int y = 0;
y = x >>2;


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

Outputs 3, one bit is lost and falls off the right hand side

```
int x = 14;
int y = 0;
y = x <<2;
System.out.println(y);
```

Outputs 56

So what do you expect to get when you right shift a negative number? You might expect the same result as right shifting a positive number except that the result keeps the negative sign. If we shift 4 places, what actually happens is that the spaces, left moving the other bits across, take on the value of the most significant bit (i.e. the sign bit). The effect of this is that each shift still divides a minus number by two. This sounds like it will be easy enough to understand until you realise the implication of twos complement storage of binary numbers.

Twos complement works a little like a physical odomenter on a cars clock. Imagine you wind back to zero and then go below zero into the negative numbers. The first number you get to will not be one, but one below the biggest number you can represent with all the available wheels. If this sounds rather unlikely fire up the windows calculator, put it into scientific mode, enter a minus number and then switch to binary mode. This will display the bit pattern for the number you just entered.

If all of this talk of bit patterns and twos complement representation does your head in a bit you may like to think of the bit shifting as a process of repeated multiplication or division by two. This approach works fine until you start to shift a negative number to the right so it loses bits from the right hand side.

Unsigned Right Shift >>>

The unsigned right shift >>> performs a shift without attaching any significance to the sign bit. Thus in an integer, all 32 bits are shifted by the value of the operand and padding on the left uses zeros. This also generally has the effect of making a negative number positive. I say generally because before the shift is performed a mod 32 operation is performed on the operand(or mod 64 for a long) The unsighed Right Shift can lead can lead to some very weird results. The following statement

```
System.out.println(-1 >>>1);
```

Results in the following output

```
2147483647
```

The exam probably won't ask you to give the exact result but it might give you some alternatives such as 0, -1 etc etc and you have to pick the most likely result.

What would you expect the result of the following statement to be?

System.out.println(-1 >>> 32);

If you read that as -1 being shifted 32 places to the right whilst ignoring the significance of the sign place then the real answer of -1 may be a surprise. The reason is the mod 32 that is performed on the operand before the shift. Thus if you divide 32 by 32 you get zero and if you perform an unsigned shift of zero places to the right you still end up with -1. Don't dismiss this as an irrelevent peculiarity as it may come up in the exam.

A mod 32 is performed on the shift operand which affects shifts of more than 32 places

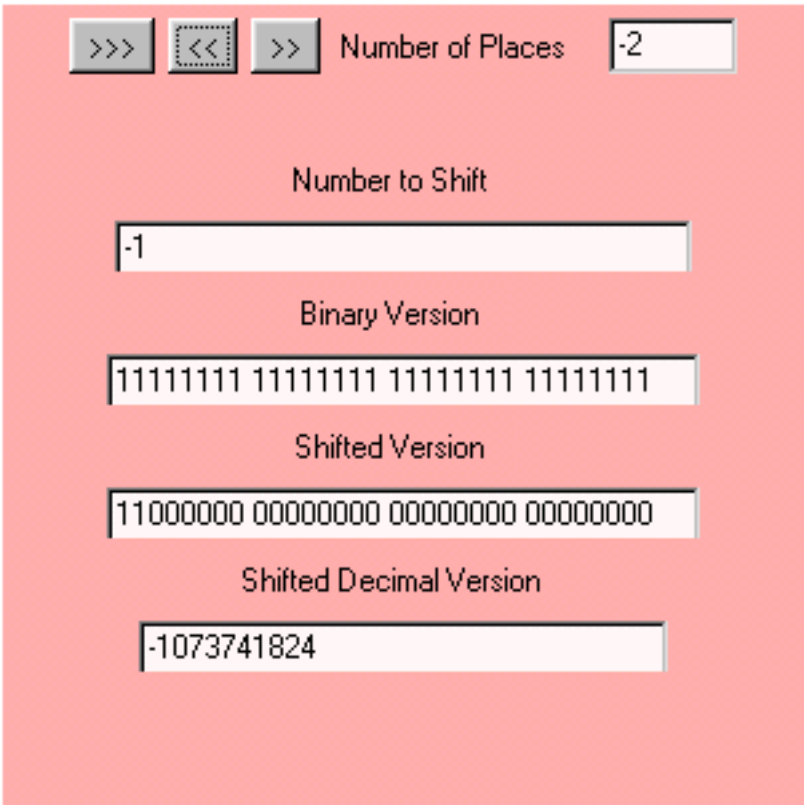
I have created an applet that allows you to try out the various shift operations and see both the decimal

and bit pattern results. I have included the source code so you can see how it works, check it out at

http://www.software.u-net.com/applets/BitShift/BitShiftAr.html

Here is a screen shot of this applet in use

BitShift Applet



Quiz

Question 1)

Given the following classes which of the following will compile without error?

```
interface IFace{}
class CFace implements IFace{}
class Base{}
public class ObRef extends Base{
public static void main(String argv[]){
        ObRef ob = new ObRef();
        Base b = new Base();
```

```
        Object o1 = new Object();
        IFace o2 = new CFace();
        }
}
```

1) o1=o2;
2) b=ob;
3) ob=b;
4) o1=b;

Question 2)

Given the following variables which of the following lines will compile without error?

```
String s = "Hello";
long l = 99;
double d = 1.11;
int i = 1;
int j = 0;
```

```
1) j= i <<s;
2) j= i<<j;
3) j=i<<d;
4)j=i<<l;
```

Question 3)

Given the following variables

```
char c = 'c';
int i = 10;
double d = 10;
long l = 1;
String s = "Hello";
```

Which of the following will compile without error?

1) c=c+i;
2) s+=i;
3) i+=s;
4) c+=s;

Question 4)

What will be output by the following statement?

```
System.out.println(-1 >>>1);
```

1) 0
2) -1
3) 1
4) 2147483647

Question 5)

What will be output by the following statement?

```
System.out.println(1 <<32);
```

1) 1
2) -1
3) 32
4)-2147483648

Question 6)

Which of the following are valid statements?

1) System.out.println(1+1);
2) int i= 2+'2';
3) String s= "on"+'one';
4) byte b=255;

Answers

Answer 1)

```
1)o1=o2;
2)b=ob;
4)o1=b;
```

Answer 2)

```
2)j= i<<j;
4)j=i<<l;
```

Answer 3)

2)s+=i;

If you want to test these possibilities, try compiling this code

```
public class Llandaff{
        public static void main(String argv[]){
```

```
        Llandaff h = new Llandaff();
        h.go();
        }

        public void go(){
                char c = 'c';
                int i = 10;
                double d = 10;
                long l = 1;
                String s = "Hello";
                //Start commenting these out till it all compiles
                c=c+i;
                s+=i;
                i+=s;
                c+=s;
        }
}
```

Answer 4)

4) 2147483647

Although you might not be able to come up with that number in your head, understanding the idea of the unsigned right shift will indicate that all the other options are not correct.

Answer 5)

1) 1

With the left shift operator the bits wil "wrap around". Thus the result of

```
System.out.println(1 <<31);
would be -2147483648
```

Answer 6)

1) System.out.println(1+1);
2) int i= 2+'2';
Option 3 is not valid because single quotes are used to indicate a character constant and not a string.
Option 4 will not compile becuase 255 is out of the range of a byte

---

Other Sources on this topic

The Sun Tutorial
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html
(nothing on *instanceof* that I could find at Sun)

Richard Baldwin

http://home.att.net/~baldwin.dick/Intro/Java022.htm#bitwiseoperations
(nothing on *instanceof* that I could find here either)

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec5.html#obj15

Twos Compliment Math
http://www.duke.edu/~twf/cps104/twoscomp.html

Article on Binary/Hex/Decimal Numbers by Jane Griscti
http://www.janeg.ca/scjp/oper/binhex.html

Shfting from JavaRanch
www.javaranch.com/campfire/StoryBits.jsp

Last updated
12 August 2001

Java2 Certification Tutorial

5) Operators and Assignments

Objective 2)

Determine the result of applying the boolean equals(Object) method to objects of any combination of the classes java.lang.String java.lang.Boolean and java.lang.Object.

If (like me) you are from a background with Visual Basic, the idea of any sort of comparison apart from using some variation of the = operator may seem alien. "In the real world" this is particularly important with reference to *Strings* as they are so commonly used,however For the purpose of the exam you may get questions that ask about the *equals* operator with reference to *Object* references and *Boolean*. Note that the question asks about the *Boolean* class not the *boolean* primitive (from which you cannot invoke a method)

The difference between *equals* and ==

The *equals* method can be considered to perform a deep comparison of the value of an object, whereas the == operator performs a shallow comparison. The *equals* method compares what an object points to rather than the pointer itself (if we can admit that Java has pointers). This indirection may appear clear to C++ programmers but there is no direct comparison in Visual Basic.

Using the *equals* method with String

The *equals* method returns a *boolean* primitive. This means it can be used to drive an *if*, *while* or other looping statement. It can be used where you would use the == operator with a primitive. The operation of the *equal* method and == operator has some strange side effects when used to compare Strings. This is one occasion when the immutable nature of *Strings*, and the way they are handled by Java, can be confusing.

There are two ways of creating a String in Java. The one way does not use the *new* operator. Thus normally a String is created

```
String s = new String("Hello");
```

but a slightly shorter method can be used

```
String s= "GoodBye";
```

Generally there is little difference between these two ways of creating strings, but the Exam may well ask questions that require you to know the difference.

The creation of two strings with the same sequence of letters without the use of the *new* keyword will create pointers to the same String in the Java String pool. The String pool is a way Java conserves resources. To illustrate the effect of this

```
String s = "Hello";
String s2 = "Hello";
if (s==s2){
    System.out.println("Equal without new operator");
    }
String t = new String("Hello");
string u = new String("Hello");
if (t==u){
    System.out.println("Equal with new operator");
    }
```

From the previous objective you might expect that the first output "Equal without new operator" would never be seen as s and s2 are different objects, and the == operator tests what an object points to, not its value. However because of the way Java conserves resources by re-using identical strings that are created without the new operator s and s2 have the same "address" and the code does output the string

"Equal without new operator"

However with the second set of strings t and u, the *new* operator forces Java to create separate strings. Because the == operator only compares the address of the object, not the value, t and u have different addresses and thus the string "Equal with new operator" is never seen.

The *equals* method applied to a String, however that String was created, performs a character by character comparison

The business of the use of the string pool and the difference between the use of == and the equals method is not obvious, particularly if you have a Visual Basic background. The best way to understand it is to create some examples for yourself to see how it works. Try it with various permutations of identical strings created with and without the new operator.

Using the *equals* method with Boolean

The requirement to understand the use of the equals operator on *java.lang.Boolean* is a potential gotcha. *Boolean* is a wrapper object for the *boolean* primitive. It is an object and using *equals* on it will test

According to the JDK documentation the *equals* method of the Boolean wrapper class

"Returns true if and only if the argument is not null and is a Boolean object that contains the same boolean value as this object".

eg

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);
if(b1.equals(b2)){
    System.out.println("We are equal");
    }
```

As a slight aside on the subject of *boolean* and *Boolean*, once you are familiar with the *if* operator in Java you will know you cannot perform the sort of implicit conversion to a boolean beloved of bearded C/C++ programmers. By this I mean

```
int x =1;
if(x){
    //do something, but not in Java
    }
```

This will not work in Java because the parameter for the *if* operator must be a boolean evaluation, and Java does not have the C/C++ concept whereby any non null value is considered to be true. However you may come across the following in Java

```
boolean b1=true;
if(b1){
    //do something in java
    }
```

Although this is rather bad programing practice it is syntactically correct, as the parameter for the *if* operation is a boolean

Using the *equals* method with Object

Due to the fundamental design of Java an instance of any class is also an instance of java.lang.Object. Testing with equals performs a test on the Object as a result of the return value of the *toString*() method. For an Object the toString method simply returns the memory address. Thus the result is the equivalent of performing a test using the == operator. As java is not designed to manipulate memory addresses or pointers this is not a particularly useful test.

Take the following example

```
public class MyParm{
public static void main(String argv[]){
        Object m1 = new Object();
        Object m2 = new Object();
        System.out.println(m1);
        System.out.println(m2);
        if (m1.equals(m2)){
               System.out.println("Equals");
                }else{
                System.out.println("Not Equals");
                }
        }
}
```

If you attempt to compile and run this code you will get an output of

```
java.lang.Object@16c80b
java.lang.Object@16c80a

Not Equals
```

Those wierd values are memory addresses, and probably not what you want at all.

Quiz

Questions

Question 1)

What will happen when you attempt to compile and run the following code?

```
public class MyParm{
public static void main(String argv[]){
        String s1= "One";
        String s2 = "One";
        if(s1.equals(s2)){
                System.out.println("String equals");
                }
        boolean b1 = true;
        boolean b2 = true;
```

```
        if(b1.equals(b2)){
                        System.out.println("true");
                        }
        }
}
```

1) Compile time error
2) No output
3) Only "String equals"
4) "String equals" followed by "true"

Question 2)

What will happen when you attempt to compile and run the following code?

```
String s1= "One";
String s2 = new String("One");
if(s1.equals(s2)){
               System.out.println("String equals");
          }
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);
if(b1==b2){
          System.out.println("Boolean Equals");
   }
```

1) Compile time error
2) "String equals" only
3) "String equals" followed by "Boolean equals"
4) "Boolean equals" only

Question 3)

What will be the result of attempting to compile and run the following code?

```
        Object o1 = new Object();
        Object o2 = new Object();
        o1=o2;
        if(o1.equals(o2))
                System.out.println("Equals");
        }
```

1) Compile time error
2) "Equals"
3) No output
4) Run time error

Answers

Answer 1)

1) Compile time error

The line b1.equals() will cause an error because b1 is a primitive and primitives do not have any methods. If it had been created as the primitive wrapper Boolean then you could call the equals method.

Answer 2)

2) "String equals" only

Testing an instance of the Boolean primitive wrapper with the == operator simply tests the memory address.

Answer 3)

2) "Equals"

Because the one instance of Object has been assigned to the other with the line

o1=o2;

They now point to the same memory address and the test with the *equals* method will return true

Other sources on this topic

Jyothi Krishnan on this topic at http://www.geocities.com/SiliconValley/Network/3693/obj_sec5.html#obj16 **Michael Thomas** http://www.michael-thomas.com/java/JCP_Operators.htm#equals()

Last updated
10 January 2000

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

5) Operators and Assignments

Objective 3)

In an expression involving the operators & | && || and variables of known values state which operands are evaluated and the value of the expression.

It is easy to forget which of the symbols mean logical operator and which mean bitwise operations, make sure you can tell the difference for the exam. If you are new to these operators it might be worth trying to come up with some sort of memory jogger so you do not get confused between the bitwise and the logical operators. You might like to remember the expression "Double Logic" as a memory jerker.

The short circuit effect with logical operators

The logical operators *(&& ||)* have a slightly peculiar effect in that they perform "short-circuited" logical AND and logical OR operations as in C/C++. This may come as a surprise if you are a from a Visual Basic background as Visual Basic will evaluate all of the operands. The Java approach makes sense if you consider that for an AND, if the first operand is false it doesn't matter what the second operand evaluates to, the overall result will be false. Also for a logical OR, if the first operand has turned out true, the overall calculation will show up as true because only one evaluation must return true to return an overall true. This can have an effect with those clever compressed calculations that depend on side effects. Take the following example.

```
public class MyClass1{
                        public static void main(String argv[]){
                        int Output=10;
                        boolean b1 = false;
                        if((b1==true) && ((Output+=10)==20))
```

```
                        {
                        System.out.println("We are equal "+Output);
                        }else
                        {
                        System.out.println("Not equal! "+Output);
                        }
                }
}
```

The output will be "Not equal 10". This illustrates that the Output +=10 calculation was never performed because processing stopped after the first operand was evaluated to be false. If you change the value of b1 to true processing occurs as you would expect and the output is "We are equal 20";.

This may be handy sometimes when you really don't want to process the other operations if any of them return false, but it can be an unexpected side effect if you are not completely familiar with it.

The bitwise operators

The & and | operators when applied to integral bitwise AND and OR operations. You can expect to come across questions in the exam that give numbers in decimal and ask you to perform bitwise AND or OR operations. To do this you need to be familiar with converting from decimal to binary and learn what happens with the bit patterns. Here is a typical example

What is the result of the following operation

```
3 | 4
```

The binary bit pattern for 3 is

```
11
```

The binary bit pattern for 4 is

```
100
```

For performing a binary OR, each bit is compared with the bit in the same position in the other number. If either bit contains a 1 the bit in the resulting number is set to one. Thus for this operation the result will be binary

```
111
```

Which is decimal 7.

The objectives do not specifically ask for knowledge of the bitwise XOR operation, performed with ^

Thinking in binary

If you do not feel comfortable thinking in binary (I am much more comfortable in decimal), you may want to do some exercises to help master this topic and also the bitwise shift operators topic. If you are running windows you may find it helpful to use the windows calculator in scientific mode. To do this

choose View and switch from the default standard to scientific mode. In Scientific mode you can switch between viewing numbers ad decimal and binary, this displays the bit pattern of numbers. Here is another handy trick I wish I had known before I wrote my BitShift applet (see the applets menu from the front of this site), is how to use the Integer to display bit patterns. Here is a little program to demonstrate this.

```
public class BinDec{
public static void main(String argv[]){
        System.out.println(Integer.parseInt("11",2));
        System.out.println(Integer.toString(64,2));
       }
}
```

If you compile and run this program the output will be

3
1000000

Note how the program converts the bit pattern 11 into the decimal equivalent of the number 3 and the decimal number 64 into its equivalent bit pattern. The second parameter to each method is the "radix" or counting base. Thus in this case it is dealing with numbers to the base 2 whereas we normally deal with numbers to the base 10.

Quiz

Questions

Question 1)

What will happen when you attempt to compile and run the following code?

```
int Output=10;
boolean b1 = false;
if((b1==true) && ((Output+=10)==20)){
    System.out.println("We are equal "+Output);
    }else
   {
    System.out.println("Not equal! "+Output);
}
```

1) Compile error, attempting to peform binary comparison on logical data type
2) Compilation and output of "We are equal 10"

3) Compilation and output of "Not equal! 20"
4) Compilation and output of "Not equal! 10"

Question 2)

What will be output by the following line of code?

System.out.println(010|4);

1) 14
2) 0
3) 6
4) 12

Question 3)

Which of the following will compile without error?

1)

```
int i=10;
int j = 4;
System.out.println(i||j);
```

2)

```
int i=10;
int j = 4;
System.out.println(i|j);
```

3)

```
boolean b1=true;
boolean b2=true;
System.out.println(b1|b2);
```

4)

```
boolean b1=true;
boolean b2=true;
System.out.println(b1||b2);
```

Answers

Answer 1)

4) Compilation and output of "Not equal! 10"

The output will be "Not equal 10". This illustrates that the Output +=10 calculation was never performed because processing stopped after the first operand was evaluated to be false. If you change the value of b1 to true processing occurs as you would expect and the output is "We are equal 20";.

Answer 2)

4) 12

As well as the binary OR objective this questions requires you to understand the octal notation which means that the leading zero not means that the first 1 indicates the number contains one eight and nothing else. Thus this calculation in decimal means

```
8|0
```

To convert this to binary means

```
1000
0100
----
1100
----
```

The | bitwise operator means that each position where there is a 1, results in a 1 in the same position in the answer.

Answer 3)

2,3,4

Option 1 will not compile because it is an attempt to perform a logical OR operation on a an integral types. A logical or can only be performed with boolean arguments.

Other sources on this topic

The Sun Tutorial
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html

Richard Baldwin
http://home.att.net/~baldwin.dick/Intro/Java022f.htm

Last updated
10 Jan 2000

Java2 Certification Tutorial

5)Operators and Assignments

Objective 4)

Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.

Note on the Objective

This objective appears to be asking you to understand what happens when you pass a value into a method. If the code in the method changes the variable, is that change visible from outside the method?. Here is a direct quote from Peter van der Lindens Java Programmers FAQ (available at http://www.afu.com)

//Quote
All parameters (values of primitive types and values that are references to objects) are passed by value. However this does not tell the whole story, since objects are always manipulated through reference variables in Java. Thus one can equally say that objects are passed by reference (and the reference variable is passed by value). This is a consequence of the fact that variables do not take on the values of "objects" but values of "references to objects" as described in the previous question on linked lists.

Bottom line: The caller's copy of primitive type arguments (int, char, etc.) _do not_ change when the corresponding parameter is changed. However, the fields of the caller's object _do_ change when the called method changes the corresponding fields of the object (reference) passed as a parameter.

//End Quote

If you are from a C++ background you will be familiar with the concept of passing parameters either by value or by reference using the & operator. There is no such option in Java as everything is passed by value. However it does not always appear like this. If you pass an object it is an object reference and you cannot directly manipulate an object reference.

Thus if you manipulate a field of an object that is passed to a method it has the effect as if you had passed by reference (any changes will be still be in effect on return to the calling method)..

Object references as method parameters

Take the following example

```
class ValHold{
        public int i = 10;
}
public class ObParm{
public static void main(String argv[]){
        ObParm o = new ObParm();
        o.amethod();
        }
        public void amethod(){
                ValHold v = new ValHold();
                v.i=10;
                System.out.println("Before another = "+ v.i);
                another(v);
                System.out.println("After another = "+ v.i);
        }//End of amethod
        public void another(ValHold v){
                v.i = 20;
                System.out.println("In another = "+ v.i);
        }//End of another
}
```

The output from this program is

```
Before another = 10
In another = 20

After another = 20
```

See how the value of the variable *i* has been modified. If Java always passes by value (i.e. a copy of a variable), how come it has been modified? Well the method received a copy of the handle or object reference but that reference acts like a pointer to the real value. Modifications to the fields will be reflected in what is pointed to. This is somewhat like how it would be if you had automatic dereferencing of pointers in C/C++.

Primitives as method parameters

When you pass primitives to methods it is a straitforward pass by value. A method gets its own copy to play with and any modifications are not reflected outside the method. Take the following example.

```
public class Parm{
public static void main(String argv[]){
                Parm p = new Parm();
                p.amethod();
        }//End of main
        public void amethod(){
              int i=10;
              System.out.println("Before another i= " +i);
              another(i);
              System.out.println("After another i= " + i);
        }//End of amethod
        public void another(int i){
              i+=10;
             System.out.println("In another i= " + i);
        }//End of another
}
```

The output from this program is as follows

```
Before another i= 10
In another i= 20
After another i= 10
```

Quiz

Questions

Question 1)

Given the following code what will be the output?

```
class ValHold{
        public int i = 10;
}
public class ObParm{
```

```
public static void main(String argv[]){
        ObParm o = new ObParm();
        o.amethod();
        }
        public void amethod(){
              int i = 99;
                ValHold v = new ValHold();
                v.i=30;
                another(v,i);
                System.out.println(v.i);
        }//End of amethod
        public void another(ValHold v, int i){
                i=0;
                v.i = 20;
                ValHold vh = new ValHold();
                v =  vh;
                System.out.println(v.i+ " "+i);
        }//End of another
}
```

1) 10,0, 30
2) 20,0,30
3) 20,99,30
4) 10,0,20

Answers

Answer 1)

4) 10,0,20

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/java/javaOO/arguments.html

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec5.html#obj18

Last updated
11 Jan 2000

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

Recommended book on this topic

Java Design Patters A Tutorial

If you really want to understand the thinking behind the design of many of the Java classes such as I/O AWT/Swing and others this book shows the underlying thought process behind this approaches. This Object Orientation in action, rather than in theory and in trivial examples. This book covers much more than you strictly need for the purposes of the certification exam but you will see these patterns constantly referred to in the world of software engineering, so you might want to get used to them sooner rather than later. You might also consider the original "GOF" book (see reviews at amazon).

Buy from Amazon.com or from Amazon.co.uk

---

6)Overloading overriding runtime type and object orientation

Objective 1)

State the benefits of encapsulation in object oriented design and write code that implements tightly encapsulated classes and the relationships "is a" and "has a".

"*Is a*" vs "*has a*" relationship

This is a very basic OO question and you will probably get a question on it in the exam. Essentially it seeks to find out if you understand when something is referring the type of class structure that an object belongs to and when it refers to a method or field that a class has.

Thus a cat IS A type of animal and a cat HAS a tail. Of course the distinction can be blurred. If you were a zoologist and knew the correct names for groups of animal types you might say a cat IS A

longlatinwordforanimalgroupwithtails.

but for the purpose of the exam this is not a consideration.

The exam questions tend to be of the type whereby you get a text description of a potential hierarchy and you get questions as to what ought to be a field and what ought to be a new class type as a child. These questions can look complex at first glance, but if you read them carefully they are fairly obvious.

Encapsulation

The Java 1.1 objectives did not specifically mention encapsulation, though you would be hard pressed to study Java and not come across the concept. Encapsulation involves separating the interface of a class from its implementation. This means you can't "accidentally" corrupt the value of a field, you have to use a method to change a value.

Encapsulation involves hiding data of a class and allowing access only through a public interface.

To do this usually involves the creation of private variables (fields) where the value is updated and retrieved via methods. The standard naming convention for these methods is

- setFieldName
- getFieldName

For example if you were changing the Color of a shape you might create a method pair in the form

```
public void setColor(Color c){
    cBack = c;
    }
public Color getColor(){
   return cBack;
    }
```

The main access control keywords for variables are

- public

- private
- protected

Do not be mislead into thinking that the access control system is to do with security. It is not designed to prevent a programmer getting at variables, it is to help avoid unwanted modification.

The standard approach using the Color example above would be for the Color field *cBack* to be private. A private field is only visible from within the current class. This means a programmer cannot accidentally write code from another class that changes the value. This can help to reduce the introduction of bugs.

The separation of interface and implementation makes it easier to modify the code within a class without breaking any other code that uses it.

For the class designer this leads to the ability to modify a class, knowing that it will not break programs that use it. A class designer can insert additional checking routines for "sanity checks" for the modification of fields. I have worked on insurance projects where it was possible for clients to have an age of less than zero. If such a value is stored in a simple field such as an integer, there is no obvious place to store checking routines. If the age were only accessible via *set* and *get* methods it will be possible to insert checks against zero or negative ages in such a way that it will not break existing code. Of course as development continues more situations may be discovered that need checking against.

For the end user of the class it means they do not have to understand how the internals work and are presented with a clearly defined interface for dealing with data. The end user can be confident that updates to the class code will not break their existing code.

Runtime type

Because polymorphism allows for the selection of which version of a method executes at runtime, sometimes it is not obvious which method will be run. Take the following example.

```
class Base {
int i=99;
public void amethod(){
        System.out.println("Base.amethod()");
        }
}

public class RType extends Base{
int i=-1;
        public static void main(String argv[]){
        Base b = new RType();//<= Note the type
        System.out.println(b.i);
        b.amethod();
        }
        public void amethod(){
                System.out.println("RType.amethod()");
```

```
        }

}
```

Note how the type of the reference is *b Base* but the type of actual class is *RType*. The call to *amethod* will invoke the version in *RType* but the call to output *b.i* will reference the field *i* in the *Base* class.

Quiz

Question 1)

Consider you have been given the following design

"A person has a name, age, address and sex. You are designing a class to represent a type of person called a patient. This kind of person may be given a diagnosis, have a spouse and may be alive". Given that the person class has already been created, what of the following would be appropriate to include when you design the patient class?

1) registration date
2) age
3) sex
4)diagnosis

Question 2)

What will happen when you attempt to compile and run the following code?

```
class Base {
int i=99;
public void amethod(){
        System.out.println("Base.amethod()");
        }
        Base(){
             amethod();
        }

}
```

```
public class RType extends Base{
int i=-1;
        public static void main(String argv[]){
        Base b = new RType();
        System.out.println(b.i);
        b.amethod();
        }
        public void amethod(){
                System.out.println("RType.amethod()");
        }
}
```

```
1)
RType.amethod
-1
RType.amethod
```

2)

```
RType.amethod
99
RType.amethod
```

3)

```
99
RType.amethod
```

4)

```
Compile time error
```

Question 3)

Your chief Software designer has shown you a sketch of the new Computer parts system she is about to create. At the top of the hierarchy is a Class called Computer and under this are two child classes. One is called LinuxPC and one is called WindowsPC. The main difference between the two is that one runs the Linux operating System and the other runs the Windows System (of course another difference is that one needs constant re-booting and the other runs reliably). Under the WindowsPC are two Sub classes one called Server and one Called Workstation. How might you appraise your designers work?

1) Give the goahead for further design using the current scheme
2) Ask for a re-design of the hierarchy with changing the Operating System to a field rather than Class type
3) Ask for the option of WindowsPC to be removed as it will soon be obsolete
4) Change the hierarchy to remove the need for the superfluous Computer Class.

Question 4)

Given the following class

```
class Base{
        int Age=33;
}
```

How might you change improve the class with respect to accessing the field Age?

1) Define the variable Age as private
2) Define the variable Age as protected
3) Define the variable Age as private and create a get method that returns it and a set method that updates it
4) Define the variable Age as protected and create a set method that returns it and a get method that updates it

Question 5)

Which of the following are benefits of encapsulation

1) All variables can be manipulated as Objects instead of primitives
2) by making all variables protected they are protected from accidental corruption
3) The implementation of a class can be changed without breaking code that uses it
4) Making all methods protected prevents accidental corruption of data

Question 6)

Name three principal characteristics of Object Oriented programming?

1) encapsulation, dynamic binding, polymorphism
2) polymorphism, overloading, overriding
3) encapsulation, inheritance, dynamic binding
4) encapsulation, inheritance, polymorphism

Question 7)

How can you implement encapsulation in a class

1) make all variables protected and only allow access via methods
2) make all variables private and only allow access via methods
3) ensure all variables are represented by wrapper classes
4) ensure all variables are accessed through methods in an ancestor class

Answers

Answer 1)

1) registration date
4) diagnosis

Registration date is a reasonable additional field for a patient, and the design specifically says that a patient should have a diagnosis. As the patient is a type of person, it should have the fields age and sex available (assuming they were not declared to be private).

Answer 2)

2)

```
RType.amethod
99
RType.amethod
```

If this answer seems unlikley, try compiling and running the code. The reason is that this code creates an instance of the RType class but assigns it to a reference of a the Base class. In this situation a reference to any of the fields such as i will refer to the value in the Base class, but a call to a method will refer to the method in the class type rather than its reference handle.

Answer 3)

2) Ask for a re-design of the hierarchy with changing the Operating System to a field rather than Class type

Answer 4)

3) Define the variable Age as private and create a *get* method that returns it and a *set* method that updates it

Answer 5)

3) The implementation of a class can be changed without breaking code that uses it

Answer 6)

4) encapsulation, inheritance, polymorphism

I got this question at a job interview once. I got the job. Can't be certain you will get anything similar in the exam, but its handy to know.

Answer 7)

2) make all variables private and only allow access via methods

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/java/concepts/index.html

Richard Baldwin Covers this topic at
http://www.Geocities.com/Athens/Acropolis/3797/Java004.htm#an initial description of oop
(This is general stuff on OOP rather than concentrating on "is a" "has a")

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec6.html#obj19

Java 1.1 Unleashed
http://www.itlibrary.com/reference/library/1575212986/htm/ch05.htm
(See the section on encapsulation)

Chapter 6 from the Roberts, Heller and Earnest book
http://developer.java.sun.com/developer/Books/certification/page1.html

Last updated
12 June 2001

Java2 Certification Tutorial

6) Overloading, overriding, runtime type and object orientation

Objective 2)

Write code to invoke overridden or overloaded methods and parental or overloaded constructors; and describe the effect of invoking these methods.

Comment on the objective

The terms *overloaded* and *overridden* are similar enough to give cause for confusion. My way of remembering it is to imagine that something that is overriden has literally been ridden over by a heavy vehicle and no longer exists in its own right. Something that is overloaded is still moving but is loaded down with lots of functionality that is causing it plenty of effort. This is just a little mind trick to distinguish the two, it doesn't have any bearing of the reality on the operations in Java.

Overloading methods

Overloading of methods is a compiler trick to allow you to use the same name to perform different actions depending on parameters.

Thus imagine you were designing the interface for a system to run mock Java certification exams (who could this be?). An answer may come in as an integer, a boolean or a text string. You could create a version of the method for each parameter type and give it a matching name thus

```
markanswerboolean(boolean answer){
        }
```

```
markanswerint(int answer){
        }

markanswerString(String answer){
        }
```

This would work but it means that future users of your classes have to be aware of more method names than is strictly necessary. It would be more useful if you could use a single method name and the compiler would resolve what actual code to call according to the type and number of parameters in the call.

This is the heart of overloading methods, part of what is known as *polymorphism*.

There are no keywords to remember in order to overload methods, you just create multiple methods with the same name but different numbers and or types of parameters. The names of the parameters are not important but the number and types must be different. Thus the following is an example of an overloaded markanswer method

```
void markanswer(String answer){
        }
void markanswer(int answer){
       }
```

The following is not an example of overloading and will cause a compile time error indicating a duplicate method declaration.

```
void markanswer(String answer){
        }

void markanswer(String title){
        }
```

The return type does not form part of the signature for the purpose of overloading.

Thus changing one of the above to have an *int* return value will still result in a compile time error, but this time indicating that a method cannot be redefined with a different return type.

Overloaded methods do not have any restrictions on what exceptions can be thrown. That is something to worry about with overriding.

Overloaded methods are differentiated only on the number, type and order of parameters,not on the return type of the method

Overriding methods

Overriding a method means that its entire functionality is being replaced. Overriding is something done in a child class to a method defined in a parent class. To override a method a new method is defined in the child class with exactly the same signature as the one in the parent class. This has the effect of shadowing the method in the parent class and the functionality is no longer directly accessible.

Java provides an example of overriding in the case of the equals method that every class inherits from the granddady parent Object. The inherited version of *equals* simply compares where in memory the instance of the class references. This is often not what is wanted, particularly in the case of a String. For a string you would generally want to do a character by character comparison to see if the two strings are the same. To allow for this the version of equals that comes with String is an overriden version that performs this character by character comparison.

Invoking base class constructors

A constructor is a special method that is automatically run every time an instance of a class is created. Java knows that a method is a constructor because it has the same name as the class itself and no return value. A constructor may take parameters like any other method and you may need to pass different parameters according to how you want the class initialised. Thus if you take the example of the Button class from the AWT package its constructor is overloaded to give it two versions. One is

- Button()
- Button(String label)

Thus you can create a button with no label and give it one later on, or use the more common version and assign the label at creation time.

Constructors are not inherited however, so if you want to get at some useful constructor from an ancestor class it is not available by default. Thus the following code will not compile

```
class Base{
public  Base(){}
public  Base(int i){}
}


public class MyOver extends Base{
public static void main(String argvp[]){
        MyOver m = new MyOver(10);//Will NOT compile
        }
}
```

The magic keyword you need to get at a constructor in an ancestor is *super*. This keyword can be used as if it were a method and passed the appropriate parameters to match up with the version of the parental constructor you require. In this modified example of the previous code the keyword *super* is used to call the single integer version of the constructor in the base class and the code compiles without complaint.

```
class Base{
```

```
public Base(){}
public Base(int i){}
}

public class MyOver extends Base{
public static void main(String arg[]){
                MyOver m = new MyOver(10);
                }
        MyOver(int i){
                super(i);
        }
}
```

Invoking constructors with *this()*

In the same way that you can call a base class constructor using *super()* you can call another constructor in the current class by using *this* as if it were a method. Thus in the previous example you could define another constructor as follows

```
        MyOver(String s, int i){
                this(i);
                }
```

Either *this* or *super* can be called as the first line from within a constructor, but not both.

As you might guess this will call the other constructor in the current class that takes a single integer parameter. If you use *super() or this()* in a constructor it must be the first method call. As only one or the other can be the first method call, you can not use both *super()* and *this()* in a constructor

Thus the following will cause a compile time error.

```
       MyOver(String s, int i){
                this(i);
                super();//Causes a compile time error
        }
```

Based on the knowledge that constructors are not inherited, it must be obvious that overriding is irrelevant. If you have a class called Base and you create a child that extends it, for the extending class to be overriding the constructor it must have the same name. This would cause a compile time error. Here is an example of this nonsense hierarchy.

```
class Base{}
class Base extends Base{} //Compile time error!
```

Constructors and the class hierarchy

Constructors are always called downward from the top of the hierarchy. You are very likely to get some questions on the exam that involve a class hierarchy with various calls to *this* and *super* and you have to pick what will be the output. Look out for questions where you have a complex hierarchy that is made irrelevant by a constructor that has a call to both *this* and *super* and thus results in a compile time error.

Constructors are called from the base (ancestor) of the hierarchy downwards.

Take the following example

```
class Mammal{
        Mammal(){
                System.out.println("Creating Mammal");
        }
}

public class Human extends Mammal{
public static void main(String argv[]){
        Human h = new Human();
        }
        Human(){
        System.out.println("Creating Human");
       }
}
```

When this code runs the string "Creating Mammal" is output first due to the implicit call to the no-args constructor at the base of the hierarchy.

Questions

Question 1)

Given the following class definition, which of the following methods could be legally placed after the comment with the commented word "//Here"?

```
public class Rid{
        public void amethod(int i, String s){}
        //Here
}
```

1) public void amethod(String s, int i){}
2) public int amethod(int i, String s){}
3) public void amethod(int i, String mystring){}
4) public void Amethod(int i, String s) {}

Question 2)

Given the following class definition which of the following can be legally placed after the comment line //Here ?

```
class Base{
public Base(int i){}
}

public class MyOver extends Base{
public static void main(String arg[]){
                MyOver m = new MyOver(10);
               }
        MyOver(int i){
                super(i);
       }

        MyOver(String s, int i){
                this(i);
                 //Here
        }
}
```

1) MyOver m = new MyOver();
2) super();
3) this("Hello",10);
4) Base b = new Base(10);

Question 3)

Given the following class definition

```
class Mammal{
        Mammal(){
                System.out.println("Mammal");
        }
}

class Dog extends Mammal{
        Dog(){
                System.out.println("Dog");
        }
}

public class Collie extends Dog {
public static void main(String argv[]){
        Collie c = new Collie();

}

        Collie(){
             this("Good Dog");
             System.out.println("Collie");
        }
        Collie(String s){
        System.out.println(s);
       }
}
```

What will be output?

1) Compile time error
2) Mammal, Dog, Good Dog, Collie
3) Good Dog, Collie, Dog, Mammal
4) Good Dog, Collie

Question 4)

Which of the following statements are true?

1) Constructors are not inherited
2) Constructors can be overriden
3) A parental constructor can be invoked using this
4) Any method may contain a call to *this* or *super*

Question 5)

What will happen when you attempt to compile and run the following code?

```
class Base{
        public void amethod(int i, String s){
        System.out.println("Base amethod");
        }
        Base(){
        System.out.println("Base Constructor");
        }

}

public class Child extends Base{
int i;
String Parm="Hello";
public static void main(String argv[]){
        Child c = new Child();
        c.amethod();
}

void amethod(int i, String Parm){
        super.amethod(i,Parm);
        }
public void amethod(){}
}
```

1) Compile time error
2) Error caused by illegal syntax super.amethod(i,Parm)
3) Output of "Base Constructor"
4) Error caused by incorrect parameter names in call to *super.amethod*

Question 6)

What will be output if you attempt to compile and run this code?

```
class Mammal{
Mammal(){
        System.out.println("Four");
        }
public void ears(){
        System.out.println("Two");
        }
}
class Dog extends Mammal{
        Dog(){
```

```
        super.ears();
        System.out.println("Three");
        }
}

public class Scottie extends Dog{
public static void main(String argv[]){
        System.out.println("One");
        Scottie h = new Scottie();
        }
}
```

1) One, Three, Two, Four
2) One, Four, Three, Two
3) One, Four, Two, Three
4) Compile time error

Answers

Answer 1)

1) public void amethod(String s, int i){}
4) public void Amethod(int i, String s) {}

The upper case A on Amethod means that this is a different method.

Answer 2)

4) Base b = new Base(10);

Any call to *this* or super must be the first line in a constructor. As the method already has a call to this, no more can be inserted.

Answer 3)

2) Mammal, Dog, Good Dog, Collie

Answer 4)

1) Constructors are not inherited

Parental constructors are invoked using *super*, not *this*.

Answer 5)

1) Compile time error

This will cause an error saying something like "you cannot override methods to be more private". The base version of *amethod* was specifically marked as public whereas the child had no specifier. OK so this was not a test of your knowlege of constructors overloading but they don't tell you the topic in the exam either. If it were not for the omission of the keyword *public* this code would output "Base constructor", option 3.

Answer 6)

3) One, Four, Two, Three

The classes are created from the root of the hierarchy downwards. Thus One is output first as it comes before the instantiation of the Scottie h. Then the JVM moves to the base of the hierarchy and runs the constructor for the grandparent Mammal. This outputs "Four". Then the constructor for Dog runs. The constructor for Dog calls the *ears* method in Mammal and thus "Two" is output. Finally the constructor for Dog completes and outputs "Three".

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/java/javaOO/methoddecl.html

Richard Baldwin covers this topic at
http://www.Geocities.com/Athens/Acropolis/3797/Java004.htm#polymorphism in general
(This is general stuff on OOP rather than concentrating on "is a" "has a"**)**

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec6.html#obj20

Last updated
12 Jan 2000

Java2 Certification Tutorial

6) Overloading, overriding, runtime type and object orientation

Objective 3)

Write code to construct instances of any concrete class including normal top level classes inner classes static inner classes and anonymous inner classes.

Note on this Objective

Some of this material is covered elsewhere, notably in Objective 4.1

Instantiating a class

Concrete classes are classes that can be instantiated as an object reference (also simply called an object) . Thus an *abstract* class cannot be instantiated and so an object reference cannot be created. Remember that a class that contains any abstract methods the class itself is abstract and cannot be instantiated.

The key to instantiating a class is the use of the *new* keyword. This is typically seen as

```
Button b = new Button();
```

This syntax means that the variable *name* is of the type Button and contains a reference to an instance of the Button. However although the type of the reference is frequently the same as the type of the class being instantiated, it does not have to be. Thus the following is also legal

```
Object b = new Button();
```

This syntax indicates that the type of the reference b is Object rather than Button.

The declaration and instantiation need not occur on the same line. Thus can construct an instance of a class thus.

```
Button b;
b = new Button();
```

Inner classes were added with the release of JDK 1.1. They allow one class to be defined within another.

Inner classes

Inner classes were introduced with the release of JDK 1.1. They allow classes to be defined within other classes, and are sometimes referred to as *nested* classes. They are used extensively in the new 1.1 event handling model. You will almost certainly get questions about nested classes and scoping on the exam.

Here is a trivial example

```
class Nest{
        class NestIn{}
}
```

The output when this code is compiled is two *class* files. One, as you would expect is

```
Nest.class
```

The other is

```
Nest$NestIn.class
```

This illustrates that nesting classes is generally a naming convention rather than a new sort of class file. Inner classes allow you to group classes logically. They also have benefits in scoping benefits where you want to have access to variables.

Nested top level classes

A nested top level class is a static member of an enclosing top level class.

Thus to modify the previous trivial example

```
class Nest{
        static class NestIn{}
}
```

This type of nesting is frequently used simply to group related classes. Because the class is static it does not require an instance of the outer class to exist to instantiate the inner class.

Member classes

I think of a member class as an "ordinary inner class". A member class is analogous to other members of a class, you must instantiate the outer class before you can create an instance of the inner class. Because of the need to be associated with an instance of the outer class Sun introduced new syntax to allow the

simultaneous creation of an instance of the outer class at the same time as the creation of an inner class. This takes the form

```
Outer.Inner i = new Outer().new Inner();
```

To make sense of the new syntax provided for this try to think of the keyword *new* as used in the above example as belonging to the current insistence of *this*,

Thus you could change the line that creates the instance of this to read

```
Inner i = this.new Inner();
```

Because a member class cannot exist without an instance of the outer class, it can have access to the variables of the outer class.

Classes created in methods

A more correct name for this is a local class, but thinking of them as created in methods gives a good flavour of where you are likely to come across them.

local classes can only access final fields or parameters of the enclosing method

A local class is visible only within it's code block or method. Code within a local class definition can only use *final* local variables of the containing block or parameters of the method. You are very likely to get a question on this in the exam.

Anonymous classes

Your first reaction to the idea of an anonymous inner class might be "why would you want to do that and how can you refer to it if it doesn't have a name?"

To answer these questions, consider the following.You might be in the situation of constantly having to think up names for instances of classes where the name was self evident. Thus with event handling the two important things to know are the event being handled and the name of the component that the handler is attached to. Giving a name to the instance of the event handling class does not add much value.

As to the question how can you refer to it if it doesn't have a name, well you can't and if you need to refer to it by name you should not create an anonymous class. The lack of a name has an additional side effect in that you cannot give it any constructors.

Anonymous classes cannot have constructors

Here is an example of the creation of an anonymous inner class

```
class Nest{
public static void main(String argv[]){
        Nest n = new Nest();
                n.mymethod(new anon(){});
       }
        public void mymethod(anon i){}
}
class anon{}
```

Note how the anonymous class is both declared and defined within the parenthesis of the call to *mymethod*.

Quiz

Questions

Question 1)

Which of the following statements are true?

1) A class defined within a method can only access static methods of the enclosing method
2) A class defined within a method can only access final variables of the enclosing method
3) A class defined with a method cannot access any of the fields within the enclosing method
4) A class defined within a method can access any fields accessible by the enclosing method

---

Question 2)

Which of the following statements are true?

1) An anonymous class cannot have any constructors
2) An anonymous class can only be created within the body of a method
3) An anonymous class can only access static fields of the enclosing class
4) The class type of an anonymous class can be retrieved using the getName method

---

Question 3)

Which of the following statements are true?

1) Inner classes cannot be marked private
2) An instance of a top level nested class can be created without an instance of its enclosing class
3) A file containing an outer and an inner class will only produce one .class output file
4) To create an instance of an member class an instance of its enclosing class is required.

Answers

Answer 1)

2) A class defined within a method can only access final variables of the enclosing method

Such a class can access parameters passed to the enclosing method

Answer 2)

1) An anonymous class cannot have any constructors

Answer 3)

2) An instance of a top level nested class can be created without an instance of its enclosing class
4) To create an instance of an member class an instance of its enclosing class is required.

An inner class gets put into its own .class output file, using the format
`Outer$Inner.class`.

A top level nested class is a static class and thus does not require an instance of the enclosing class. A member class is an ordinary non static class and thus an instance of its enclosing class is required.

Other sources on this topic

The Sun Tutorial
http://java.sun.com/docs/books/tutorial/java/more/nested.html

Richard Baldwin
http://www.geocities.com/Athens/7077/Java094.htm

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec6.html#obj21

Last updated
13 Jan 2000

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

Recommended book on this topic

Java Thread Programming by Paul Hyde

I own this Threading book and the O'Reilly Threading book. Paul Hydes book is better. Thread questions come up frequently on the exam and it is a complex topic. I recommend you buy this book (and if you buy it from these links I'll get a small commission).

Buy from Amazon.com or from Amazon.co.uk

---

7) Threads

Objective 1)

Write code to define, instantiate and start new threads using both java.lang.Thread and java.lang.Runnable

What is a thread?

Threads are lightweight processes that appear to run in parallel with your main program. Unlike a process a thread shares memory and data with the rest of the program. The word thread is a contraction of "thread of execution", you might like to imagine a rope from which you have frayed the end and taken one thread. It is still part of the main rope, but it can be separated from the main and manipulated on its own. An example of where threads can be useful is in printing. When you click on a print button you probably don't want the main program to stop responding until printing has finished. What would be nice is that the printing process started running "in the background" and allowed you to continue using the main portion of the program.

It would also be useful if the main program would respond if the printing thread encountered a problem. A common example used to illustrate threads is to create a GUI application that launches a bouncing ball every time a button is clicked. Unlike most language threading is embedded at the heart of the Java language, much of it at the level of the ultimate ancestor class called Object.

The two ways of creating a thread

Of the two methods of creating a new thread the use of Runnable is probably more common, but you must know about both for the purpose of the exam. Here is an example of a class created with the Runnable interface.

```
class MyClass implements Runnable{
 public void run(){//Blank Body}
}
```

Creating the thread of execution.

```
MyClass mc = new MyClass();
```

Any class that implements an interface must create a method to match all of the methods in the interface. The methods need not do anything sensible, i.e. they may have blank bodies, but they must be there. Thus I include the method *run* even in this little example, because you must include a run method if you implement Runnable. Not including a run method will cause a compile time error.

To do anything useful when you create a thread of execution from a class you would, of course need to put something where I have put

```
//Blank Body.
```

The other method for creating a thread is to create a class that is descended from Thread. This is easy to do but it means you cannot inherit from any other class, as Java only supports single inheritance. Thus if you are creating a Button you cannot add threading via this method because a Button inherits from the AWT Button class and that uses your one shot at inheritance. There is some debate as to which way of creating a thread is more truly object oriented, but you do need to go into this for the purpose of the exam.

Instantiating and starting a Thread

Although the code that runs in your thread is in a method called run, you do not call this method directly, instead you call the start method of the thread class. The Runnable interface does not contain a start method, so to get at this and the other useful methods for threads (sleep, suspend etc etc), you pass your class with the Runnable interface as the constructor to an instance of the Thread class.

Thus to cause the thread to execute from a class that implements Runnable you would call the following

```
MyClass mc = new MyClass();
Thread t = new Thread(mc);
t.start();
```

Again note that was a call to *start*, not a call to *run*, even though it is the code in the *run* method in your class that actually executes.

Key Concept

Although it is the *run* method code that executes, a thread is actually started via the *start* method

If you create your class as a sub class of Thread you can simply call the start method. The drawback of sub classing the Thread class is that due to only supporting single inheritance you cannot inherit the functionality of any other class.

Quiz

Questions

Question 1)

What will happen when you attempt to compile and run this code?

```
public class Runt implements Runnable{
public static void main(String argv[]){
        Runt r = new Runt();
        Thread t = new Thread(r);
        t.start();
        }
        public void start(){
        for(int i=0;i<100;i++)
                System.out.println(i);
        }
}
```

1) Compilation and output of count from 0 to 99
2) Compilation and no output
3) Compile time error: class Runt is an abstract class. It can't be instantiated.
4) Compile time error, method *start* cannot be called directly

---

Question 2)

Which of the following statements are true?

1) Directly sub classing Thread gives you access to more functionality of the Java threading capability than using the Runnable interface
2) Using the Runnable interface means you do not have to create an instance of the Thread class and can call *run* directly
3) Both using the Runnable interface and subclassing of Thread require calling *start* to begin execution of a Thread
4) The Runnable interface requires only one method to be implemented, this is called *run*

Question 3)

What will happen when you attempt to compile and run the following code?

```
public class Runt extends Thread{
public static void main(String argv[]){
        Runt r = new Runt();
        r.run();
        }

        public void run(){
        for(int i=0;i<100;i++)
                System.out.println(i);
        }
}
```

1) Compilation and output of count from 0 to 99
2) Compilation and no output
3) Compile time error: class Runt is an abstract class. It can't be instantiated.
4) Compile time error, method *start* has not been defined

Question 4)

Which of the following statements are true?

1) To implement threading in a program you must import the class java.io.Thread
2) The code that actually runs when you start a thread is placed in the *run* method
3) Threads may share data between one another
4) To start a Thread executing you call the *start* method and not the *run* method

Answers

Answer 1)

3) Compile time error: class Runt is an abstract class. It can't be instantiated.

The class implements Runnable but does not define the run method.

Answer 2)

3) Both using the Runnable interface and subclassing of Thread require calling *start* to begin execution of a Thread
4) The Runnable interface requires only one method to be implemented, this is called *run*

Answer 3)

1) Compilation and output of count from 0 to 99

However, note that this code does not start the execution of the Thread and the run method should not be called in this way.

Answer 4)

2) The code that actually runs when you start a thread is placed in the *run* method
3) Threads may share data between one another
4) To start a Thread executing you call the *start* method and not the *run* method

You do not need to import any classes as Threading is an integral part of the Java language

---

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/threads/customizing.html

Richard Baldwin Covers this topic at
http://www.Geocities.com/Athens/Acropolis/3797/Java058.htm#two ways to thread

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec7.html#obj22

Thread part of of Elliot Rusty Harolds Tutorial Course
http://www.ibiblio.org/javafaq/course/week11/index.html

Last updated
9 November 2000

Java2 Certification Tutorial

7)Threads

Objective 2)

Recognize conditions that might prevent a thread from executing.

Comment on the objective

The expression "prevent a thread from executing" is slightly ambiguous, does it mean a thread that has been deliberately paused, or does it also include threads that have died?. A thread that is prevented from executing is said to be blocked.

Reasons a thread may be blocked

A thread may be blocked because

- 1) It has been put to sleep for a set amount of time
- 2) It is suspended with a call to suspend() and will be blocked until a resume() message
- 3) The thread is suspended by call to wait(), and will become runnable on a *notify* or *notifyAll* message.

For the purposes of the exam *sleep*(), and *wait/notify* are probably the most important of the situations where a thread can be blocked.

The *sleep* method is static and pauses execution for a set number of milliseconds. There is a version that is supposed to pause for a set number of nanoseconds, though I find it hard to believe many people will

work on a machine or Java implementation that will work to that level of accuracy. Here is an example of putting a Thread to sleep, note how the sleep method throws InterruptedException. The thread

```
public class TSleep extends Thread{
public static void main(String argv[]){
       TSleep t = new TSleep();
       t.start();
       }
      public void run(){
          try{
             while(true){
                  this.sleep(1000);
                  System.out.println("looping while");
                 }
            }catch(InterruptedException ie){}
       }
}
```

With the release of the Java2 platform the Thread methods *stop*, *suspend* and *resume* have been deprecated (no longer recommended for use, and will produce a warning at compile time). The JDK notes have the contain the following notice

//Quote

Deprecated. This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.

//End Quote

A generally reliable source (Kathy Kozel) has indicated that you may need to be aware of this for the purpose of the exam. I will assume that you do not need to know how to actually use them.

Thread blocking via the *wait/notify* protocol is covered in the next topic 7.3.

Quiz

Questions

Question 1)

What will happen when you attempt to compile and run this code?

```
public class TGo implements Runnable{
public static void main(String argv[]){
        TGo tg = new TGo();
        Thread t = new Thread(tg);
        t.start();
        }
        public void run(){
                while(true){
                        Thread.currentThread().sleep(1000);
                        System.out.println("looping while");
                       }
       }
}
```

1) Compilation and no output
2) Compilation and repeated output of "looping while"
3) Compilation and single output of "looping while"
4) Compile time error

Question 2)

Which of the following are recommended ways a Thread may be blocked?

1) sleep()
2) wait/notify
3) suspend
4) pause

Question 3)

Which of the following statements are true?

1) The sleep method takes parameters of the Thread and the number of seconds it should sleep
2) The sleep method takes a single parameter that indicates the number of seconds it should sleep
3) The sleep method takes a single parameter that indicates the number of milliseconds it should sleep
4) The sleep method is a static member of the Thread class

Answers

Answer 1)

4) Compile time error
The *sleep* method throws InterruptedException and thus this code will not compile until the while loop is surrounded by a *try/catch* block

Answer 2)

1) sleep()
2) wait/notify

For the Java2 platform the *suspend* method has been deprecated and thus is valid but not recommended

Answer 3)

3) The *sleep* method takes a single parameter that indicates the number of milliseconds it should sleep
4) *sleep* is a static method of the Thread class

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/threads/waitAndNotify.html

Commentry on deprecated Thread methods at
http://java.sun.com/docs/books/tutorial/post1.0/preview/threads.html

Richard Baldwin Covers this topic at
http://www.geocities.com/Athens/Acropolis/3797/Java058.htm#the notify() and wait() methods

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec7.html#obj23

Last updated
19 Jan 2000

Java2 Certification Tutorial

7)Threads

Objective 3)

Write code using synchronized wait *notify* and *notifyAll* to protect against concurrent access problems and to communicate between threads. Define the interaction between threads and between threads and object locks when executing synchronized wait *notify* or *notifyAll*.

Why do you need the wait/notify protocol?

One way to think of the *wait/notify* protocol is to imagine an item of data such as an integer variable as if it were a field in a database. If you do not have some locking mechanism in the database you stand a chance of corruption to the data.

Thus one user might retrieve the data and perform a calculation and write back the data. If in the meantime someone else has retrieved the data, performed the calculation and written it back, the second users calculations will be lost when the first person writes back to the database. In the way that a database has to handle updates at unpredictable times, so a multi threaded program has to cater for this possibility.

The *synchronized* keyword

The *synchronized* keyword can be used to mark a statement or block of code so that only one thread may execute an instance of the code at a time. Entry to the code is protected by a monitor lock around it. This process is implemented by a system of locks. You may also see the words monitor, or mutex (mutually exclusive lock) used. A lock is assigned to the object and ensures only one thread at a time can access the

code. Thus when a thread starts to execute a synchronized block it grabs the lock on it. Any other thread will not be able to execute the code until the first thread has finished and released the lock. Note that the lock is based on the object and not on the method.

For a method the *synchronized* keyword is placed before the method thus

```
synchronized void amethod() { /* method body */}
```

For a block of code the synchronized keyword comes before opening and closing brackets thus.

```
synchronized (ObjectReference) { /* Block body */ }
```

The value in parentheses indicates the object or class whose monitor the code needs to obtain. It is generally more common to synchronize the whole method rather than a block of code.

When a synchronized block is executed, its object is locked and it cannot be called by any other code until the lock is freed.

```
synchronized void first();
synchronized void second();
```

There is more to obtaining the benefits of synchronization than placing the keyword synchronized before a block of code. It must be used in conjunction with code that manages the lock on the synchronized code .

wait/notify

In addition to having a lock that can be grabbed and released, each object has a system that allows it to pause or *wait* whilst another thread takes over the lock. This allows Threads to communicate the condition of readiness to execute. Because of the single inheritance nature of Java, every object is a child of the great grand ancestor *Object* class from which it gets this Thread communication capability.

wait and _notify_ should be placed within synchronized code to ensure that the current code owns the monitor

A call to *wait* from within synchronized code causes the thread to give up its lock and go to sleep. This normally happens to allow another thread to obtain the lock and continue some processing. The *wait* method is meaningless without the use of *notify* or *notifyAll* which allows code that is waiting to be notified that it can wake up and continue executing. A typical example of using the *wait/notify* protocol to allow communication between Threads appears to involve apparently endless loops such as

```
//producing code
while(true){
try{
        wait();
        }catch (InterruptedException e) {}
}
```

```
//some producing action goes here
notifyAll();
```

As *true* is notorious for staying true this, code looks at first glance like it will just loop forever. The wait method however effectively means *give up the lock* on the object and wait until the notify or *notifyAll* method tells you to wake up.

Thread scheduling is implementation dependent and cannot be relied on to act in the same way on every JVM

Unlike most aspects of Java, Threading does not act the same on different platforms. Two areas of difference are Thread scheduling and Thread priorities. The two approaches to scheduling are

- Preemptive
- Time slicing

In a pre-emptive system one program can "pre-empt" another to get its share of CPU time. In a time sliced system each thread gets a "slice" of the CPU time and then gets moved to the ready state. This ensures against a single thread getting all of the CPU time. The downside is that you cannot be certain how long a Thread might execute or even when it will be running. Although Java defines priorities for threads from the lowest at 1 to the highest at 10, some platforms will accurately recognise these priorities whereas others will not.

The *notify* method will wake up one thread waiting to reacquire the monitor for the object. You cannot be certain which thread gets woken. If you have only one waiting thread then you do not have a problem. If you have multiple waiting threads then it will probably the thread that has been waiting the longest that will wake up. However you cannot be certain, and the priorities of the threads will influence the result. As a result you are generally advised to use *notifyAll* instead of notify, and not to make assumptions about scheduling or priorities. Of course this is not always possible and you may have to try to test your code on as many platforms as possible.

Questions

Question 1)

Which of the following keywords indicates a thread is releasing its Object lock?

1) release
2) wait
3) continue
4) notifyAll

Question 2)

Which best describes the *synchronized* keyword?

1) Allows more than one Thread to access a method simultaneously
2) Allows more than one Thread to obtain the Object lock on a reference
3) Gives the notify/*notifyAll* keywords exclusive access to the monitor
4) Means only one thread at a time can access a method or block of code

---

Question 3)

What will happen when you attempt to compile and run the following code?

```
public class WaNot{
int i=0;
public static void main(String argv[]){
        WaNot w = new WaNot();
        w.amethod();
        }
        public void amethod(){
        while(true){
                try{
                        wait();
                    }catch (InterruptedException e) {}
                i++;
            }//End of while

        }//End of amethod
}//End of class
```

1)Compile time error, no matching notify within the method
2)Compile and run but an infinite looping of the while method

```
3)Compilation and run
4)Runtime Exception "IllegalMonitorStatException"
```

Question 4)

How can you specify which thread is notified with the wait/notify protocol?

1) Pass the object reference as a parameter to the *notify* method
2) Pass the method name as a parameter to the *notify* method
3) Use the *notifyAll* method and pass the object reference as a parameter
4) None of the above

Question 5)

Which of the following are true

1) Java uses a time-slicing scheduling system for determining which Thread will execute
2) Java uses a pre-emptive, co-operative system for determining which Thread will execute
3) Java scheduling is platform dependent and may vary from one implementation to another
4) You can set the priority of a Thread in code

Answers

Answer 1)

2) wait

Answer 2)

4) Means only one thread at a time can access a method or block of code

Answer 3)

4) Runtime Exception "IllegalMonitorStateException"

The wait/notify protocol can only be used within code that is synchronized. In this case calling code does not have a lock on the object and will thus cause an Exception at runtime.

Answer 4)

4) None of the above.

The wait/notify protocol does not offer a method of specifying which thread will be notified.

Answer 5)

3) Java scheduling is platform dependent and may vary from one implementation to another
4) You can set the priority of a Thread in code

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/threads/waitAndNotify.html

Richard Baldwin Covers this topic at
http://www.geocities.com/Athens/Acropolis/3797/Java058.htm

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec7.html#obj24

Bruce Eckel Thinking in Java
Chapter 14

Last updated
20 Jan 2000

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

8) The java.awt package - Layout

Objective 1)

Write code using component container and layout manager classes of the *java.awt* package to present a GUI with specified appearance and resize the behavior and distinguish the responsibilities of layout managers from those of containers.

Note on this Objective

Although it does not mention it specifically this objective involves a new objective compared with the 1.1 exam. This is the GridBaglayout It makes sense to cover this as it is a very useful LayoutManager, but because of its power it can take some learning. Peter van der Linden in Just Java and Beyond 3rd Edition describes it as excessively complicated and doesn't recommend it. Core Java merely says "using grid bag layouts can be incredibly complex". Whilst it is complex to use by hand, the various GUI tools such as VisualCafe, Visual Age, JBuilder etc etc make it easier to understand.

Comparing Visual Basic and Java layout

Java uses a different philosophy to layout compared with tools such as Visual Basic or Delphi (if philosophy is not too grand an expression for laying out a program). Most design tools use an XY pixel based approach to placing a component. Thus in Visual Basic you can pick up a text box from the component palette and drop it at a location on a form, and its location is set. By contrast Java uses Layout classes to control where a component is placed according to the current screen.

Part of the reason for this is the cross platform nature of Java. A Java applet may display on anything from a palm top computer to a 19 inch Sun Workstation. I have tried writing Visual Basic applications

that take account of more than one screen resolution and it is not a trivial activity. Be warned, if you have a background in other RAD tools you may find the Layout Manager approach a little weird at first.

The LayoutManager philosophy

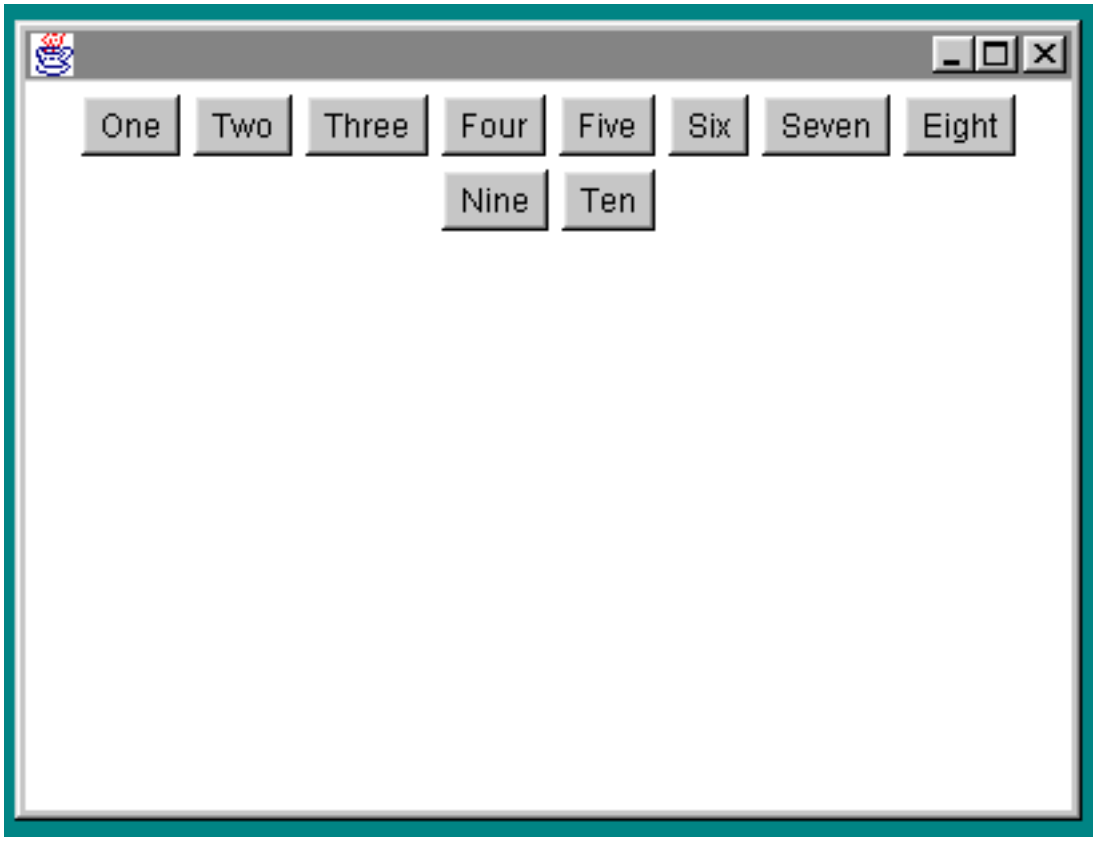
The FlowLayout manager is a good place to start as it is the default for Applets. The FlowLayout manager simply places components on a background one after the other from left to right. If it runs out of space to the right it wraps around the components to the next line.

The following code creates a very simple application and adds a series of buttons

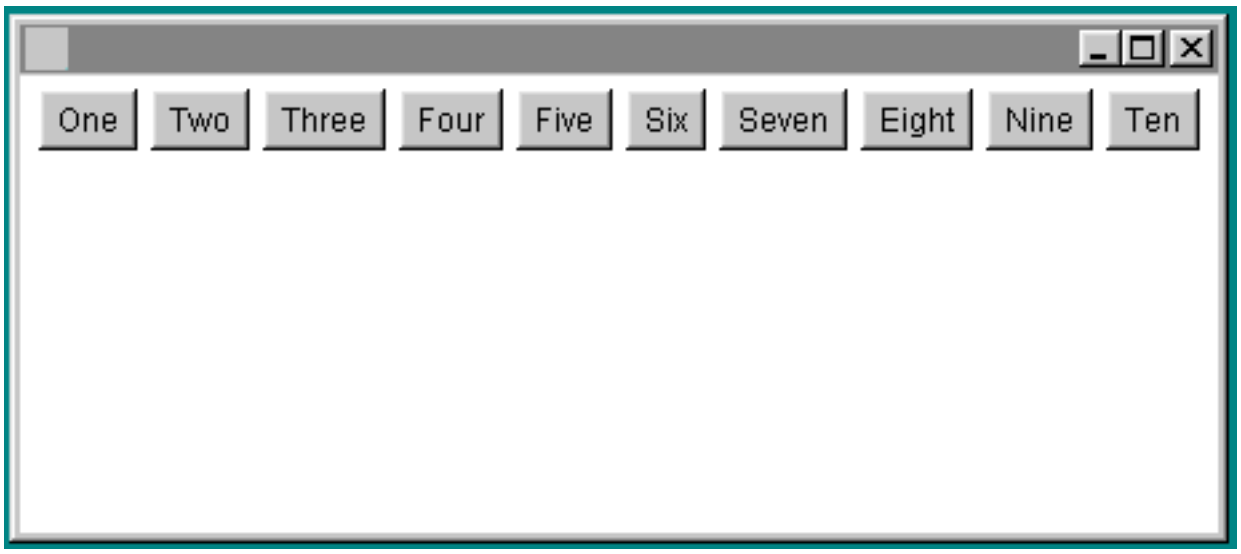
```
import java.awt.*;
public class FlowAp extends Frame{
public static void main(String argv[]){
            FlowAp fa=new FlowAp();
            //Change from BorderLayout default
            fa.setLayout(new FlowLayout());
            fa.setSize(400,300);
            fa.setVisible(true);
    }
    FlowAp(){
                add(new Button("One"));
                add(new Button("Two"));
                add(new Button("Three"));
                add(new Button("Four"));
                add(new Button("Five"));
                add(new Button("Six"));
                add(new Button("Seven"));
                add(new Button("Eight"));
                add(new Button("Nine"));
                add(new Button("Ten"));
        }//End of constructor
}//End of Application
```

The following image is the default appearance when you fire it up from the command line.

Default appearance of application FlowAp



FlowAp after changing width



Bear in mind that both images are the display for exactly the same Java code. The only thing that has changed is the width. The FlowLayout manager automatically changes the layout of the components when the Frame is re-sized. If you were to make the Frame very small the FlowLayout manager would change the layout so that the buttons were wrapped around in several rows.

When you first come across this approach to the management of components it may seem a little arbitrary. Some of the GUI building tools such as Symantec Visual Cafe or Borland/Inprise JBuilder offer ways of specifically placing components. For the purposes of the exam though you must become

familiar with the Layout Manager approach to GUI creation.

Layout Managers you need to know for the exam

For the exam you need to know the following layout managers

- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout

(note: the first editions of the Roberts, Heller and Ernest book on certification say you do not need to know about the GridBagLayout, but this has been corrected in the online errata, see my FAQ)

Responsibilities of the layout manager vs containers

Containers and Layout Managers work in partnership. The LayoutManager generally controls where a component is positioned. A Container will control the default font for its components. A component may be specifically assigned a font for itself. Questions on this seemed to come up regularly in the 1.1 exam. You were given a text description of a Component/Container setup and then asked what background color or font a Button or label would display.

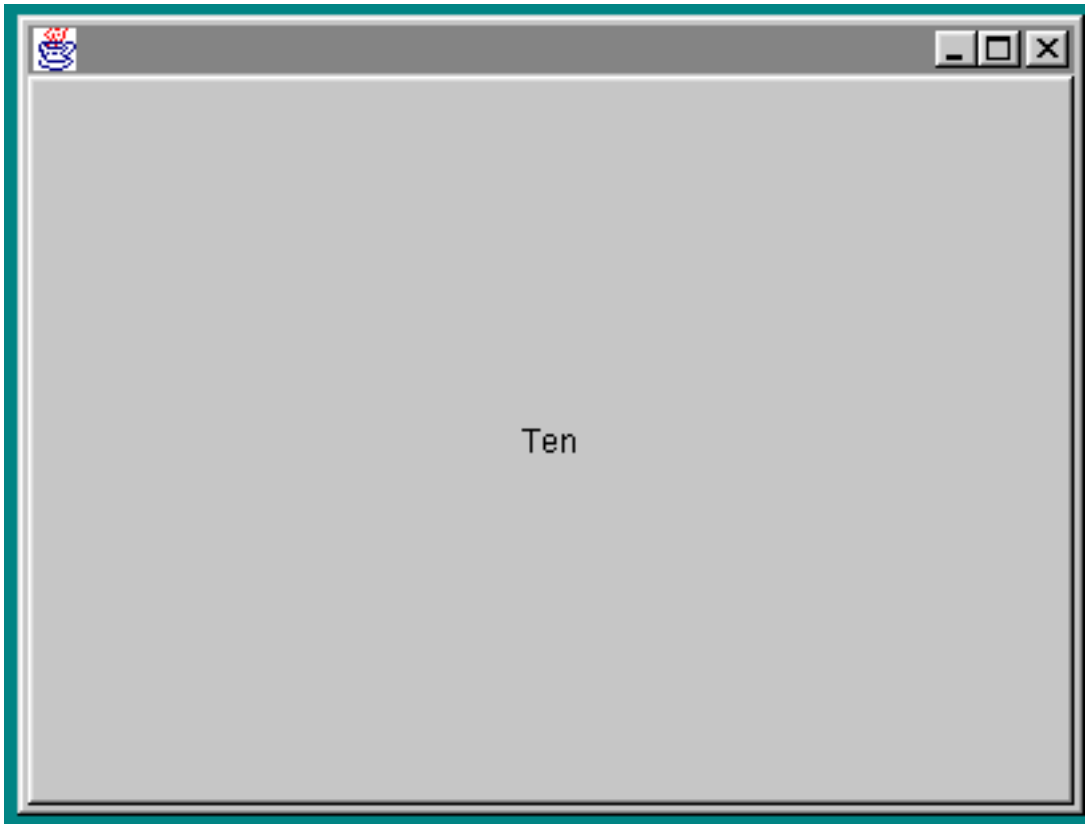
Oddities of the BorderLayout manager

If you add multiple components to a Container that uses the BorderLayout but do not pass a Constraint parameter (North, South, etc), you may get unexpected results. Here is a sample that illustrates this.

```
import java.awt.*;
public class FlowAp extends Frame{
public static void main(String argv[]){
    FlowAp fa=new FlowAp();
    // fa.setLayout(new FlowLayout());
    fa.setSize(400,300);
    fa.setVisible(true);
}

FlowAp(){
        add(new Button("One"));
        add(new Button("Two"));
        add(new Button("Three"));
        add(new Button("Four"));
        add(new Button("Five"));
        add(new Button("Six"));
        add(new Button("Seven"));
        add(new Button("Eight"));
        add(new Button("Nine"));
        add(new Button("Ten"));
```

```
    }//End of constructor
}//End of Appl
```

Using the default BorderLayout



The reason you get this unexpected big button in the center is that the BorderLayout uses a set of coordinates when arranging components. It divides its surface area up into

- North
- South
- East
- West
- Center

You might guess that the default when laying out components would be for them to be placed clockwise around the points of the compass or some such arrangement. Instead the designers decided to make the default the center of the layout area. Thus in this example every button has been laid out on the previous button, taking up the entire available area. As a result it appears that you only have one button, the last one added.

Because the BorderLayout only divides the area up into the five mentioned coordinates it is not the most useful of Layout Managers. However you need to be aware of it for the exam and you need to be aware of the way it defaults to placing all components in the center.

The GridLayout Manager

The *GridLayout* manager does approximately what you might expect. It divides the surface area up into a grid and when you add components it places them one after the other from left to right, top to bottom. Unlike the *BorderLayout* and *FlowLayout* it ignores any preferred size of the component. For example the preferred size of a button will be wide enough to show its text. The *FlowLayout* manager attempts to ensure that a button is this preferred size. The *GridLayout* has a more bondage and discipline approach. The only thing it cares about is making sure the component fits into the grid.

The following code lays out a set of buttons within a Frame using a *GridLayout* that has been set up to have 2 rows and 5 columns.

```
import java.awt.*;
public class GridAp extends Frame{
public static void main(String argv[]){
        GridAp fa=new GridAp();
        //Setup GridLayout with 2 rows and 5 columns
        fa.setLayout(new GridLayout(2,5));
        fa.setSize(400,300);
        fa.setVisible(true);
    }
GridAp(){
        add(new Button("One"));
        add(new Button("Two"));
        add(new Button("Three"));
        add(new Button("Four"));
        add(new Button("Five"));
        add(new Button("Six"));
        add(new Button("Seven"));
        add(new Button("Eight"));
        add(new Button("Nine"));
        add(new Button("Ten"));
        }//End of constructor
}//End of Application
```

GridLayout sample

Note how the buttons are enlarged to fill all of the available space.

GridBagLayout

Peter van der Linden in Just Java and Beyond 3rd Edition describes the GridBagLayout manager as *"excessively complicated "* and doesn't recommend it. Core Java merely says "using grid bag layouts can be incredibly complex". Whilst it is complex to use by hand, the various GUI tools such as VisualCafe, Visual Age, JBuilder etc etc make it easier to use, if not understand. Thus JBuilder will happily modify the *add* statement to include the following details for the GridBagLayout class.

```
add(pAps,new GridBagConstraints2(1, GridBagConstraints.RELATIVE,
GridBagConstraints.RELATIVE, 3, 0.0, 0.0,GridBagConstraints.CENTER,
GridBagConstraints.NONE, new Insets(0, 0, 0, 0), -3, 45));
```

```
But when you create your code by hand it does not need to look as
complex as this.
```

Feedback from people who have taken the exam indicates that the questions on the GridBagLayout are not very in-depth and a basic understanding of the various fields of the GridBagConstraints class may well be adequate.

My favorite Java Tool is Borland/Inprise JBuilder which has its own Layout Manager called the XYLayout manager. This seems to be easier to use than the GridBagLayout, but if you are writing for the net it would require users to download that additional class, causing additional overhead.

GridBagLayout is a little like the GridLayout except that different cell rows can have different heights,

and columns can have different widths. The Java2 Docs come with a demonstration applet that shows what can be done with the GridBagLayout manager.

In the exam this is a prime moment to take advantage of the scrap paper to write out a grid and consider the effect of each cell. One of the problems with the GridBagLayout is that instead of being based strictly on the underlying grid, Java tries to guess the cells from the information given. The GridBagLayout manager uses a helper class GridBagConstraints which has a set of member variables that can be set to affect the appearance of each component. The fields that you modify on the GridBagConstraints class act as "suggestions" as to where the components will go. An instance of GridBagConstraints is passed as a parameter with the add method, in the form

- add(component, GridBagConstraint);

GridBagConstraints goes against the general convention in Java in that you might expect its attributes to be configured with

```
setFooParam()
```

methods, where FooParam might be WeightX/Y or Padding between components.

Instead it takes the form

```
GridBagLayout gbl=new GridBagLayout();
gbl.weightx=100;
```

If you use the GridBagLayout without the GridBagConstraints class it acts a little like a FlowLayout, simply dropping the components onto the background one by one.

I have created a simple demonstration applet with source that shows how nothing much happens unless you play with the GridBagConstraints class..

http://www.software.u-net.com/Applets/GridBagDemo/GridBagTest.htm

The GridBagLayout acts a little more like the GridLayout if you use the GridBagConstraints class and use the gridx and gridy fields to assign a position in a "virtual" grid to each component as you add it. This applet demonstrates this possibility. This is still a little dull and very like the other layout managers. Things start to get much more interesting when you start to modify other fields of the GridBagConstraints class to modify the appearance of different components within this "virtual" grid.

Remember that although you need to understand this for the purposes of the exam, it might be easier when programming in the real world to use a combination of container controls added with other layout managers. An example of when this is not an option is when you need to dynamically re-size components. This is a situation where GUI builders such as Visual Cafe or JBuilder are not much help and an understanding of the GridBagLayout may be essential.

I have created a demonstration applet that shows the affect of dynamically changing the padding parameters for a single button in a group of buttons set out with a GridbagLayout manager

The fields for the GridBagConstraints class are

- gridx gridy
- gridwidth and gridheight
- fill
- ipadx and ipady
- insets
- anchor
- weightx and weighty

Whilst browsing Bill Brogdens excellent Java2 Exam Cram Book I found a pointer to a comprehensive demo of the GridBagLayout at

http://www.austria.eu.net/java/programs/applets/missgable/index.htm

Using *gridx* and *gridy* to suggest component placing

For this example you are doing some basic code to design a appointment calendar program. It will show times down the left hand side and appointment details down the right. The time units will be in half hour chunks.

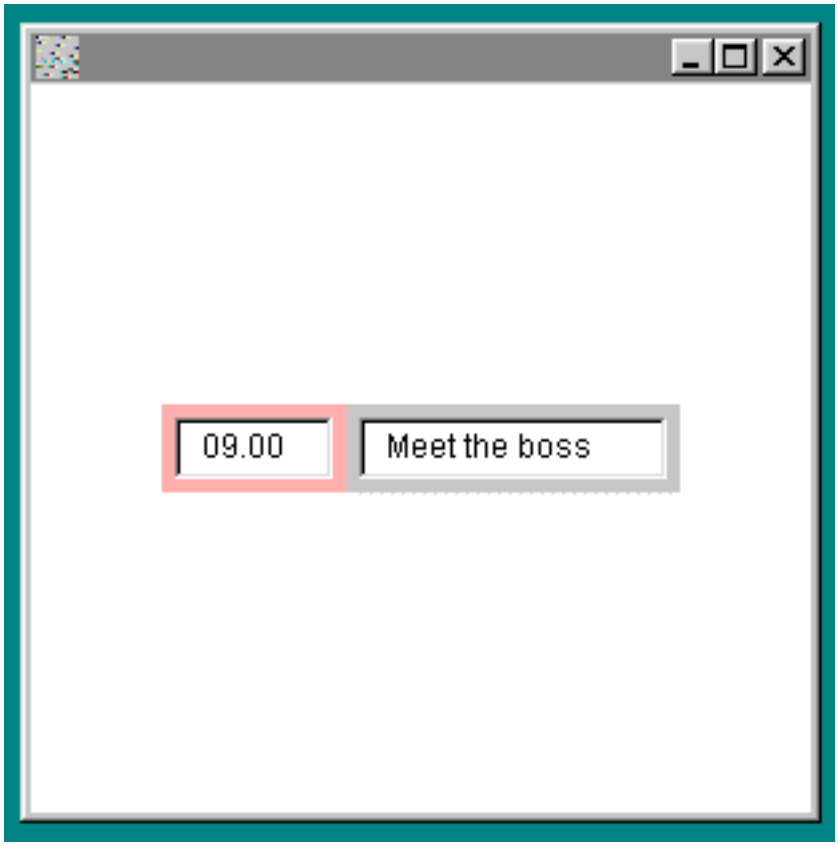
Because an appointment may cover more than one time unit, ie may last an hour and a half you need to be able to dynamically change the height of an appointment to cover more than one half hour time unit. Because of this requirement to have a varying height for the appointments, a GridLayout is not suitable.

You will be placing panels on the Frame as containers. The first step is to ensure that each panel sits side by side on the main Frame of the Application.

```
import java.awt.*;
import java.awt.event.*;
public class GBCal extends Frame{
    Panel pTimes=new Panel();
    Panel pAps=new Panel();
    TextField txTimes=new TextField("09.00");
    TextField txAps=new TextField("Meet the boss");
    GridBagLayout gbl=new GridBagLayout();
    GridBagConstraints gbc=new GridBagConstraints();
    public static void main(String argv[]){
    GBCal gbc=new GBCal();
    gbc.setLayout(new FlowLayout());
}

public GBCal() {
    pTimes.add(txTimes);
    pAps.add(txAps);
    setLayout(gbl);
    gbc.gridx=0;
    gbc.gridy=0;
```

```
    pTimes.setBackground(Color.pink);
    add(pTimes,gbc);
    gbc.gridx=1;
    gbc.gridy=0;
    pAps.setBackground(Color.lightGray);
    add(pAps,gbc);
    setSize(300,300);
    setVisible(true);
    }
}
```

The output will appear as follows



Note how the GridBagLayout and the GridBagConstraints classes work together. The GridBagConstraints instance gbc gets re-used for each time a component is added. At no point do you specifically state the number of rows and columns for the Grid as the GridBagLayout class deduces it from the gridx and gridy fields of the GridBagConstraints instance.

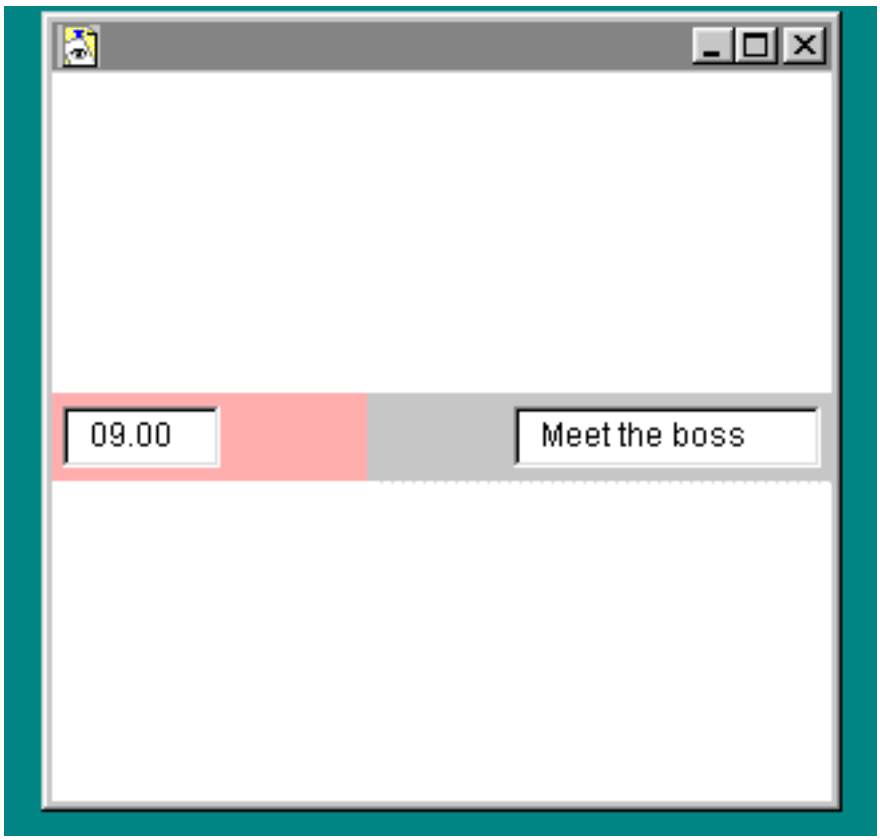
ipadx and *ipady* to control the internal padding of components

The code has set the background color so you can see the extent of the panel rather than simply the width of the text fields. This is fine but now you want the fields to stretch all the way from left to right of the

main application Frame. This can be performed by modifying the *ipadx* field of the GridBagConstraints class. This is peformed by setting

```
gbc.ipadx=30;
For the times and
gbc.ipadx=100;
For the appointments
```

The result is as follows



Components within a panel with the GridBaglayout

For the next step I want to give each panel its own *GridBagLayout* manager and add additional time slots and appointments. For the purpose of this example I will add just one more time slot and simply stretch the single appointment to cover the time slots between 9.00 and 9.30.

To do this I will create a new instance of GridBagLayout called *gbBut* and use it to set up the grid for the *pTimes* panel to place the time slot fields one on top of the other vertically.

The code that performs this is

```
//Control the Times panel with a GridBagLayout
pTimes.setLayout(gbBut);
gbc.gridx=0;
gbc.gridy=0;
pTimes.add(txTimes9,gbc);
gbc.gridx=0;
```

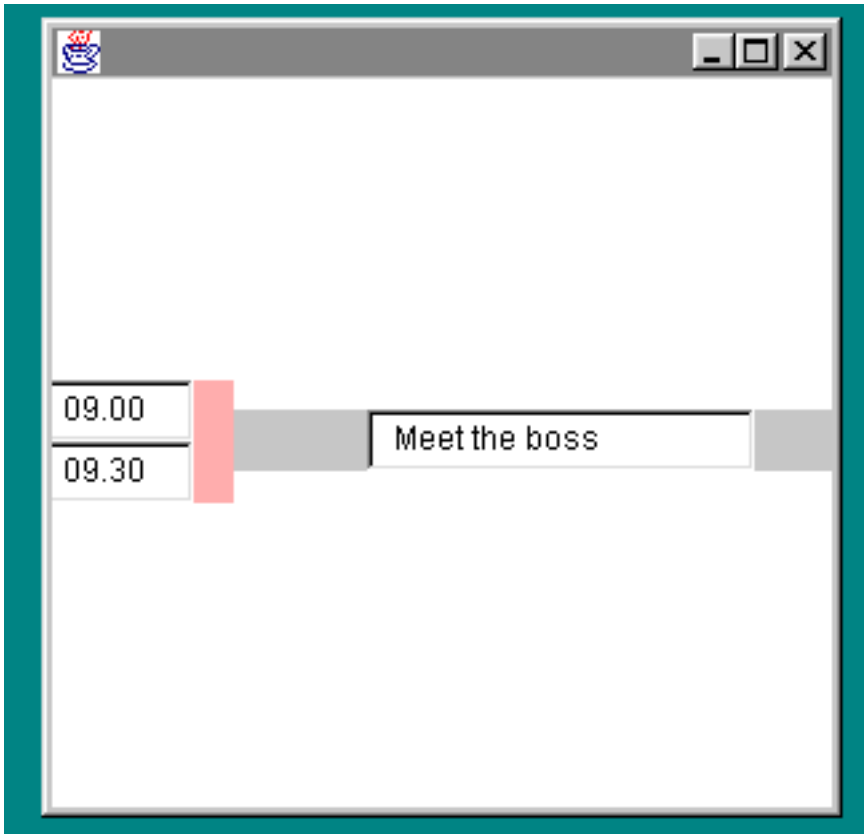
```
gbc.gridy=1;
pTimes.add(txTimes930,gbc);
```

Here is the complete code to the revised program

```
import java.awt.*;
import java.awt.event.*;
public class GBCal extends Frame{
Panel pTimes=new Panel();
Panel pAps=new Panel();
TextField txTimes9=new TextField("09.00");
TextField txTimes930=new TextField("09.30");
TextField txAps=new TextField("Meet the boss");
GridBagLayout gbl=new GridBagLayout();
GridBagLayout gbBut=new GridBagLayout();
GridBagConstraints gbc=new GridBagConstraints();
public static void main(String argv[]){
    GBCal gbc=new GBCal();
    gbc.setLayout(new FlowLayout());
    }
public GBCal() {
    setLayout(gbl);
    //Control the Times panel with
    //a GridBagLayout
    pTimes.setLayout(gbBut);
    gbc.gridx=0;
    gbc.gridy=0;
    pTimes.add(txTimes9,gbc);
    gbc.gridx=0;
    gbc.gridy=1;
    pTimes.add(txTimes930,gbc);
    pTimes.setBackground(Color.pink);
    //Re-using gbc for the main panel layout
    gbc.gridx=0;
    gbc.gridy=0;
    gbc.ipadx=30;
    add(pTimes,gbc);
    pAps.setLayout(gbBut);
    gbc.gridx=0;
    gbc.gridy=1;
    pAps.add(txAps,gbc);
    gbc.gridx=1;
    gbc.gridy=0;
    gbc.ipadx=100;
    pAps.setBackground(Color.lightGray);
    add(pAps,gbc);
    setSize(300,300);
```

```
    setVisible(true);
    }
}//End of class
```

The resulting output is as followed



This has worked up to a point. We now have the two time slots, but unfortunatly the one appointment has defaulted to the centre of the appointments field and it is only one row thick. What I want is for it to be anchored at the top of the appointments area and to stretch to cover both time slots.

Anchoring components within the grid

f a component does not fill the whole area, you can specify where in the area you want it using the *anchor* field of the GridBagConstraints class. The possible values are

```
GridBagconstraints.CENTER
GridBagconstraints.NORTH
GridBagconstraints.NORTHEAST
```

etc etc

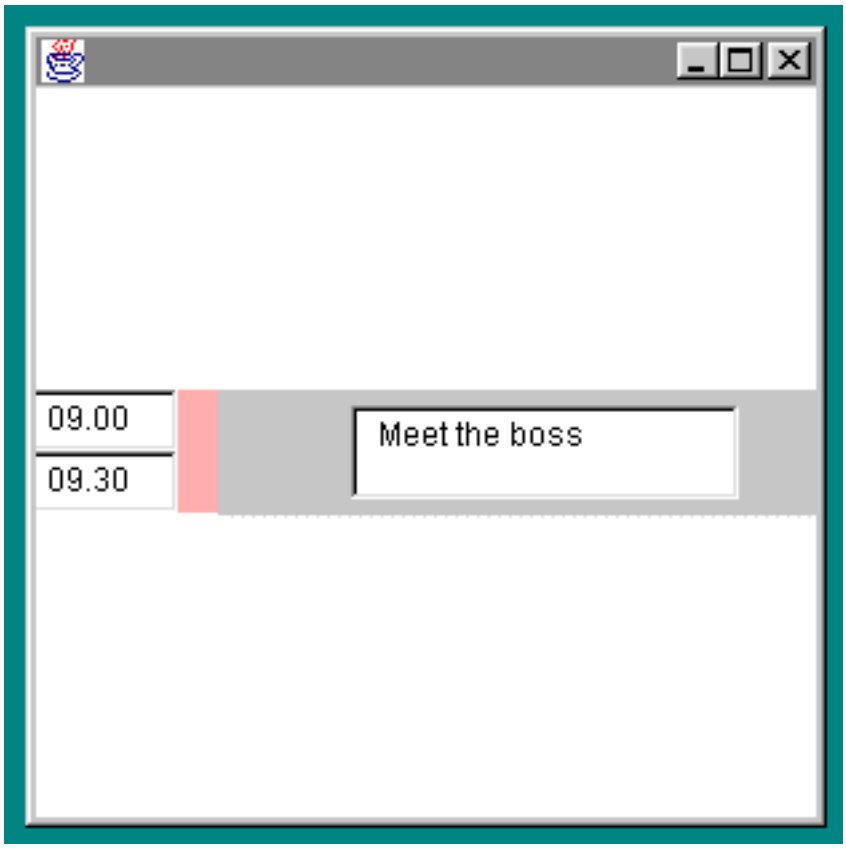
In this case I want to position the fields at the top (North) of the containing panels. I have increased the depth of the Appointment field by increasing the *ipady* value for the address field.

Here is the code to do this.

```
import java.awt.*;
import java.awt.event.*;
public class GBCal extends Frame{
```

```
Panel pTimes=new Panel();
Panel pAps=new Panel();
TextField txTimes9=new TextField("09.00");
TextField txTimes930=new TextField("09.30");
TextField txAps=new TextField("Meet the boss");
GridBagLayout gbl=new GridBagLayout();
GridBagLayout gbBut=new GridBagLayout();
GridBagConstraints gbc=new GridBagConstraints();
public static void main(String argv[]){
        GBCal gbc=new GBCal();
        gbc.setLayout(new FlowLayout());
    }

public GBCal() {
    setLayout(gbl);
    //Control the Times panel with a GridBagLayout
    pTimes.setLayout(gbBut);
    //Ensure the componants sit at
   //the top of the containers
    gbc.anchor=GridBagConstraints.NORTH;
    gbc.gridx=0;
    gbc.gridy=0;
    pTimes.add(txTimes9,gbc);
    gbc.gridx=0;
    gbc.gridy=1;
    pTimes.add(txTimes930,gbc);
    pTimes.setBackground(Color.pink);
    //Re-using gbc for the main panel layout
    gbc.gridx=0;
    gbc.gridy=0;
    gbc.ipadx=30;
    add(pTimes,gbc);
    pAps.setLayout(gbBut);
    gbc.gridx=0;
    gbc.gridy=1;
    gbc.ipady=12;
    pAps.add(txAps,gbc);
    gbc.gridx=1;
    gbc.gridy=0;
    gbc.ipadx=100;
    pAps.setBackground(Color.lightGray);
    add(pAps,gbc);
    setSize(300,300);
    setVisible(true);
    }
}//End of class
```

The output of this code is as follows



GridBag items not covered by this exercise

This exercise has covered the following GridBagConstraints fields

- ipadx/y
- gridx/y
- anchor

The GridBagConstraints class has the following important fields

- weightx/y
- fill
- gridwidth/height
- insets

The weight fields control how an area grows or shrinks beyond its initial size. So if you set the weighty field to zero the field will remain a constant height when you resize the window.

The fill field controls how a component stretches to fill the area. Like the anchor field you set the *fill* values using constants of the GridBagConstraints class. These are

```
GridBagConstraints.NONE
GridBagConstraints.HORIZONTAL
GridBagConstraints.VERTICAL
GridBagConstraints.BOTH
```

The *gridwidth/height* fields determine how many columns and rows a component occupies.

The *insets* field indicates the "external" padding along the cell boundaries.

Quiz

Questions

Question 1)

What best describes the apprearance of an applet with the following code?

```
import java.awt.*;
public class FlowAp extends Frame{
public static void main(String argv[]){
    FlowAp fa=new FlowAp();
    fa.setSize(400,300);
    fa.setVisible(true);
}
FlowAp(){
        add(new Button("One"));
        add(new Button("Two"));
        add(new Button("Three"));
        add(new Button("Four"));
    }//End of constructor
}//End of Application
```

1) A Frame with buttons marked One to Four placed on each edge.
2) A Frame with buutons Makred One to four running from the top to bottom
3) A Frame with one large button marked Four in the Centre
4) An Error at run time indicating you have not set a LayoutManager

---

Question 2)

How do you indicate where a component will be positioned using Flowlayout?

1) North, South,East,West
2) Assign a row/column grid reference
3) Pass a X/Y percentage parameter to the add method

4) Do nothing, the FlowLayout will position the component

Question 3)

How do you change the current layout manager for a container

1) Use the setLayout method
2) Once created you cannot change the current layout manager of a component
3) Use the setLayoutManager method
4) Use the updateLayout method

Question 4)

What will happen if you add a vertical scroll bar to the North of a Frame?

1) The Frame will enlarge to allow the scrollbar to become its preferred size
2) It will be wide, fat and not very useful
3) You cannot add a vertical scroll bar to the North of a frame, only the East or West
4) The scrollbar will stretch from the top to the bottom of the Frame

---

Question 5)

What happens if you add more buttons to a GridLayout than can fit and and fully display the button labels?

1) The size of the container is increased to allow the button labels to fully display
2) The GridLayout ignores the size of the label and the labels will be truncated
3) A compile time error indicating the Buttons cannot be the preferred size
4) A run time error indicating the buttons cannot be the preffered size.

Question 6)

Which of the following statements are true?

1) You can control component placing by calling *setLayout(new GridBagConstraints())*
2) The FlowLayout manager can be used to control component placing of the GridBagLayout
3) The GridBagLayout manager takes constraints of North, South, East, West and Center
4) None of these answers is true

Question 7)

Which of the following are fields of the GridBagConstraints class?

1) ipadx

2) fill
3) insets
4)width

Question 8)

What most closely matches the appearance when this code runs?

```
import java.awt.*;
public class CompLay extends Frame{
    public static void main(String argv[]){
    CompLay cl = new CompLay();
    }
CompLay(){
    Panel p = new Panel();
    p.setBackground(Color.pink);
    p.add(new Button("One"));
    p.add(new Button("Two"));
    p.add(new Button("Three"));
    add("South",p);
    setLayout(new FlowLayout());
    setSize(300,300);
    setVisible(true);
    }
}
```

1) The buttons will run from left to right along the bottom of the Frame
2) The buttons will run from left to right along the top of the frame
3) The buttons will not be displayed
4) Only button three will show occupying all of the frame

Question 9)

Which statements are correct about the *anchor* field?

1) It is a field of the GridBagLayout manager for controlling component placement
2) It is a field of the GridBagConstraints class for controlling component placement
3) A valid settting for the anchor field is GridBagConstraints.NORTH
4) The anchor field controls the height of components added to a container

---

Question 10)

When using the GridBagLayout manahger, each new component requires a new instance of the GridBagConstraints class. Is this statement

1) true

2) false

Answers

Answer 1)

3) A Frame with one large button marked Four in the Centre

If you do not specify a constraint any components added to a Container with the BorderLayout will be placed in the centre. The default layout for a Frame is the BorderLayout

Answer 2)

4) Do nothing, the FlowLayout will position the component

Answer 3)

1) Use the setLayout method

Answer 4)

2) It will be wide, fat and not very useful

Answer 5)

2) The GridLayout ignores the size of the label and the labels will be truncated

Answer 6)

4) None of these answers is true

Answer 7)

1) ipadx
2) fill
3) insets

Answer 8)

2) The buttons will run from left to right along the top of the frame

When the layout manager is changed to FlowLayout the default BorderLayout no longer applies and the panel is placed at the top of the Frame

Answer 9)

2) It is a field of the GridBagConstraints class for controlling component placement
3) A valid settting for the anchor field is GridBagconstraints.NORTH

Answer 10)

2) false

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html

Richard Baldwin Covers this topic at
BorderLayout
http://www.geocities.com/Athens/7077/Java114.htm
FlowLayout
http://www.geocities.com/Athens/7077/Java116.htm
GridLayout
http://www.geocities.com/Athens/7077/Java118.htm
Richard does not appear to cover the GridBagLayout

Jan Newmarsh in Australia has created this page
http://pandonia.canberra.edu.au/java/xadvisor/gridbag/gridbag.html

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec8.html#obj25

Last updated
12 Novemberr 2000

Java2 Certification Tutorial

8)The java.awt package

Objective 2)

Write code to implement listener classes and methods and in listener methods extract information from the event to determine the affected component, mouse position nature and time of the event. State the event classname for any specified event listener interface in the java.awt.event package.

Note on this objective

This objective can seem quite a tall order as there are many different graphical elements that generate different types of event. Thus a mouse will create one sort of event whereas a frame opening or closing will create an altogether different type of event. However much of what is required is memorisation so part of the task is just repetition untill you are familiar with the classes, interfaces and event methods.s

The listener event model

To write any useful GUI applications with Java you need to understand the listener classes and how to extract information from the events they process. The Java event handling system changed significantly between versions 1.0x and 1.1. In version 1.0x the event handling code concept was a little like plain C code for windows, i.e. fairly horrible. It required the creation of huge *case* statements where you would put in code to process a particular event according to parameters. This system is quite easy to understand for trivial examples but does not scale well for larger programs.

I get the impression that the only thing you need to know about the 1.1 exam for the 1.1 or Java2 exam is that the 1.1 approach is not backwardly compatible. In theory, code written for the 1.0x style of event handling should work OK in later versions of the JDK.

The JDK 1.1 event model

The Java 1.1 system involves using listener classes that are effectively "attached" to components to process specific events. This lends itself well for GUI builders to generate event handling code. If you examine the code generated by a GUI builders it can seem a little opaque, partly because it tends to involve inner classes created within methods. For the purpose of learning you can treat the event handling classes as top level classes.

One of the complicating factors for event handling is that it is based on Interfaces but is easiest to use with a series of classes known as the *Adapter* classes, that simply implement the event Interfaces. When you use an interface you need to implement all of its methods, thus direct use of the EventListener interface requires the creation of blank bodies for any unused event handling methods. By using the Adapter classes you only need to create the bodies of event handling methods you actually use.

The adapter classes allow you to use the Listener Interfaces without having to create a body for every method.

One of the most essential events to handle for a stand alone application is the simple ability to shut down an application in response to choosing the *close* option from the system menu. It may come as a surprise at first that this does not come as a default with a Java AWT Frame. If you create an application that extends Frame, but do not create code to handling closing, you will have to either kill it from the task manager or go back to the command line and hit control-c.

The equivalent *Swing* component to Frame, JFrame does process closing as a default action, but the certification does not cover the Swing components. As you must do this for the AWT Frame it is a good place to start covering the subject of event handling

The methods for *WindowEvent* handling are not as intuitive as some of the other Event methods. Thus it is not obvious at first if you need to respond to

windowClosed or windowClosing

In fact it is the windowClosing method method that needs to be processed. The simplest way to destroy the window is to use

```
System.exit(0);
```

Thus if you have an application with the main display of a frame you can give it the ability to close itself by creating a class that sub classes the WindowAdapter class and overrides the WindowClosing event with a new version that simply has the line

System.exit(0);

as the body of that method.

Here is an example of a simple application that shows a Frame that will respond by disappearing when you click the System/close menu choice. I

```
import java.awt.event.*;  //Make event handling available
import java.awt.*;
public class ShutHello extends Frame{
public static void main(String argv[]){
        ShutHello h = new ShutHello();
    }

    ShutHello(){
        Button b = new Button("ShutHello");
        //Create a new instance of the WindowCloser class
        WindowCloser wc = new WindowCloser();
        //Attach that listener to this program
        addWindowListener(wc);
        this.add(b);
        setSize(300,300);
        setVisible(true);
    }
}

class WindowCloser extends WindowAdapter{
        //override one of the methods in the Adapter class
        public void windowClosing(WindowEvent e){
        System.exit(0);
        }
}
```

The following example demonstrates how to use the interface classes directly rather than using the Adapter classes that wrap them and eliminate the need for blank method bodies.

The second half of the objective asks you to know the event class name for any event listener interface. The following table lists all of the Listener interfaces along with their methods. Do not be too put off by the apparent number of Interfaces and methods as they fit naturally into fairly intuitive groups based around things you would expect to be able to do with GUI components.

Thus the *MouseListener* interface offers methods for

- clicked
- pressed
- released
- entered
- exited

If you compare this with event handlers in Visual Basic 5 the only significant area not covered is a set of methods for handling dragdrop events.

The name of the Event class passed to each method is fairly intuitive and based on the name of the Listener class. Thus all of the ActionListener methods take a parameter of ActionEvent, the ComponentListener methods take a ComponentEvent type, ContainerListener takes ComponentEvent etc etc etc.

There are 11 Listener Interfaces in all, but only 7 of them have multiple methods. As the point of the adapters is to remove the need to implement blank methods, Adapters classes are only implemented for these 7 Interfaces.

These are as follows

- ComponentAdapter
- ContainerAdapter
- FocusAdapter
- KeyAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter

The following table shows the full list of Event handling interfaces

Event Handling Interfaces

ActionListener	actionPerformed(ActionEvent)	addActionListener()
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	addAdjustmentListener()
ComponentListener	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)	addComponentListener()
ContainerListener	componentAdded(ContainerEvent) componetRemoved(ContainerEvent)	addContainerListener()
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	addFocusListener()
ItemListener	itemStateChanged(ItemEvent)	addItemListener()
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	addKeyListener()
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	addMouseListener()

MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)	addMouseMotionListener()
TextListener	textValueChanged(TextEvent)	addTextListener()
WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) |windowOpened(WindowEvent)	addWindowListener()

Quiz

Questions

Question 1)

Which of the following statements are true?

1) For a given component events will be processed in the order that the listeners were added
2) Using the Adapter approach to event handling means creating blank method bodies for all event methods
3) A component may have multiple listeners associated with it
4) Listeners may be removed once added

Question 2)

Which of the following are correct event handling methods?

1) mousePressed(MouseEvent e){}
2) MousePressed(MouseClick e){}
3) functionKey(KeyPress k){}
4) componentAdded(ContainerEvent e){}

Question 3)

What will happen when you attempt to compile and run the following code?

```
import java.awt.*;
import java.awt.event.*;
public class MClick extends Frame implements MouseListener{
public static void main(String argv[]){
            MClick s = new MClick();
      }
       MClick(){
                this.addMouseListener(this);
        }
        public void mouseClicked(MouseEvent e){
                System.out.println(e.getWhen());
        }
}
```

1) Compile time error
2) Run time error
3) Compile and at runtime the date and time of each click will be output
4) Compile and at runtime a timestamp wil be output for each click

---

Question 4)

Which of the following statments are true about event handling?

1) The 1.1 Event model is fully backwardly compatible with the 1.0 event model
2) Code written for the 1.0x Event handling will run on 1.1 versions of the JVM
3) The 1.1 Event model is particularly suited for GUI building tools
4) The dragDrop event handler was added with the 1.1 version of event handling.

Answers

Answer 1)

3) A component may have multiple listeners associated with it
4) Listeners may be removed once added

Answer 2)

1) mousePressed(MouseEvent e){}
4) componentAdded(ContainerEvent e){}

Answer 3)

1) Compile time error

Because this code uses the Event listener, bodies must be created for each method in the Listener. This code will cause errors warning that MClick is an abstract class.

Answer 4)

2) Code written for the 1.0x Event handling will run on 1.1 versions of the JVM
3) The 1.1 Event model is particularly suited for GUI building tools

Code written for the 1.1 event handling will not work with a 1.0x version of the JVM. I invented the name *dragdrop* method.

Other sources on this topic

The Sun Tutorial
http://java.sun.com/docs/books/tutorial/uiswing/events/intro.html

Richard Baldwin
http://www.Geocities.com/Athens/7077/Java080.htm#design goals of the jdk 1.1 delegation

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec8.html#obj25

David Reilly
http://www.davidreilly.com/jcb/java107/java107.html

Last updated
24 Mar 2001

Java2 Certification Tutorial

9) The java.lang package

Objective 1)

Write code using the following methods of the java.lang.Math class: abs ceil floor max min random round sin cos tan sqrt.

Note on this objective

The Math class is final and these methods are static. This means you cannot subclass Math and create modified versions of these methods. This is probably a good thing, as it reduces the possibility of ambiguity. You will almost certainly get questions on these methods and it would be a real pity to get any of them wrong just because you overlooked them.

abs

Due to my shaky Maths background I had no idea what *abs* might do until I studied for the Java Programmer Certification Exam. It strips off the sign of a number and returns it simply as a number. Thus the following will simply print out 99. If the number is not negative you just get back the same number.

```
System.out.println(Math.abs(-99));
```

ceil

This method returns the next whole number up that is an integer. Thus if you pass

```
ceil(1.1)
```

it will return a value of 2.0

If you change that to

```
ceil(-1.1)
```

the result will be -1.0;

floor

According to the JDK documentation this method returns

the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer.

If that is not entirely clear, here is a short program and its output

```
public class MyMat{
public static void main(String[] argv){
        System.out.println(Math.floor(-99.1));
        System.out.println(Math.floor(-99));
        System.out.println(Math.floor(99));
        System.out.println(Math.floor(-.01));
        System.out.println(Math.floor(0.1));
        }
}
```

And the output is

```
-100.0
-99.0
99.0
-1.0
0.0
```

max and min

Take note of the following two methods as they take two parameters. You may get questions with faulty examples that pass them only one parameter. As you might expect these methods are the equivalent of

"which is the largest THIS parameter or THIS parameter"

The following code illustrates how these methods work

```
public class MaxMin{
public static void main(String argv[]){
        System.out.println(Math.max(-1,-10));
        System.out.println(Math.max(1,2));
        System.out.println(Math.min(1,1));
        System.out.println(Math.min(-1,-10));
        System.out.println(Math.min(1,2));
```

```
        }
}
```

Here is the output

```
-1
2
1
-10
1
```

random

Returns a random number between 0.0 and 1.0.

Unlike some random number system Java does not appear to offer the ability to pass a seed number to increase the randomness. This method can be used to produce a random number between 0 and 100 as follows.

For the purpose of the exam one of the important aspects of this method is that the value returned is between 0.0 and 1.0. Thus a typical sequence of output might be

0.9151633320773057
0.25135231957619386
0.10070205341831895

Often a program will want to produce a random number between say 0 and 10 or 0 and 100. The following code combines math code to produce a random number between 0 and 100.

```
System.out.println(Math.round(Math.random()*100));
```

round

Rounds to the nearest integer. So, if the value is more than half way towards the higher integer, the value is rounded up to the next ingeter. If the number is less than this the next lowest integer is returned. So for example if the input to round is x then :

2.0 <=x < 2.5. then Math.round(x)==2.0
2.5 <=x < 3.0 the Math.round(x)==3.0

Here are some samples with output

```
System.out.println(Math.round(1.01));
System.out.println(Math.round(-2.1));
System.out.println(Math.round(20));
1
-2
```

20

sin cos tan

These trig methods take a parameter of type *double* and do just about what trig functions do in every other language you have used. In my case that is 12 years of programming and I have never used a trig function. So perhaps the thing to remember is that the parameter is a double.

sqrt

returns a *double* value that is the square root of the parameter.

Summary

- max and min take two parameters
- random returns value between 0 and 1
- abs chops of the sign component
- round rounds to the nearest integer but leaves the sign

Quiz

Questions

Question 1)

Which of the following will compile correctly?

1) System.out.println(Math.max(x));
2) System.out.println(Math.random(10,3));
3) System.out.println(Math.round(20));
4) System.out.println(Math.sqrt(10));

Question 2)

Which of the following will output a random with values only from 1 to 10?

1) System.out.println(Math.round(Math.random()* 10));

2) System.out.println(Math.round(Math.random() % 10));
3) System.out.println(Math.random() *10);
4) None of the above

Question 3)

What will be output by the following line?

```
System.out.println(Math.floor(-2.1));
```

1) -2
2) 2.0
3) -3
4) -3.0

Question 4)

What will be output by the following line?

```
System.out.println(Math.abs(-2.1));
```

1) -2.0
2) -2.1
3) 2.1
4) 1.0

Question 5)

What will be output by the following line?

```
System.out.println(Math.ceil(-2.1));
```

1) -2.0
2) -2.1
3) 2.1
3) 1.0

Question 6)

What will happen when you attempt to compile and run the following code?

```
class MyCalc extends Math{
public int random(){
        double iTemp;
        iTemp=super();
        return super.round(iTemp);
        }
```

```
}

public class MyRand{
public static void main(String argv[]){
        MyCalc m = new MyCalc();
        System.out.println(m.random());
        }
}
```

1) Compile time error
2) Run time error
3) Output of a random number between 0 and 1
4) Output of a random number between 1 and 10

Answers

Answer 1)

3) System.out.println(Math.round(20));
4) System.out.println(Math.sqrt(10));

Option one is incorrect as max takes two parameters and option two is incorrect because random takes no parameters.

Answer 2)

4) None of the above
The closest is option 1 but the detail to remember is that random will include the value zero and the question asks for values between 1 and 10.

Answer 3)

4) -3.0

Answer 4)

3) 2.1

Answer 5)

1) -2.0

Answer 6)

1) Compile time error

The math class is final and thus cannot be subclassed (MyCalc is defined as extending Math). This code is a mess of errors, you can only use super in a constructor but this code uses it in the random method.

Other sources on this topic
Jyothi Krishnan on this topic at http://www.geocities.com/SiliconValley/Network/3693/obj_sec9.html#obj28

Last updated
25 Dec 2000

most recent version at www.software.u-net.com

Java2 Certification Tutorial

9) The java.lang package

Objective 2)

Note on this objective

Descibe the significance of the immutability of String objects

The theory of the immutability of the String class says that once created, a string can never be changed. Real life experience with Java programming implies that this is not true.

Take the following code

```
public class ImString{
public static void main(String argv[]){
        String s1 = new String("Hello");
        String s2 = new String("There");
        System.out.println(s1);
        s1=s2;
        System.out.println(s1);
        }
}
```

If Strings cannot be changed then *s1* should still print out Hello, but if you try this snippet you will find that the second output is the string "There". What gives?

The immutability really refers to what the String reference points to. When *s2* is assigned to *s1* in the example, the String containing "Hello" in the String pool is no longer referenced and *s1* now points to the same string as *s2*. The fact that the "Hello" string has not actually been modified is fairly theorectical as

you can no longer "get at it".

The objective asks you to recognise the implications of the immutability of strings, and the main one seems to be that if you want to chop and change the contents of "strings" the StringBuffer class comes with more built in methods for the purpose.

Because concatenating string causes a new string to be instantiated "behind the scenes", there can be a performance overhead if you are manipulating large numbers of strings, such as reading in a large text file. Generally String immutability doesn't affect every day programming, but it will be questioned on the exam. Remember whatever round about way the question asks it, once created a String itself cannot be changed even if the reference to it is changed to point to some other String. This topic is linked to the way Strings are created in a "String pool", allowing identical strings to be re-used. This is covered in topic 5.2 as part of how the=operator and *equals* method acts when used with strings. Although neither the Java2 nor Java 1.1 objectives specifically mention it I am fairly confident that some questions require a knowledge of the StringBuffer class.

Quiz

Questions

Question 1)

You have created two strings containing names. Thus

String fname="John";
String lname="String"

How can you go about changing these strings to take new values within the same block of code?

1)
fname="Fred";
lname="Jones";
2)
String fname=new String("Fred");
String lname=new String("Jones");

3)
StringBuffer fname=new StringBuffer(fname);
StringBuffer lname=new StringBuffer(lname);

4) None of the above

Question 2)

You are creating a program to read in an 8MB text file. Each new line read adds to a String object but you are finding the performance sadly lacking. Which is the most likely explanation?

1) Java I/O is designed around a lowest common denominator and is inherently slow
2) The String class is unsuitable for I/O operations, a character array would be more suitable
3) Because strings are immutable a new String is created with each read, changing to a StringBuffer may increase performance
4) None of the above

Answers

Answer 1)

4) None of the above
Once created a String is read only and cannot be changed Each one of the options actually creates a new string "behind the scenes" and does not change the original. If that seems to go against your experience and understanding read through information on the immuatbility of strings

---

Answer 2)

3) Because strings are immutable a new String is created with each read, changing to a StringBuffer may increase performance

I hope none of you C programmers suggested a character array?

Other sources on this topc

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/strings/stringsAndJavac.html
(doesn't go into much detail)

Jyothi Krishnan on this topic at

http://www.geocities.com/SiliconValley/Network/3693/obj_sec9.html#obj29

Last updated
16 Sep 2000

most recent version at www.jchq.net

Java2 Certification Tutorial

10) The java.util package

Objective 1)

Make appropriate selection of collection classes/interfaces to suit specified behavior requirements.

Note on this Objective

Although it does not mention it specifically, this objective involves one of the new objectives for the Java2 version of the exam, a knowledge of the collection classes. The exam questions on these new collections are fairly basic, requiring a knowledge of where and how you might use them, rather than a detailed knowledge of the fields and methods.

The old collections

The Java 2 API includes new interfaces and classes to enhance the collections available. Earlier versions of Java included

- vector
- hashtable
- array
- BitSet

Of these, only *array* was included in the objectives for the 1.1 certification exam. One of the noticeable omissions from Java 1.1 was support for sorting, a very common requirement in any programming situation,

The new collections

At the root of the Collection API is the *Collection* interface. This gives you a series of common methods that all collection classes will have. You would probably never create your own class that implements Collection as Java supplies a series of sub-interfaces and classes that uses the *Collection* interface.

The Java2 API includes the following new collection interfaces

- Sets
- Maps

Classes that implement the *Collection* interface store objects as elements rather than primitives. This approach has the drawback that creating objects has a performance overhead and the elements must be cast back from *Object* to the appropriate type before being used. It also means that the collections do not check that the elements are all of the same type, as an object can be just about anything.

A Set

A *Set* is a collection interface that cannot contain duplicate elements. It thus matches nicely onto concepts such as a record set returned from a relational database. Part of the magic of the *Set* interface is in the *add* method.

add(Object o)

Any object passed to the *add* method must implement the *equals* method so the value can be compared with existing objects in the class. If the set already contains this object the call to *add* leaves the set unchanged and returns *false*. The idea of returning *false* when attempting to add an element seems more like the approach used in C/C++ than Java. Most similar java methods would seem to throw an Exception in this type of situation.

A List

A list is a sorted collection interface that can contain duplicates

Some important methods are

- add
- remove
- clear

The JDK documentation gives the example of using *List* to manage an actual GUI list control containing a list of the names of the Planets.

A Map

Map is an interface, classes that implement it cannot contain duplicate keys, and it is similar to a hashtable.

Why use Collections instead of arrays?.

The big advantage of the collections over arrays is that the collections are growable, you do not have to assign the size at creation time. The drawback of collections is that they only store objects and not primitives and this comes with an inevitable performance overhead. Arrays do not directly support sorting, but this can be overcome by using the static methods of the Collections. Here is an example.

```
import java.util.*;
public class Sort{
    public static void main(String argv[]){
    Sort s = new Sort();
    }
Sort(){
    String s[] = new String[4];
    s[0]="z";
    s[1]="b";
    s[2]="c";
    s[3]="a";
    Arrays.sort(s);
    for(int i=0;i< s.length;i++)
    System.out.println(s[i]);
    }
}
```

Set and Map collections ensure uniqueness, List Collections do not ensure uniqueness but are sorted (ordered)

Using Vectors

The following example illustrates how you can add objects of different classes to a *Vector*. This contrasts with arrays where every element must be of the same type. The code then walks through each object printing to the standard output. This implicitly access the *toString*() method of each object.

```
import java.awt.*;
import java.util.*;
public class Vec{
public static void main(String argv[]){
    Vec v = new Vec();
    v.amethod();
    }//End of main

public void amethod(){
        Vector mv = new Vector();
        //Note how a vector can store objects
        //of different types
```

```
        mv.addElement("Hello");
        mv.addElement(Color.red);
        mv.addElement(new Integer(99));
        //This would cause an error
        //As a vector will not store primitives
        //mv.addElement(99)
        //Walk through each element of the vector
        for(int i=0; i< mv.size(); i++){
            System.out.println(mv.elementAt(i));
            }
   }//End of amethod
}
```

Prior to Java2 the *Vector* class was the main way of creating a re-sizable data structure. Elements can be removed from the Vector class with the *remove* method.

Using Hashtables

Hashtables are a little like the Visual Basic concept of a Collection used with a key. It acts like a *Vector*, except that instead of referring to elements by number, you refer to them by key. The *hash* part of the name refers to a math term dealing with creating indexes. A hashtable can offer the benefit over a *Vector* of faster look ups.

BitSet

A *BitSet* as its name implies, stores a sequence of Bits. Don't be misled by the "set" part of its name its not a set in the mathematical or database sense, nor is it related to the Sets available in Java2. It is more appropriate to think of it as a bit vector. A *BitSet* may useful for the efficient storage of bits where the bits are used to represent *true/false values*. The alternative of using some sort of collection containing Boolean values can be less efficient.

According to Bruce Eckel in "Thinking in Java"

It’s efficient only from the standpoint of size; if you’re looking for efficient access, it is slightly slower than using an array of some native type.

The BitSet is somewhat of a novelty class which you may never have a need for. I suspect that it might be handy for the purposes of cryptography or the processing of images. Please let me know if you come across a question relating to it in the Java2 exam.

Quiz

Question 1)

Which of the following are collection classes?

1) Collection
2) Iterator
3) HashSet
4) Vector

Question 2)

Which of the following are true about the Collection interface?

1) The Vector class has been modified to implement Collection
2) The Collection interface offers individual methods and Bulk methods such as addAll
3) The Collection interface is backwardly compatible and all methods are available within the JDK 1.1 classes
4) The collection classes make it unnecessary to use arrays

Question 3)

Which of the following are true?

1) The Set interface is designed to ensure that implementing classes have unique members
2) Classes that implement the List interface may not contain duplicate elements
3) The Set interface is designed for storing records returned from a database query
4) The Map Interface is not part of the Collection Framework

Question 4)

Which of the following are true

1) The elements of a Collection class can be ordered by using the sort method of the Collection interface
2) You can create an ordered Collection by instantiating a class that implements the List interface
3) The Collection interface sort method takes parameters of A or D to change the sort order, Ascending/Descending

4) The elements of a Collection class can be ordered by using the order method of the Collection interface

---

Question 5)

You wish to store a small amount of data and make it available for rapid access. You do not have a need for the data to be sorted, uniqueness is not an issue and the data will remain fairly static Which data structure might be most suitable for this requirement?

1) TreeSet
2) HashMap
3) LinkedList
4) an array

Question 6)

Which of the following are Collection classes?

1) ListBag
2) HashMap
3) Vector
4) SetList

Question 7)

How can you remove an element from a Vector?

1) delete method
2) cancel method
3) clear method
4) remove method

Answers

Answer 1)

3) HashSet
4) Vector

The other two are Interfaces not classes

Answer 2)

1) The Vector class has been modified to implement Collection
2) The Collection interface offers individual methods and Bulk methods such as addAll

The Collection classes are new to the JDK1.2 (Java2) release. With the exception of the classes that have been retrofitted such as Vector and BitSet the, if you run any of the Collections through an older JDK you will get a compile time error.

Answer 3)

1) The Set interface is designed to ensure that implementing classes have unique members

Elements of a class that implements the List interface may contain duplicate elements. Although a class that implements the Set interface might be used for storing records returned from a database query, it is not designed particularly for that purpose.

Answer 4)

2) You can create an ordered Collection by instantiating a class that implements the List interface

Answer 5)

4) an array

For such a simple requirement an ordinary array will probably be the best solution

Answer 6)

2) HashMap
3) Vector

With the release of JDK 1.2 (Java2) the Vector class was "retro-fitted" to become a member of the Collection Framework

Answer 7)

4) remove method

Other sources on this topic

The Sun Tutorial
http://java.sun.com/docs/books/tutorial/collections/index.html

Jyothi Krishnan on this topic at
http://www.geocities.com/SiliconValley/Network/3693/obj_sec10.html#obj30

Last updated
21 May 2002

most recent version at http://www.jchq.net

index

home

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

The Rusty Harold/O'Reilly Java I/O book

O'Reilly have published a book specifically about Java I/O It get very good reviews at amazon. If you buy it from the following links I will get a small commission on the purchase

Buy from Amazon.com or from Amazon.co.uk

---

11) The java.io package

Objective 1)

Write code that uses objects of the file class to navigate a file system.

In his excellent book Just Java and Beyond Peter van der Linden starts his chapter on File I/O by saying

"It is not completely fair to remark, as some have, that support for I/O in java is "bone headed".

I think he was implying that it is not the perfect system, and so it is an area worthy of double checking your knowledge of before you go for the exam. When you are learning it you have the compensation that at least it is a useful area of the language to understand.

The java.io package is concerned with input and output. Any non trivial program will require I/O. Anything from reading a plain comma delimeted text file, a XML data file or something more exotic such as a network stream. The good news is that the Programmer Certification Exam only expects you to understand the basics of I/O, you do not have to know about Networking or the more exotic aspects of I/O.

Java I/O is based on the concept of streams. The computer term streams was first popularised with the Unix operating system and you may like to consider it as being an analogy with a stream of water. You have a stream of bits coming in at one end, you apply certain filter to process the stream. Out the other end of the pipe you send a modified version of the stream which your program can process..

The names of the I/O Stream classes are not intuitive and things do not always work as you might guess.

The File Class

The File class is not entirely descriptive as an instance of the File class represents a file or directory name rather than a file itself.

My first assumption when asked about navigating a file system would be to look for a method to change directory. Unfortunately the File class does not have such a method and it seems that you simply have to create a new File object with a different directory for the constructor.

Also the exam may ask you questions about the ability to make and delete files and directories which may be considered to come under the heading of navigating the file system.

Creating an instance of the File class does not create a file in the underlying operating system

The file class offers

delete()

To delete a file or directory

mkdir() and mkdirs()

To create directories.

The File class contains the list() which returns a string array containing all of the files in a directory. This is very handy for checking to see if a file is available before attempting to open it. An example of using list.

```
import java.io.*;
public class FileNav{
public static void main(String argv[]){
          String[] filenames;
          File f = new File(".");
          filenames = f.list();
          for(int i=0; i< filenames.length; i++)
            System.out.println(filenames[i]);
          }
}
```

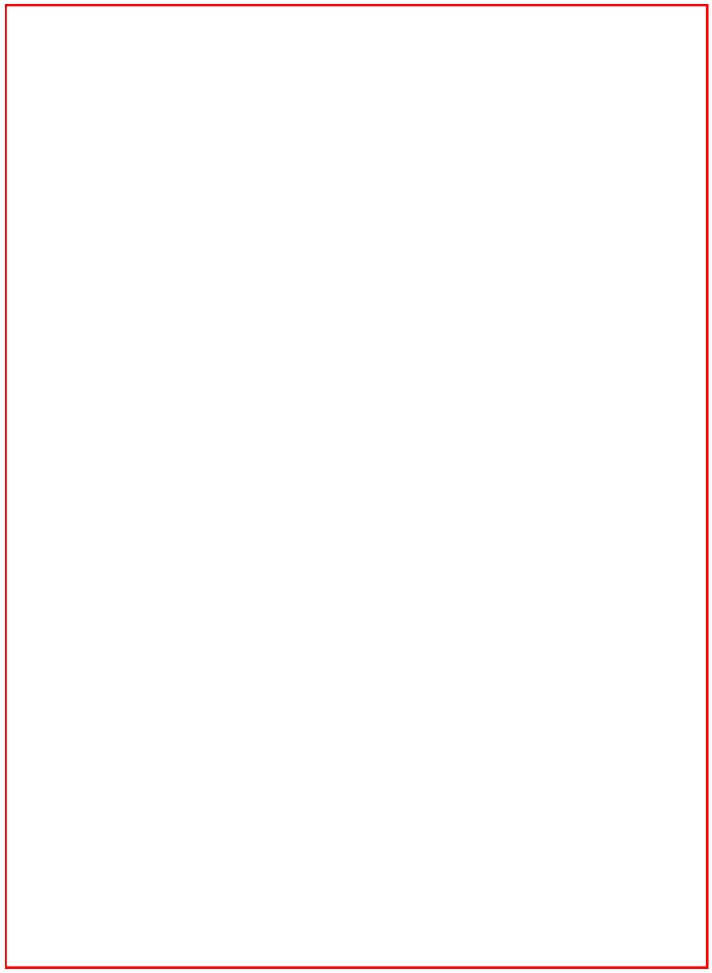
This simply outputs a list of the files in the current directory ("*.*")

Platform Independence

The file class is important in writing pure java. I used to think that pure Java was just about not including native code, but it also refers to writing platform independent code. Because of the differences between in the way File systems work it is important to be aware of platform dependencies such as the directory separator character. On Win/DOS it is a backslash \, on Unix it is a forward slash / and on a Mac it is something else. You can get around this dependency by using the File.separator constant instead of hard coding in the separator literal. You can see this in use in the Filer example program that follows.

A program to navigate the file system

The following code is rather long (90 odd lines), but if you can make sense of this you will know most of what you need to understand the objective. The program allows you to browse the files in a directory and to change directories. It was partly inspired by some code in the Java in a Nutshell Examples book from O'reilly. A book I highly recommend. Here is a screen shot of this program in action under Linux



```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
public class Filer extends Frame implements ActionListener{
  /*************************************************************
  Marcus Green October 2000 Part of the Java Programmer Certification
  tutorial available at http://www.jchq.net. Addressing the objective to be able
  to use the File class to navigate the File system.This program will show a
  list of files in a directory .Clicking on a directory will change to the directory
  and show the contentsNote the use of File.separator to allow this to work on
  Unix or PC (and maybe even the Mac)
  *************************************************************/
  List lstFiles;
  File currentDir;
  String[] safiles;
  int iEntryType = 6;
  int iRootElement = 0;
  int iElementCount = 20;

  public static void main(String argv[]){
    Filer f = new Filer();
    f.setSize(300,400);
    f.setVisible(true);
  }
  Filer(){
    setLayout(new FlowLayout());
    lstFiles = new List(iElementCount);
```

```
  lstFiles.addActionListener(this);
  //set the current directory
  File dir = new File(System.getProperty("user.dir"));
  currentDir = dir;
  listDirectory(dir);
  add(lstFiles);
  addWindowListener(
  new WindowAdapter(){
    public void windowClosing(WindowEvent e){
      System.exit(0);
    }

  } );


}

public  void actionPerformed(ActionEvent e){
  int i = lstFiles.getSelectedIndex();
  if(i==iRootElement){
    upDir(currentDir);
  }else{
  String sCurFile = lstFiles.getItem(i);
  //Find the length of the file name and then
  //chop of the filetype part (dir or file)
  int iNameLen = sCurFile.length();
  sCurFile = sCurFile.substring(iEntryType,iNameLen);
  File fCurFile = new File(currentDir.toString()+File.separator + sCurFile);
  if(fCurFile.isDirectory()){
       listDirectory(fCurFile);
        }
  }

}
public void  upDir(File currentDir){
  File fullPath = new File(currentDir.getAbsolutePath());
  String sparent = fullPath.getAbsoluteFile().getParent();
 if(sparent == null) {
  //At the root so put in the dir separator to indicate this
    lstFiles.remove(iRootElement);
    lstFiles.add(" "+File.separator+" ",iRootElement);
    return;
  }else{
    File fparent = new File(sparent);
    listDirectory(fparent);
  }
}
public void listDirectory(File dir){
  String sCurPath = dir.getAbsolutePath()+File.separator ;
  //Get the directorie entries
  safiles = dir.list();
  //remove the previous lis and add in the entry
  //for moving up a directory
  lstFiles.removeAll();
```

```
   lstFiles.addItem("[ .. ]");
   String sFileName = new String();
   //loop through the file names and
   //add them to the list control
   for(int i=0; i< safiles.length; i++){
     File curFile = new File(sCurPath + safiles[i]);
      if(curFile.isDirectory()){
         sFileName = "[dir ]" + safiles[i];
         }else{
           sFileName = "[file]"+safiles[i];
       }
     lstFiles.addItem(sFileName);
   }
   add(lstFiles);
 currentDir=dir;
  }

}
```

Quiz

Questions

Question 1)

Which of the following will distinguish between a directory and a file

1) FileType()
2) isDir()
3) isDirectory()
4) getDirectory()

Question 2)

Which of the following methods of the File class will delete a directory or file

1) The file class does not allow you to delete a file or directory
2) remove()
3) delete()
4) del()

Question 3)

How can you obtain the names of the files contained in an instance of the File class called dir?

1) dir.list()
2) dir.list

3) dir.files()
4) dir.FileNames()

Question 4)

Which of the following will populate an instance of the File class with the contents of the current directory?

1) File f = new File();
2) File f = new File("*.*");
3) File f = new File('*.*');
4) File f = new File(".");

Question 5)

Given the following code

File f = new File("myfile.txt");

What method will cause the file "myfile.txt" to be created in the underlying operating system.?

1) f.write();
2) f.close();
3) f.flush();
4) none of the above

Question 6)

Which of the following will chenge to the next directory above the current directory
1) chDir("..");
2) cd(".");
3) up();
4) none of the above

Question 7)

Which of the following are fields or methods of the File class
1) getParent()
2) separator
3) dirname
4) getName();

Answers

Answer to Question 1)

3) isDirectory()

Answer to Question 2)

3) delete()

Answer to Question 3)

1) dir.list()

The list method will return a string array containing the contents of the current directory.

Answer to Question 4)

4) File f = new File(".");

This construction for the File class will obtain the contents of the current directory on a Dos or Unix style system but I am not sure what might happen on some other system with a more exotic file structure such as the Mac OS.

Answer to Question 5)

4) none of the above

The File class mainly just describes a file that might exist. To actually create it in the underlying operating system you need to pass the instance of the File class to an instance of one of the OutputStream classes.

Answer to Question 6)

4) none of these
Java has no direct way to change the current directory. A way around this is to create a new instance of the file class pointing to the new directory

Answer to Question 7)

1) getParent()
2) separator
4) getName();

Other Sources on this topic

You can browse the samples of the O'Reilly Java I/O book at

http://metalab.unc.edu/javafaq/books/javaio/index.html

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/io/

The Java API on the File class at Sun
http://java.sun.com/products/jdk/1.2/docs/api/java/io/File.html

The JLS on Java IO a bit academic and bare
http://www.infospheres.caltech.edu/resources/langspec-1.0/javaio.doc.html

Richard Baldwin on I/O
http://home.att.net/~baldwin.rg/Intermediate/Java060.htm

Joyothi has some handy tables for the I/O classes at
http://www.geocities.com/SiliconValley/Network/3693/io.html

Last updated
24 Oct 2000

most recent version at www.jchq.net

Java2 Certification Tutorial

You can discuss this topic with others at http://www.jchq.net/discus
Read reviews and buy a Java Certification book at http://www.jchq.net/bookreviews/jcertbooks.htm

---

11) The java.io package

Objective 2)

Write code that uses objects of the classes InputStreamReader and OutputStreamWriter to translate between Unicode and either platform default or ISO 8859-1 character encodings.

I was surprised that this objective was not be emphasized in the JDK1.2 exams as internationalization has been enhanced and is a big feature with Java. It's nice to sell software to a billion Europeans and Americans but a billion Chinese would be a nice additional market (even if you only got 10% of it). This is the kind of objective that even experienced Java programmers may not have experience with, so take note!.

Java Character Encoding: UTF and Unicode

Java uses two closely related encoding systems UTF and unicode. Java was designed from the ground up to deal with multibyte character sets and can deal with the vast numbers of characters that can be stored using the unicode character set. Unicode characters are stored in two bytes which allows for up to 65K worth of characters. This means it can deal with Japanese Chinese, and just about any other character set known. You will be pleased to know that you don't have to give examples of any of these for the exam.

Although unicode can represent almost any character you would ever likely to use it is not an efficient coding method for programming. Most of the text data within a program uses standard ASCII, most of which can easily be stored within one byte. For reasons of compactness Java uses a system called UTF-8 for string literals, identifiers and other text within programs. This can result in a considerable saving by comparison with using unicode where every character requires 2 bytes.

The StreamReader Class

The StreamReader class converts a byte input (i.e. not relating to any character set) into a character input stream, one that has a concept of a character set. If you are only concerned with ASCII style character sets you will probably only use these Reader classes in the form with the constructor

InputStreamReader(InputStream in)

This version uses the platform-dependent default encoding. In JDK 1.1 this default is identified by the file.encoding system property. There seems to be no standard way of finding out what encodings are supported on your platform The

default encoding is generally ISO Latin-1 except on a Mac where it is MacRoman.. If this system property is not defined, the default encoding identifier is 8859_1 (ISO-LATIN-1). The assumption seems to be that if all else fails, revert back to English. Experimenting with other character sets is problematic as the characters may not show up correctly if you environment is not configured appropriately. Thus if you attempt to output a character from the Chinese character set you system may not support it.

If you are dealing with other character sets you can use

```
InputStreamReader(InputStream in, String encoding);
```

The StreamReader and Writer classes can take either a character encoding parameter or be left to use the platform default encoding

Remember that the InputStream comes first and encoding second.

The read and and write methods

The InputStreamReader class has a read() method and the OutputStreamWriter has a write() method that read and write characters. When the read method is called it reads bytes from the input stream and converts them to Unicode characters using the encoding specified in the stream constructor. When the write() method is called the the characters from the stream are converted to their corresponding byte encoding and stored in an internal buffer. When the buffer becomes full the contents are written to the underlying byte output stream.

GreekWriter Example

The sample code for GreekWriter writes a text output file containing some letters in the Greek alphabet. If you open this file Out.txt with an editor you will just see what looks like junk.

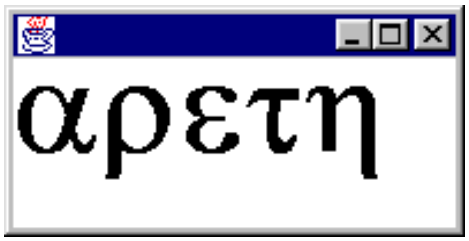
```
import java.io.*;

class GreekWriter {
    public static void main(String[] args) {
        String str = "\u03B1\u03C1\u03B5\u03C4\u03B7";

        try {
            Writer out =
                new OutputStreamWriter(new FileOutputStream("out.txt"), "8859_7");
                //8859_7 is the ISO code for ASCII plus greek, although this
                //example also works on my machine if it is set to UTF8
            out.write(str);
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

GreekReader Example

```
import java.io.*;
import java.awt.*;
```

```
class GreekReader extends Frame{
/*********************************************************
*Companion program to GreekWriter to illustrate
*InputStreamReader and OutputStreamWriter as part
*of the objectives for the Sun Certified Java Programmers
*exam. Marcus Green 2000
*********************************************************/
String str;
    public static void main(String[] args) {
        GreekReader gr = new GreekReader();
        gr.go();
        gr.setWin();
    }
public void go(){

        try {
        FileInputStream fis = new FileInputStream("out.txt");
        InputStreamReader isr = new InputStreamReader(fis,"8859_7");
        Reader in = new BufferedReader(isr);

            StringBuffer buf = new StringBuffer();
            int ch;
            while ((ch = in.read()) > -1) {
                buf.append((char)ch);
            }
            in.close();
           str = buf.toString();

        } catch (IOException e) {
            e.printStackTrace();
        }

 }

    public void paint(Graphics g) {
        //paint method automatically called by the system
        Insets insets = getInsets();
        int x = insets.left, y = insets.top;
        //Add 30 to y or we will only see the
        //downstrokes of the letters
        g.drawString(str, x, y +30);
    }
public void setWin(){
     //Nice big font so we can see the characters.
     Font font = new Font("Monospaced", Font.BOLD, 59);
     setFont(font);
     setSize(200,200);
     setVisible(true);
     //Show the frame
        show();
    }

}
```

Screen Capture from GreekReader



Quiz

Questions

Question 1)

Which of the following statements are true?

1) The OutputStreamWriter must take a character encoding as a constructor parameter
2) The default encoding for the OutputStreamWriter is ASCII
3) The InputStreamWriter and OutputStreamWriter may take a character encoding as a constructor
4) InputStreamReader must take a stream as one of its constructors

Question 2)

Which of the following statements are true?

1) Java can display character sets independently of the underlying operating system
2) The InputStreamReader class may take an instance of another InputStream class as a constructor
3) An InputStreamReader may act as a constructor to an OutputStreamReader to convert between character sets
4) Java uses the ASCII encoding system to store strings internally

Question 3)

Which of the following are correct signatures for InputStreamReader?

1) InputStreamReader(InputStream in, String encoding);

2) InputStreamReader(String encoding,InputStream in);
3) InputStreamReader(String encoding,File f);
4) InputStreamReader(InputStream in);

Question 4)

Which of the following are methods of the InputStreamReader class?

1) read()
2) write()
3) getBuffer()
4) getString()

Question 5)

Which of the following statements are true?

1) Java uses unicode to internally to store string literals
2) Java uses ASCII to internally store string literals
3) Java uses UTF-8 to internally store string literals
4) Java uses the platform native encoding to store string literals

Answers

Answer to Question 1)

3) The InputStreamWriter and OutputStreamWriter may take a character encoding as a constructor
4) InputStreamReader must take a stream as one of its constructors

Answer to Question 2)

1) Java can display character sets independently of the underlying operating system
2) The InputStreamReader class takes may take an instance of another InputStream class as a constructor

Although Java can store characters independently of the underlying operating system, the appropriate font must be installed on the underlying operating system in order to display those characters. Generally streams are chained with like streams, ie InputStreams take constructors of other InputStreams and OutputStreams take constructors of OutputStreams. Java uses the UTF encoding system to store strings internally.

Answer to Question 3)

1) InputStreamReader(InputStream in, String encoding);
4) InputStreamReader(InputStream in);

If you do not specify an encoding the JVM will assume the platform default encoding

Answer to Question 4)

1) read()

Answer to Question 5)

3) Java uses UTF-8 to internally store string literals

Other sources on this topic

The Sun API docs on InputStreamReader and OutputStreamWriter
http://java.sun.com/products/jdk/1.2/docs/api/java/io/OutputStreamWriter.html
http://java.sun.com/products/jdk/1.2/docs/api/java/io/InputStreamReader.html

JavaCaps on this topic
http://www.javacaps.com/sjpc_io_obj2.html

Everything you could want to know about unicode
http://www.unicode.org/

Last updated
8 Nov 2000

most recent version at http://www.jchq.net

Java2 Certification Tutorial

---

11) The java.io package

Objective 3)

Distinguish between conditions under which platform default encoding conversion should be used and conditions under which a specific conversion should be used.

This could be a "bondage and discipline" type of objective. By this I mean that some purists might take the attitude that you should always specify the encoding conversion because you never know where, when and how your program will be used. It was because so many programmers assumed that the code they wrote that would never have to cope with the year 2,000 that there is such a mess at the moment. Well it's a well paying mess for some programmers.

If you take a more benign view, this objective asks you identify if your code is likely to ever have to deal with anything but the default encoding. If your home default encoding is not ISO-LATIN-1 and you consider that English is the international language of Business, or you may need to deal with other character sets, then take advantage of the ability to do specific conversions.

If some of this means nothing to you, re-read the previous section about the Reader and Writer classes.

Other sources on this topic

Sun documentation on internationalisation
http://java.sun.com/docs/books/tutorial/i18n/text/stream.html
http://java.sun.com/products/jdk/1.1/docs/guide/intl/
http://java.sun.com/docs/books/tutorial/i18n/index.html

Last updated
09 Oct 2000

most recent version at www.jchq.net

Java2 Certification Tutorial

11) The java.io package

Objective 4)

Select valid constructor arguments for subclasses from a list of classes in the java.io.package.

The emphasis on this objective is to know that constructors are valid. The most obvious break in the possible constructors is that the RandomFile class does not take a Stream constructor, for more information on RandomAccessFile see the next section.

These children of classes take instances of other streams as constructors. Thus the exam might ask you if they could take an instance of file, a string file, a writer or a path to see if you understand the valid constructors. A valid constructor will be some kind of stream plus possible other parameters.

The Filtering in these classes allow you to access information more usefully than a stream of bytes. It might be useful not to worry about the names FilterInputStream and FilterOutputStream as it is the Subclasses that contain the useful methods. These main subclasses are

FileInputStream and OutputStream

The FileInputStream and FileOutputStream take some kind of File as a constructor. This can be a String containing the file name, and instance of the File class or a File descriptor. These classes are often used to construct the first step in a chain of Stream classes. Typically an FileInputStream will be connected to a File and that will be connected to an instance of InputStreamReader to read text characters. Here is an example of chaining the FileInputStream to the InputStream reader. This program will print out its own source code.

```
import java.io.*;
public class Fis{
public static void main(String argv[]){
        try{
                FileInputStream in = new FileInputStream("Fis.java");
                InputStreamReader isr = new InputStreamReader(in);
                int ch=0;
                while((ch = in.read())> -1){
                        StringBuffer buf = new StringBuffer();
                        buf.append((char)ch);
                        System.out.print(buf.toString());
                        }
                } catch (IOException e){System.out.println(e.getMessage());}
```

```
        }
}
```

It is probably advisable when programming in the "real world" to use the InputStreamReader class in this type of situation to allow for easy of internationalisation.. See the GreekReader example in section 11.01 for an example of this.

BufferedInputStream and BufferedOutputStream

The Buffered streams are direct descendents of the Filter streams. They read in more information than is immediately needed into a buffer. This increases efficiency as when a read occurs it is more likely to be from memory (fast) than from disk (slow). This buffering means they are particularly useful if you are reading in large amounts of data. An example might be where you are processing several tens of megabytes of text data. The BufferedInputStream and BufferedOutputStream take an instance of a stream class as a constructor but may take a size parameter so you can tune the size of the buffer used.

Here is an example of using the BufferedInputStream, note how similar it is the the previous example with InputStreamReader replaced by BufferedInputStream

```
import java.io.*;
public class BufIn{
public static void main(String argv[]){
        try{
                FileInputStream fin = new FileInputStream("BufIn.java");
                BufferedInputStream bin = new BufferedInputStream(fin);
                int ch=0;
                while((ch=bin.read())> -1){
                        StringBuffer buf = new StringBuffer();
                        buf.append((char)ch);
                        System.out.print(buf.toString());
                        }

                }catch(IOException e){System.out.println(e.getMessage());};
        }
}
```

DataInputStream and DataOutputStream

The DataInputStream and OutputStream are used to read binary representations of Java primitives in a portable way. It gives you access to a range of methods such as readDoble, readIn that will work the same on different platforms. In JDK1.0 this was one of the main ways to access unicode text, but has been superceeded by the Reader classes since JDK 1.1. These classes take an instance of a Stream as a constructor

The following examples write a single character to the file system and then read it back and print it to the console.

```
//Write the file
import java.io.*;
        public class Dos{
        public static void main(String argv[]){
        try{
                FileOutputStream fos = new FileOutputStream("fos.dat");
                DataOutputStream dos = new DataOutputStream(fos);
                dos.writeChar('J');
                }catch(IOException e){System.out.println(e.getMessage());}
        }
}
```

```
//Read the file
import java.io.*;
public class Dis{
public static void main(String argv[]){
        try{
        FileInputStream fis= new FileInputStream("fos.dat");

        DataInputStream dis = new DataInputStream(fis);
        System.out.println(dis.readChar());
        }catch(IOException e){System.out.println(e.getMessage());}
        }
}
```

The File class

The File class has three constructor versions. These are

File(String path);
File(String path, String name)
File (File dir, String name);

The three are very similar and perform effectivly the same function. The simple simple String constructor takes the name of the file in a single sting. This can be either an absolute or relative path to the file. The second version takes the path and file name as separate Strings and the third option is very similar to the first except that the first parameter for the directory has the File type instead of String.

RandomAccessFile

The important thing to be aware of with the constructors for RandomAccessFile is that it takes two constructor parameters and the second parameter is a String containing the file mode. See the next section for details of how to use RandomAccessFile.

Questions

Question 1)

Which of the following are valid constructors for the FileInputStream class?

1) File
2) String
3) File descriptor
4) RadomAccessFile

Question 2)

Which of the following are valid constructors for the BufferedInputStream class?

1) BufferedInputStream(FileInputStream in, int size)
2) BufferedInputStream(FileInputStream in)
3) BufferedInputStream(FileOutputStream fos)
4) BufferedInputStream(RandomAccessFile ram)

Question 3

Which of the following are valid constructors for the DataInputStream class

1) DataInputStream(FileInputStream in, int size)
2) DataInputStream(FileInputStream in)
3) DataInputStream(File f)
4) DataInputStream(String s)

Question 4

Given the following code which of the following statements are true?

```
import java.io.*;
public class Dos{
public static void main(String argv[]){
        FileOutputStream fos = new FileOutputStream("fos.dat");
        DataOutputStream dos = new DataOutputStream(fos);
        BufferedOutputStream bos = new BufferedOutputStream(dos);
        dos.write('8');
        }
}
```

1) The code will not compile
2) No compilation because BufferedOutputStream cannot have a DataOutputStream constructor
3) The code will compile and write the byte 8 to the file
4) The code will compile and write the string "8" to the file

Question 5)

Which of the following are valid constructor parameters?
1) File (String path);
2) File(String path, String name)
3) RandomAccessFile(File)
4) File(RandomAccesFile name)

Question 6)

Given the following code

```
import java.io.*;
public class Ppvg{
public static void main(String argv[]){
        Ppvg p = new Ppvg();
        p.go();
        }

public void go(){
        try{
                DataInputStream dis = new DataInputStream(System.in);
                dis.read();
                }catch(Exception e){}
        System.out.println("Continuing");
        }
}
```

Which of the following statements are true?

1) The code will compile and pause untill a key is hit
2) The code will not compile because System.in is a static class
3) The code will compile and run to completion without output
4) The code will not compile because System.in is not a valid constructor for DataInputStream

Answers

Answer to question 1)

1) File
2) String
3) File descriptor

Answer to Question 2

1) BufferedInputStream(FileInputStream in, int size)
2) BufferedInputStream(FileInputStream in)

It should be fairly obvious that an InputStream would not take an instance of an outputstream (option 3) and the RandomAccesFile is not a stream class (option 4)

Answer to Question 3

2) DataInputStream(FileInputStream in)

Answer to Question 4)

1) The code will not compile

The code will not compile because there is no try/catch block. A BufferedOutputStream may take a DataOutputStream as a constructor.

Answer to Question 5

Which of the following are valid constructor parameters?

1) File (String path);
2) File(String path, String name)
RandomAccessFile must take a mode parameter (see the next section for details of the RandomAccessFile class).

Answer to Question 6)

1) The code will compile and pause untill a key is hit

Other sources on this topic

The Sun API docs
Buffered I/O
http://java.sun.com/products/jdk/1.2/docs/api/java/io/BufferedInputStream.html
http://java.sun.com/products/jdk/1.2/docs/api/java/io/BufferedOutputStream.html

Data I/O streams
http://java.sun.com/products/jdk/1.2/docs/api/java/io/DataInputStream.html

http://java.sun.com/products/jdk/1.2/docs/api/java/io/DataOutputStream.html

Last updated
6 Nov 2000

most recent version at www.jchq.net

Java2 Certification Tutorial

11) The java.io package

Objective 5)

Write appropriate code to read, write and update files using FileInputStream, FileOutputStream, and RandomAccessFile objects.

FileInputStream and FileOutputStream

The following example creates a text file called Out.txt and writes the text Hello into it. If you type out the resulting file you will see that file comes out as H e l l o (with a gap between each letter) . I suspect this is because a character is 16 bits and because ASCII is 8 bit, the upper 8 bits result in a blank character.

This also illustrates the previous objective in that it uses the FilterOutputStream descendent DataOutputStream. This allows human readable chars to be written. If this example only used FileOutputStream methods you would be limited to writing bytes or ints. If you do this and then type the results to the console, it just shows up as junk characters.

```
import java.io.*;

public class Fos{
                    String s = new String("Hello");
                    public static void main(String argv[]){
                    Fos f = new Fos();
                    f.amethod();
                   }



            public void amethod(){
                try{
                FileOutputStream fos = new FileOutputStream("Out.txt");
                //DataOutputStream allows you to write chars
                DataOutputStream dos = new DataOutputStream(fos);
```

```
                dos.writeChars(s);
                }catch(IOException ioe) {}
              }
}
```

The following example will read the text file produced by the last example and output the text to the console. Note that because the previous program will have written Out.txt as unicode, if you create a text file with an editor containing a a string such as "Hello" , the following code will probably print out junk such as question marks.

```
import java.io.*;

public class Fis{
                 public static void main(String argv[]){
                 Fis f = new Fis();
                 f.amethod();
                }


      public void amethod(){
                try{
                FileInputStream fis = new FileInputStream("Out.txt");
                DataInputStream dis = new DataInputStream(fis);
                while(true){
                    char c =dis.readChar();
                    System.out.println(c);
                    }
                }catch(IOException ioe) {}
          }
}
```

RandomAccessFile

The RandomAccessFile class is way out on it's own and doesn't fit neatly with the Stream I/O classes. If you get answer possibilities that mix instances of RandomAccessFile with other streams, then they are almost certainly the wrong answers. RandomAccessFile is more like the File class than the Stream classes. It is useful if you want to move backwards and forwards with in a file without re-opening it.

For its constructor RandomAccessFile takes either an instance of a File class or a string and a read/write mode argument. The mode argument can be either "r" for read only or "rw" can be read and written to. Memorise those two options, don't get caught out in the exam by a bogus mode such as "w", "ro" or "r+w";

Unlike the File class, if you attempt pass the name of a file as a constructor, RandomAccessFile will attempt to open that file. If you pass only the "r" parameter mode and the file does not exist an exception will be thrown. If you pass the mode as "rw" RandomAccessFile will attempt to create the file in the underlying operating system.

The Random Access does not take a stream as a constructor parameter.

This example will read the Out.txt file created by the Fos.java example shown earlier.

Because of the blank high byte (not bit), the results show a question mark with each letter.

```
import java.io.*;

public class Raf{
               public static void main(String argv[]){
                 Raf r = new Raf();
                 r.amethod();
                 }


            public void amethod(){
                 try{
                RandomAccessFile raf = new RandomAccessFile("Out.txt","rw");
                for(int i=0; i<10;i++){
                                raf.seek(i);
                                char myc = raf.readChar();
                                //?Show for high bytes
                                System.out.println(myc);
                                }
                           } catch(IOException ioe) {}
              }
}
```

Quiz

Questions

Question 1)

Assuming any exception handling has been set up, which of the following will create an instance of the RandomAccessFile class?

1) RandomAccessFile raf = new RandomAccessFile("myfile.txt","rw");
2) RandomAccessFile raf = new RandomAccessFile(new DataInputStream());
3) RandomAccessFile raf = new RandomAccessFile("myfile.txt");
4) RandomAccessFile raf = new RandomAccessFile(new File("myfile.txt"));

Question 2)

Which of the following statements are true?

1) The RandomAccessFile class allows you to move forwards and backwards without re-opening the file
2) An instance of RandomAccessFile may be used as a constructor for FileInputStream
3) The methods of RandomAccessFile do not throw exceptions
4) Creating a RandomAccessFile instance with a constructor willl throw an exception if the file does not exist.

Question 3)

Which of the following statements are true?

1) The FileInputStream can take either the name of a file or a an instance of the File class as a constructor
2) FileInputStream will throw an exception if the file name passed as a constructor does not exist
3) The methods of the FileInputStream are especially appropriate for manipulating text files
4) The delete method of the FileOutputStream class will remove a file from the operating system

Question 4)

Question 4)

What will happen when you attempt to compile and run the following code

```
import java.io.*;
public class Fos{
                    String s = new String("Hello");
                    public static void main(String argv[]){
                    Fos f = new Fos();
                    f.amethod();
                   }

                public void amethod(){
                       FileOutputStream fos = new FileOutputStream("Out.txt");
                       fos.write(10);
             }
}
```

1) Compile time error
2) Runtime error
3) Creation of a file called Out.txt containing the text "10"
4) Creation of a file called Out.txt

Question 5)

Which of the following statements are true?

1) The seek method of FileInputStream will set the position of the file pointer
2) The read method of FileInputStream will read bytes from a FileInputStream
3) The get method of FileInputStream will read bytes from a FileInputStream
4) A FileOutputStream can be closed using the close method

Answers

Answer to Question 1)

1) RandomAccessFile raf = new RandomAccessFile("myfile.txt","rw");

The RandomAccessFile is an anomoly in the Java I/O architecture. It descends directly from Object and is not part of the Streams architecture.

Answer to Question 2

1) The RandomAccessFile class allows you to move forwards and backwards without re-opening the file
4) Creating a RandomAccessFile instance with a constructor willl throw an exception if the file does not exist

Answer to Question 3)

1) The FileInputStream can take either the name of a file or a an instance of the File class as a constructor
2) FileInputStream will throw an exception if the file name passed as a constructor does not exist
Option 3 is a reasonable description of the Reader and Writer classes, the FileInput and Output clases are designed to read and write bytes rather than text.

Answer to Question 4)

1) Compile time error

This code will throw an error something like

Fos.java:10: Exception java.io.IOException must be caught, or it must be declare in the throws clause of this method.
FileOutputStream fos = new FileOutputStream("Out.txt");

Answer to Question 5)

2) The read method of FileInputStream can be used to read bytes from a FileInputStream
4) A FileOutputStream can be closed using the close method

Option one referring to a seek method is fairly implausible as the concept of file pointer is appropriate to the RandomAccessFile rather than the stream classes. The get method of option three is implausible as the prefix get is almost always follwed by what you are getting.

Other sources on this topic

This topic is covered in the Sun Tutorial at
http://java.sun.com/docs/books/tutorial/essential/io/

The JLS on Java IO a bit academic and bare
http://www.infospheres.caltech.edu/resources/langspec-1.0/javaio.doc.html

Richard Baldwin on I/O
http://home.att.net/~baldwin.rg/Intermediate/Java060.htm

Oreilly have published a book specifically about Java I/O It probably goes into more detail than is necesary for the Certificaiton exam but browsing the online samples might give you some insights. The book gets generally good reviews at www.amazon.com
http://www.oreilly.com/catalog/javaio/

Joyothi has some handy tables for the I/O classes at
http://www.geocities.com/SiliconValley/Network/3693/io.html

Last updated
14 Oct 2001

most recent version at www.jchq.net