Chapter 7 Fundamentals

7.1 Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.

Structure of source files:

- An empty file is a valid java src file.
- Package statement (if exists) should be the first statement (comments before it are ok!). Next should be the import statements (if any).
- Last, should be the class/interface declaration.
- Method declaration for the standard main:

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

- Main method can also be final, native, synchronized. No matter whether other declarations (like private, protected) work on your m/c, for the purpose of the exam, it should be public.

Variable Declarations and Identifiers:

Rules:

- A valid java identifier is composed of a sequence of java letters (this includes uppercase and lowercase ASCII latin letters and _ , $) and digits, the first of which must be a letter. (Valid: _123, a$2 NOT VALID: 123$, goto)
- It cannot be same as any java Keywords (eg. while, for, class etc) or literals (ie. true, false or null).
- Class names can serve as a valid identifier. eg. String String = "asd"; //This is valid.
- Pitfalls:
- If you have a main method like: public static int main(String[] args){ return 10; } (Note return type.) It will compile but the program will throw exception at runtime saying there is no main method. Same will be the case if you have: public static void main(String args){}
- A class without a main method may be run by JVM, if its base class has a valid main method.

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 conventions 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 those 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. Utility code is often kept in static methods, thus the Math class class has an entire set of utility methods such as random, sin, and round, and the primitive wrapper classes Integer, Double etc have static methods for manipulating the primitives they wrap, such as returning the matching int value from the string "2".

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. Iin the following example of a non static int three instances of the int 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 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"
```

Static methods cannot be overriden in a child class but they can be hidden.

Here is an example of some code that appears to show overriding of static methods

```
class Base{
        public static void stamethod(){
                System.out.println("Base");
        }
}
public class ItsOver extends Base{
        public static void main(String argv[]){
                ItsOver so = new ItsOver();
                so.stamethod();
        }
        public static void stamethod(){
                System.out.println("amethod in StaOver");
        }
}

This code will compile and output "amethod in StaOver"
```

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 or 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 void 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 overridden. 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

7.2 Given an example of a class and a command-line, determine the expected runtime behavior.

Consider this:

```
public class TestClass {
        public static void main(String[] args){
                System.out.println("Hello, World!");
        }
}
```

Points to remember:

- args will NEVER be null.
- If no argument is passed, args.length will be 0.
- If the above program is run with the command line: "java TestClass hello world", then args[0] will be "hello" and args[1] will be "world".
- UNLIKE IN C/C++, the word 'java' or the name of the class is not passed.
- Language keywords: Here's a list of Java's keywords. These words are reserved--you cannot use any of these words as names in your Java programs. true, false, and null are not keywords but they are reserved words, so you cannot use them as names in your programs either.

7.3 Determine the effect upon object references and primitive values when they are passed into methods that perform assignments or other modifying operations on the parameters.

VERY IMPORTANT FACT: In java EVERY THING is passed by value. For primitive, it's value is passed (as expected). For object, the value of it's reference is passed. Read a detailed example explanation here : Pass by value

```
void changeObjects(String str, StringBuffer sb) {
        str = "123";    // makes the reference str to point to new string object
                        // containing "123". you are changing the reference here.
        str = str + "123";      // strings are immutable. It will create a new
                                // string containing "abc123". The original "abc"
                                // will remain as it is.
        sb.append("123");       // changes the actual object itself. you are NOT
                                // changing the reference here.
}

....
String s = "abc";
StringBuffer sb = new StringBuffer("abc");

changeString(s, sb);

System.out.println(s);          //Will still print "abc".
System.out.println(sb);         //Will still print "abc123".
```

//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 straightforward 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
```

7.4 Given a code example, recognize the point at which an object becomes eligible for garbage collection, and determine what is and is not guaranteed by the garbage collection system. Recognize the behaviors of System.gc and finalization.

You can answer all the questions if you keep the following rules in mind

- Only thing guaranteed by the GC mechanism is : IF an object is really being destroyed, it's finalize() method would already have been called.
- Note: It doesn't say anything about when will an object be GCed etc.
- An object is eligible for garbage collection, if the only references to that object are from other objects that are also eligible for garbage collection.
- Note: It doesn't say anything about circular references. It depends on the actual JVM implementation.
- You CANNOT precisely say when the GC thread will run. Neither can you make the GC thread to run when you want.

  Note: You can call System.gc() etc. but this only requests the JVM to run the GC thread.

Some Points to Remember:

- You may set all the reference variables pointing to an object to null. This will enable the GC to collect this object. But that does not mean the object will really be GCed. It is possible that the GC thread may not run at all for the whole life of the program.
- finalize() Method: Signature : protected void finalize() throws Throwable { }
  It is used to release system resources like File handles, Network connections etc. But not memory. Memory can only be release by the GC thread.
- All objects have a finalize method as it is implemented in the Object class. But unlike constructors, finalize() does not call super class's finalize(). So, it is advisable (NOT REQUIRED) to put super.finalize() in the code of your finalize() method so as to give a chance to the super class to cleanup it's resources.
- The order in which finalize methods are called may not reflect the order in which objects are destroyed.
- It will be called ONLY ONCE for an object by the garbage collector. If any exception is thrown in finalize, the object is still eligible for garbage collection (depends on the GC mechanism).
- You can resurrect an object by creating an active reference to it in this method.
- You may call finalize() explicitly, but it would be just like another method call and will not release the memory
- finalize can be overloaded, but only the method with above mentioned signature will be called by the GC.

State the behavior that is guaranteed by the garbage collection system

How and when garbage collection occurs is left up to the implementation of the JVM, all you can know is that an object becomes eligible for garbage collection when there are no longer any references to it. At that stage, no variable points to the object, therefore your code has no way of accessing it and the object can safely be removed by the system.

Finalize Method

The finalize() method of a class is called before it is garbage collected. The method takes no arguments and must return void, and throws Throwable. Do not confuse the finalize() method (garbage collection) with the finally keyword (exception handling).

Write code that explicitly makes objects eligible for garbage collection

My definition of explicit differs, as I would categorize this as implicit:

```
Date d = new Date();
d = null;
```

There is no longer a reference pointing to the Date created on the first line, therefore that object is eligible for garbage collection. No matter what you do, there is no way your code could get back a reference to that object, so your code cannot be affect by its removal. If on the other hand you had passed the object into a Collection, or it was pointed to by some class, instance or local variable somewhere, then it is not ready for garbage collection (for example you could pass it to a method or constructor before setting d to null).

```
Date d = new Date();
d = new Date(0);
```

The new Date created on the second line holds the value midnight, January 1, 1970. The previous object referred to by d, which held the current Date is now eligable for garbage collection, but the new value of d (1/1/1970) is unaffected.

Recognize the point in a piece of source code at which an object becomes eligible for garbage collection

An object is eligible for garbage collection when there are no longer any references to it. This can happen if a variable is assigned to a newly created object, but later is assigned to null or a different object, so there is nothing pointing to the original object. Or when a method exits, all the local variables go out of scope, so all the objects are eligible for garbage collection, unless there is a reference to them somewhere outside the method.

Additional Notes For Garbage 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
```

A guaranteed behaviour: finalize

Java guarantees that the finalize method will be run before an object is Garbage collected. Note that unlike most other Garbage collection related behaviour this is guaranteed. But what exactly does a finalize method do?

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. Here is a quote from the JDK1.4 documentation on the finalize method.

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Because garbage collection is "non deterministic" ie you cannot predict when it will happen, you thus cannot predict exactly when the finalize method will run. You will probably be pleased to know that the exam does not expect you to know much (anything?) about the finalize method.

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 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.

Unreachability

An object becomes eligible for garbage collection when it becomes unreachable by any code. Two ways for this to happen are when it is explicitly set to null and when the reference that points to it is pointed to some other object. You may get an exam question that has some code that sets a reference to null and you have to work out where the object becomes eligible for garbage collection. This type of question should be easy. The other type where a reference gets pointed at another object is not quite so obvious. Take the following code example.

```
An object becomes eligible for garbage collection when it becomes
unreachable
```

```
class Base{
        String s;
        Base(String s){
                this.s = s;
        }
        public void setString(String s){
```

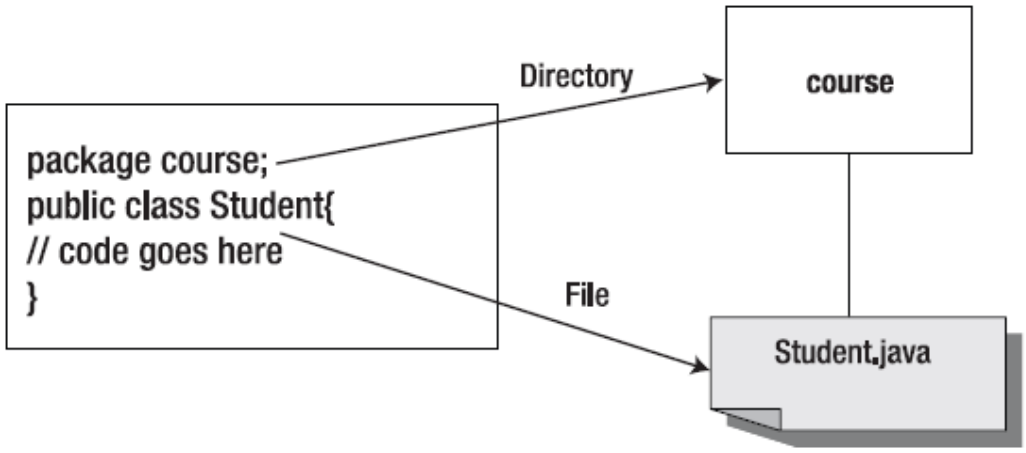
```
                this.s = s;
        }
}

public class UnReach{
        public static void main(String argv[]){
                UnReach ur = new UnReach();
                ur.go();
        }
        public void go(){
                Base b1 = new Base("One");
                b1.setString("");
                Base b2 = new Base("Two");
                b1 = b2;
        }
}
```

At what point does the object referenced by b1 become eligible for garbage collection.? Assuming you are ignoring the nonsense about setting the string to blank you will have worked out that when b1 is assigned to point to the same object as b2 then the original object that b1 pointed to is not reachable by any code.

7.5 Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully. Write code that correctly applies the appropriate operators including assignment operators (limited to: =, +=, -=), arithmetic operators (limited to: +, -, *, /, %, ++, --), relational operators (limited to: <, <=, >, >=, ==, !=), the instanceof operator, logical operators (limited to: &, |, ^, !, &&, ||), and the conditional operator (? :), to produce a desired result. Write code that determines the equality of two objects or two primitives.

A beginner in Java usually runs into compiler and execution errors related to finding the classes. These errors arise due to the confusion about the namespace. So, if it happens to you, rest assured that you are not the only one. The package name, the class name, and the classpath variable are the three players that you must be familiar with to avoid any confusion in the namespace area. You know from the previous section that the .class file name matches the name of the class. You can bundle related classes and interfaces into a group called a package. You store all the class files related to a package in a directory whose name matches the name of the package. How do you specify to which package a class belongs? You specify it in the source file in which you write the class by using the keyword package. There will be at most one package statement in a .java file. For example, consider a class Student defined in a file, as shown in figure bellow.



The file name of the file in which the class Student exists will be Student.java, which will exist inside a directory named course, which may exist anywhere on the file system. The qualified name for the class is course.Student, and the path name to it is course/Student.java.

You can use this class in the code by specifying its qualified name, as in this example:

```
course.Student student1 = new course.Student();
```

However, it is not convenient to use the qualified name of a class over and over again. The solution to this problem is to import the package in which the class exists, and use the simple name for the class, as shown here:

```
import course.Student;
```

You place the import statement in the file that defines the class in which you are going to use the Student class. The import statement must follow any package statement and must precede the first class defined in the file. After you have imported the package this way, you can use the Student class by its simple name:

```
Student student1 = new Student();
```

You can import all the classes in the package by using the wildcard character *:

```
import course.*;
```

However, you cannot use the wildcard character in specifying the names of the classes. For example, the following statement to refer to the class Student will generate a compiler error:

```
import course.Stud*;
```

You can have more than one import statement in a .java file. Bundling related classes and interfaces into a package offers the following advantages:

- It makes it easier to find and use classes.
- It avoids naming conflicts. Two classes with the same name existing in two different packages do not have a name conflict, as long as they are referenced by their fully qualified name.
- It provides access control. You will learn more about access control when access modifiers are discussed later in this chapter.

So, the files in a directory may be composed into a package by the declaration with keyword package:

```
package <PackageName>;
```

Then you may use this package in another file by the statement with keyword import:

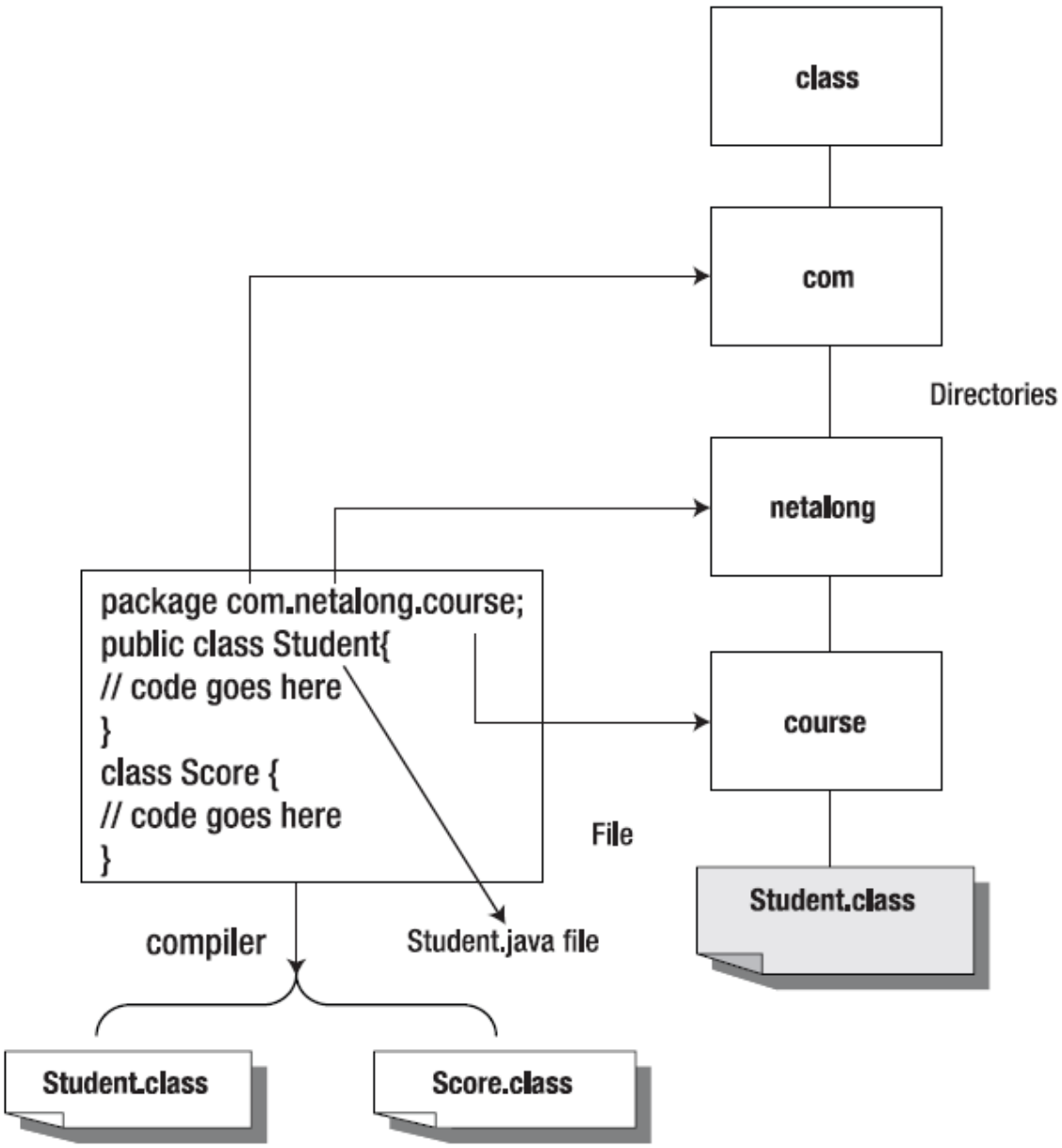
```
import <PackageName>;
```

In a file, package declaration must precede the import statement, which must precede the class definition. For example, the .java file with the following code will not compile because the import statement appears before the package statement:

```
import otherPackage;
package thisPackage;
class A { }
```

In a Java file, the package declaration, the import statement, and the class definition must appear in this order.

You need to manage your source and class files in such a way that the compiler and the interpreter can find all the classes and interfaces that your program uses. Companies usually follow the convention of stating the package names with the reversed domain name. For example, the full package name of course in a fictitious company netalong.com will be com.netalong.course. Each component of a package name corresponds to a directory. In our example, the directory com contains the directory netalong, which in turn contains the directory course, and the Student file is in the course directory. This relationship is shown in figure bellow.



The directory structure corresponding to the package name goes into a directory called the top-level directory. We assume in this example that the com directory is in the class directory. When you compile the source file Student.java, it will produce two class files, named Student.class and Score.class. It's a good practice to keep the class files separate from the source files. Let's assume that we put the class files in the class/com/netalong/course directory, and that the source and the class directories exist in the C:\app directory on a Microsoft Windows machine. So, the top-level directory for our package is

```
c:\app\class
```

When the compiler encounters a class name in your code, it must be able to find the class. In fact, both the compiler and the interpreter must be able to find the classes. As said earlier, they look for classes in each directory or a JAR (Java Archive) file listed in the classpath: an environment variable that you defined after installing JDK. In our example, the classpath must include this path name:

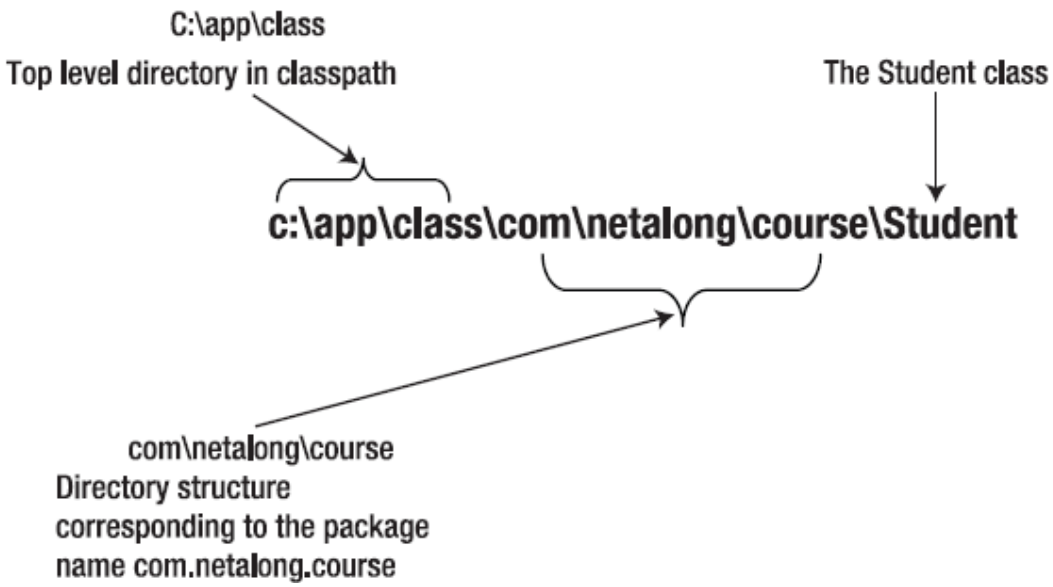
```
c:\app\class
```

A classpath is an environment variable whose value is a list of directories or JAR files in which the compiler and the interpreter searches for the class files. Following is an example of a classpath:

```
c:\jdk1.5.0_01; c:\jdk1.5.0_01\bin; c:\myclasses
```

Note that the directories are separated by a semicolon (;).

So, each directory listed in the classpath is a top-level directory that contains the package directories. The compiler and interpreter will construct the full path of a class by appending the package name to the top-level directory name (in the classpath) followed by the class name, as shown in figure bellow



The .java files and .class files live in directories. All the directories of an application can be compressed into what is called a JAR file.

The JAR Files

When the compiler (or the interpreter) encounters a class name in your code, it looks for that class in each directory or a JAR (Java Archive) file listed in the classpath. You can think of a JAR file as a tree of directories. Using the JAR facility of Java, the whole application (a number of source files and class files in a number of folders) can be compressed into one file, the JAR file. The file extension of a JAR file is .jar, and can be created with the jar command. For example, consider the following command:

```
jar cf myApp.jar topDir
```

This command will compress the whole directory tree (along with files in the directories) with topDir as the root directory into one file named myApp.jar. You can look at the list of directories and files in this JAR file by issuing the following command:

```
jar –tf myApp.jar
```

You can even execute your application contained in the JAR file by issuing the following command:

```
java –jar myApp.jar
```

When you specify the path to a JAR file, you must include the JAR file name at the end of the path, such as the following:

```
c:\jdk1.5.0\jre\lib\charsets.jar
```

To make the java –jar command work, you need to specify which class contains the main(…) method by adding an entry like the following to the MANIFEST.MF file:

```
Main-Class: MyClass
```

In this entry, MyClass is the name of the class (and also the name of a .class file) that contains the main(…) method.

To the contrary, when you mention a directory name in a path, all the files in the directory will automatically be included in the path. By including a JAR file in the classpath, you can use the classes in it that may be referred to by another application. Another way of including the external classes in your application is by importing them, as discussed earlier in the chapter. J2SE 5.0 introduces a special kind of import called static import.

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 a 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 be 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 ++ and –– Operators

You would be hard pressed to find a non trivial Java program that does not use the ++ or –– operators, and it can be easy to assume that you know all you need to know for the purposes of the exam. However these operators can be used in pre-increment or post increment. If you do not understand the difference you can loose exam points on an apparently easy question. To illustrate this, what do you think will be sent to the console when the following code is compiled and run?

```
public class PostInc{
        static int i=1;
        public static void main(String argv[]){
                System.out.println(i++);
        }
}
```

If you compile and run this code the output is 1 and not 2. This is because the ++ is placed after the number 1, and thus the incrementation (the adding of 1) will occur after the line is run. The same principle is true for the –– operator.

The bit shifting operators

Typical examples of where bit shifting is used "in the real world" is cryptography and low level manipulation of image files. You can do an awful lot of Java programming without ever having to shift a single bit yourself. 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. The exam generally presents at least on question on the bit shifting operators. If you are from a C++ background you can be mislead into thinking that all of your knowledge can transfer directly to the Java language.

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
```

Not only do you need to appreciate binary, you need to have a general grasp of the concept of "twos compliment" numbering system. With this system of representing numbers the leading bit indicates if a number is positive or negative or positive. So far so intuitive, but it starts to get strange when you

understand that numbering system works like a car odometer. Imagine every little wheel had a 1 or a zero on it. If you were to turn it back from showing

```
00000000 00000000 00000000 00000001
```

one more click it would show

```
11111111 11111111 11111111 11111111 11111111
```

This would represent -1. If you click it back one more place it would show

```
11111111 11111111 11111111 11111111 1111110
```

Those examples are slightly oversimplified. Until I studied for the Java Programmers exam I had assumed that twos compliment representation only referred to the use of the leading bit to indicate the sign part of the number. As you can see it is somewhat more complex than that. To help you understand the notation a little more I have written an apparently trivial program that will show the bit pattern for a number given on the command line. It could be much improved by breaking the bits into chunks of eight, but it is still handy for getting the general picture.

```
public class Shift{
        public static void main(String argv[]){
                int i = Integer.parseInt(argv[0]);
                System.out.println(Integer.toBinaryString(i));
        }
}
```

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.

Unsigned right shift >>> of a positive number

I will start with the unsigned right shift because it is probably the weirdest of the bit shifts and requires an understanding of twos complement number representation to fully understand. It gets extra weird when negative numbers are involved so I will start with positive numbers. The unsigned right shift operation treats a number as purely a bit pattern and ignores the special nature of the sign bit. Remember that once you start looking at a number as a sequence of bits any bit level manipulation can have some unexpected results when viewed as a normal number.

The unsigned right shift operation takes two operands, the first number is the number that will have its bits shifted and the number after the operand is the number of places to be shifted. Thus the following

```
3 >>> 1
```

Means that you will shift the bits in the number 3 one place to the right.

The twos complement numbering system means that the leading bit in a number indicates if it is positive or negative. If this value is zero the number is positive, if it is 1 it is negative. With the unsigned right shift the leading bit position is always filled with a zero. This means an unsigned right shift operation always results in a positive number.

If you think of the number 3 as being represented by

```
011
```

And you shift it one place to the right with

```
3 >> 1
```

you will end up with

```
001
```

Note that the bits that have new values moved into them "fall off the end" of the number and are effectively thrown away.

If you perform the shift two places to the right it will come as little surprise that the number becomes zero as the number zero is moved into all of the bit positions. If you keep increasing the number of places you are shifting by such as 6 places, 10 places 20 places you will find the result stays at zero as you might expect. If you persist however when you get to

```
3 >>> 32
```

The surprising result is 3. Why is this so?

Behind the scenes before the shift a mod 32 is performed on the operand. The modulus operator (indicated in java by the % character divides one number by another and returns the remainder. Whilst the mod operator is being performed on a number smaller than itself the original number is returned so whilst the number of places being shifted is less than 32 the mod operation is not noticed. Once you get to 32 places it starts to kick in.

Thus 32 % 32 returns zero as there is nothing left over and the number returned by the operation

```
3 >>> 32
```

is 3, ie 3 is shifted 0 places.

I did not find this at all intuitive at first so I wrote the following code

```
public class shift{
        static int i=2;
        public static void main(String argv[]){
                System.out.println(32 % 32);
                System.out.println( 3  >>> 32);
        }
}
```

The output of this code is

```
0
3
```

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

Unsigned shift >>> of a negative number

An unsigned shift of a negative number will generally result in a positive number. I say generally as an exception is if you shift by exactly 32 places you end up with the original number including the sign bit. As explained earlier, the reason you generally get a positive number is that any unsigned right shift replaces the leading sign bit with a zero which indicates a positive number.

The results of an unsigned shift of a negative number can seem very odd at times, take the following

```
System.out.println( -3 >>> 1);
```

You might think that this would result in a number such as

```
1
```

ie the sign bit is replaced by a zero resulting in a positive number and the bits are shifted one place to the right. This is not what happens, the actual result is

```
2147483646
```

Strange but true.

The reason behind this odd result is to do with the way twos complement number representation works. If you imagine the bits in a number represented by the wheels on the display of a car odometer, what happens when you count down from the largest possible number to zero, and then go to the first number below zero?. All of the digits are set to 1 including the sign bit to indicate a negative number. When you perform an unsigned right shift you are breaking this way of interpreting the numbers and treating the sign bit as just another number. So although you started off with a small negative number such as -3 in the example you end up with a large positive number. You may get a question on this in the exam that asks you to identify the result of an unsigned shift of a negative number. The one correct answer may seem very unlikely.

```
A unsigned right shift >>> by a positive amount of a small
negative number will result in a large positive number returned.
```

The signed shift operators << and >>

The << and >> operators set "new" bits to zero. Thus in the example

```
System.out.println(2 << 1)
```

This shift moves all bits in the number 1 two places to the left placing a zero in each place from the right.

Thus the value

```
010
```

becomes

```
100
```

Or decimal four. You can think of this operation as a repeated multiplication by two of the original number. Thus the result of

```
System.out.println(2 << 4)
```

is 32

Which you can think of as

```
2 * 2 = 4 (for the first place of the shift)
2 * 4 = 8 (for the second place of the shift)
2 * 8 = 16 (for the third place of the shift)
2 * 16 = 32 (for the fourth place of the shift)
```

This way of thinking all goes horribly wrong when you get to the point of bits "falling off the end". Thus the output of

```
System.out.println(2<<30)
```

is

```
-2147483648
```

This may seem horribly counter-intuitive, but if you think that the single bit representing the 2 has been moved to the left most position it now represents the largest negative number that can be stored as an integer. If you shift by one more place ie

```
System.out.println(2 << 31)
```

The result is zero as every bit position is now zero and the bit from the number 2 has fallen off the end to be discarded.

With the signed right shift, the left hand (new) bits take the value of the most significant bit before the shift (by contrast with the zero that is put in the new places with the left shift). This means that the right hand shift will not affect the sign of the resulting number.

```
2 >> 2;
```

This shift moves all bits in the number 2 two places to the right Thus

Thus the value

```
0010
```

Becomes

```
0000
```

Or decimal zero.

This is the equivalent of performing a repeated integer division, in this case resulting in zeros in every position.

```
The signed right shift operation >> results in a number with the
same sign bit
```

Operator Precedence

Operator precedence is the order of priority in which operators get performed. The following table is a summary of the operator precedence.

Operator Precedence

()

++expr --expr +expr -expr ~ !

* / %

+ -

<< >> >>>

< > <= >= instanceof

== !=

&

^

|

&&

||

? :

= += -= *= /= %= &= ^= |= <<= >>= >>>=

I tems on the same row have the same precedence. Generally in real world programming you will indicate the order that operators are expected to be performed by surrounding expressions with braces. This means you can get by without actually learning the order of precedence and it should make it obvious for other programmers who read your code. However it is possible that you may get questions in the exam that depend on understanding operator precedence, particularly on the common operators +, -, *.

If the idea of operator precedence doesn't mean much to you, try to work out what the output of the following code will be.

```
public class OperPres{
        public static void main(String argv[]){
                System.out.println(2 + 2 * 2);
                System.out.println(2 + (2 * 2));
                System.out.println(8 / 4 + 4);
                System.out.println(8 /(4 +4));
                int i = 1;
                System.out.println(i++ * 2);
        }
}
```

Does the first statement 2 + 2 * 2 mean add 2 + 2 and then times the result by 2 resulting in an output of 8, or does it mean multiply 2 * 2 and then add 2 to give a result of 6. Similar questions can be asked of the other calculations. The output of this program by the way is 66612.

The & | && and || operators

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.
