Chapter 5 OO Concepts

5.1 Develop code that implements tight encapsulation, loose coupling, and high cohesion in classes, and describe the benefits.

Encapsulation

Imagine you wrote the code for a class, and another dozen programmers from your company all wrote programs that used your class. Now imagine that later on, you didn't like the way the class behaved, because some of its instance variables were being set (by the other programmers from within their code) to values you hadn't anticipated. Their code brought out errors in your code. (Relax, this is just hypothetical.) Well, it is a Java program, so you should be able just to ship out a newer version of the class, which they could replace in their programs without changing any of their own code.

This scenario highlights two of the promises/benefits of Object Orientation: flexibility and maintainability. But those benefits don't come automatically. You have to do something. You have to write your classes and code in a way that supports flexibility and maintainability. So what if Java supports OO? It can't design your code for you. For example, imagine if you made your class with public instance variables, and those other programmers were setting the instance variables directly, as the following code demonstrates:

```
public class BadOO {
        public int size;
        public int weight;
        ...
}
public class ExploitBadOO {
        public static void main (String [] args) {
                BadOO b = new BadOO();
                b.size = -5; // Legal but bad!!
        }
}
```

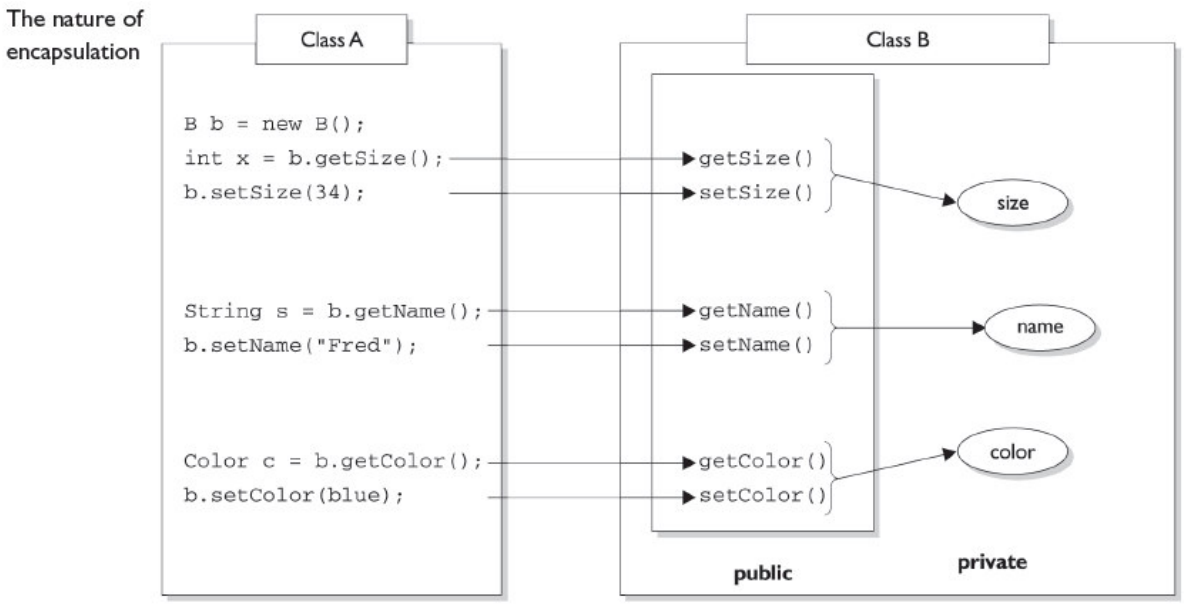
And now you're in trouble. How are you going to change the class in a way that lets you handle the issues that come up when somebody changes the size variable to a value that causes problems? Your only choice is to go back in and write method code for adjusting size (a setSize(int a) method, for example), and then protect the size variable with, say, a private access modifier. But as soon as you make that change to your code, you break everyone else's!

The ability to make changes in your implementation code without breaking the code of others who use your code is a key benefit of encapsulation. You want to hide implementation details behind a public programming interface. By interface, we mean the set of accessible methods your code makes available for other code to call—in other words, your code's API. By hiding implementation details, you can rework your method code (perhaps also altering the way variables are used by your class) without forcing a change in the code that calls your changed method.

If you want maintainability, flexibility, and extensibility (and of course, you do), your design must include encapsulation. How do you do that?

- Keep instance variables protected (with an access modifier, often private).
- Make public accessor methods, and force calling code to use those methods rather than directly accessing the instance variable.
- For the methods, use the JavaBeans naming convention of set<someProperty> and get<someProperty>.

Figure bellow illustrates the idea that encapsulation forces callers of our code to go through methods rather than accessing variables directly.



Class A cannot access Class B instance variable data without going through getter and setter methods. Data is marked private; only the accessor methods are public.

We call the access methods getters and setters although some prefer the fancier terms accessors and mutators. (Personally, we don't like the word "mutate".) Regardless of what you call them, they're methods that other programmers must go through in order to access your instance variables. They look simple, and you've probably been using them forever:

```
public class Box {
        // protect the instance variable; only an instance
        // of Box can access it " d " "dfdf"
        private int size;
        // Provide public getters and setters
        public int getSize() {
                return size;
        }
        public void setSize(int newSize) {
                size = newSize;
        }
}
```

Wait a minute...how useful is the previous code? It doesn't even do any validation or processing. What benefit can there be from having getters and setters that add no additional functionality? The point is, you can change your mind later, and add more code to your methods without breaking your API. Even if today you don't think you really need validation or processing of the data, good OO design dictates that you plan for the future. To be safe, force calling code to go through your methods rather than going directly to instance variables. Always. Then you're free to rework your method implementations later, without risking the wrath of those dozen programmers who know where you live.

Coupling and Cohesion

These two topics, coupling and cohesion, have to do with the quality of an OO design. In general, good OO design calls for loose coupling and shuns tight coupling, and good OO design calls for high cohesion, and shuns low cohesion. As with most OO design discussions, the goals for an application are

- Ease of creation
- Ease of maintenance
- Ease of enhancement

Coupling

Let's start by making an attempt at a definition of coupling. Coupling is the degree to which one class knows about another class. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled…that's a good thing. If, on the other hand, class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter…not a good thing. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

Using this second scenario, imagine what happens when class B is enhanced. It's quite possible that the developer enhancing class B has no knowledge of class A, why would she? Class B's developer ought to feel that any enhancements that don't break the class's interface should be safe, so she might change some non-interface part of the class, which then causes class A to break.

At the far end of the coupling spectrum is the horrible situation in which class A knows non-API stuff about class B, and class B knows non-API stuff about class A… this is REALLY BAD. If either class is ever changed, there's a chance that the other class will break. Let's look at an obvious example of tight coupling, which has been enabled by poor encapsulation:

```
class DoTaxes {
        float rate;
        float doColorado() {
                SalesTaxRates str = new SalesTaxRates();
                rate = str.salesRate;  // ouch this should be a method call:
                                       // rate = str.getSalesRate("CO");
                                       // do stuff with rate
        }
}

class SalesTaxRates {
        public float salesRate; // should be private
        public float adjustedSalesRate; // should be private
        public float getSalesRate(String region) {
                salesRate = new DoTaxes().doColorado(); // ouch again!
                                            // do region-based calculations
                return adjustedSalesRate;
        }
}
```

All non-trivial OO applications are a mix of many classes and interfaces working together. Ideally, all interactions between objects in an OO system should use the APIs, in other words, the contracts, of the objects' respective classes. Theoretically, if all of the classes in an application have well-designed APIs, then it should be possible for all interclass interactions to use those APIs exclusively. As we discussed earlier in this chapter, an aspect of good class and API design is that classes should be well encapsulated. The bottom line is that coupling is a somewhat subjective concept. Because of this, the exam will test you on really obvious examples of tight coupling; you won't be asked to make subtle judgment calls.

Cohesion

While coupling has to do with how classes interact with each other, cohesion is all about how a single class is designed. The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose. Keep in mind that cohesion is a subjective concept. The more focused a class is, the higher its cohesiveness—a good thing. The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes. Let's take a look at a pseudo-code example:

```
class BudgetReport {
```

```
        void connectToRDBMS(){ }
        void generateBudgetReport() { }
        void saveToFile() { }
        void print() { }
}
```

Now imagine your manager comes along and says, "Hey you know that accounting application we're working on? The clients just decided that they're also going to want to generate a revenue projection report, oh and they want to do some inventory reporting also. They do like our reporting features however, so make sure that all of these reports will let them choose a database, choose a printer, and save generated reports to data files..." Ouch!

Rather than putting all the printing code into one report class, we probably would have been better off with the following design right from the start:

```
class BudgetReport {
        Options getReportingOptions() { }
        void generateBudgetReport(Options o) { }
}

class ConnectToRDBMS {
        DBconnection getRDBMS() { }
}

class PrintStuff {
        PrintOptions getPrintOptions() { }
}

class FileSaver {
        SaveOptions getFileSaveOptions() { }
}
```

This design is much more cohesive. Instead of one class that does everything, we've broken the system into four main classes, each with a very specific, or cohesive, role. Because we've built these specialized, reusable classes, it'll be much easier to write a new report, since we've already got the database connection class, the printing class, and the file saver class, and that means they can be reused by other classes that might want to print a report.

5.2 Given a scenario, develop code that demonstrates the use of polymorphism. Further, determine when casting will be necessary and recognize compiler vs. runtime errors related to object reference casting.

Remember, any Java object that can pass more than one IS-A test can be considered polymorphic. Other than objects of type Object, all Java objects are polymorphic in that they pass the IS-A test for their own type and for class Object.

Remember that the only way to access an object is through a reference variable, and there are a few key things to remember about references:

- A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).
- A reference is a variable, so it can be reassigned to other objects, (unless the reference is declared final).
- A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
- A reference variable can refer to any object of the same type as the declared reference, or—this is the big one—it can refer to any subtype of the declared type!
- A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that implements the interface.

Earlier we created a GameShape class that was extended by two other classes (see objectives 5.5), PlayerPiece and TilePiece. Now imagine you want to animate some of the shapes on the game board. But not all shapes can be animatable, so what do you do with class inheritance?

Could we create a class with an animate() method, and have only some of the GameShape subclasses inherit from that class? If we can, then we could have PlayerPiece, for example, extend both the GameShape class and Animatable class, while the TilePiece would extend only GameShape. But no, this won't work! Java supports only single inheritance! That means a class can have only one immediate superclass. In other words, if PlayerPiece is a class, there is no way to say something like this:

```
class PlayerPiece extends GameShape, Animatable { // NO!
        // more code
}
```

A class cannot extend more than one class. That means one parent per class. A class can have multiple ancestors, however, since class B could extend class A, and class C could extend class B, and so on. So any given class might have multiple classes up its inheritance tree, but that's not the same as saying a class directly extends two classes.

Some languages (like C++) allow a class to extend more than one other class. This capability is known as "multiple inheritance." The reason that Java's creators chose not to allow multiple inheritance is that it can become quite messy. In a nutshell, the problem is that if a class extended two other classes, and both superclasses had, say, a doStuff() method, which version of doStuff() would the subclass inherit? This issue can lead to a scenario known as the "Deadly Diamond of Death," because of the shape of the class diagram that can be created in a multiple inheritance design. The diamond is formed when classes B and C both extend A, and both B and C inherit a method from A. If class D extends both B and C, and both B and C have overridden the method in A, class D has, in theory, inherited two different implementations of the same method. Drawn as a class diagram, the shape of the four classes looks like a diamond.

So if that doesn't work, what else could you do? You could simply put the animate() code in GameShape, and then disable the method in classes that can't be animated. But that's a bad design choice for many reasons, including it's more errorprone, it makes the GameShape class less cohesive (more on cohesion in a minute), and it means the GameShape API "advertises" that all shapes can be animated, when in fact that's not true since only some of the GameShape subclasses will be able to successfully run the animate() method.

So what else could you do? You already know the answer—create an Animatable interface, and have only the GameShape subclasses that can be animated implement that interface. Here's the interface:

```
public interface Animatable {
        public void animate();
}
```

And here's the modified PlayerPiece class that implements the interface:

```
class PlayerPiece extends GameShape implements Animatable {
        public void movePiece() {
                System.out.println("moving game piece");
        }
        public void animate() {
                System.out.println("animating...");
        }
        // more code
}
```

So now we have a PlayerPiece that passes the IS-A test for both the GameShape class and the Animatable interface. That means a PlayerPiece can be treated polymorphically as one of four things at any given time, depending on the declared type of the reference variable:

- An Object (since any object inherits from Object)
- A GameShape (since PlayerPiece extends GameShape)
- A PlayerPiece (since that's what it really is)
- An Animatable (since PlayerPiece implements Animatable)

The following are all legal declarations. Look closely:

```
PlayerPiece player = new PlayerPiece();
Object o = player;
GameShape shape = player;
Animatable mover = player;
```

There's only one object here—an instance of type PlayerPiece—but there are four different types of reference variables, all referring to that one object on the heap. Pop quiz: which of the preceding reference variables can invoke the display() method? Hint: only two of the four declarations can be used to invoke the display() method.

Remember that method invocations allowed by the compiler are based solely on the declared type of the reference, regardless of the object type. So looking at the four reference types again—Object, GameShape, PlayerPiece, and Animatable— which of these four types know about the display() method?

You guessed it—both the GameShape class and the PlayerPiece class are known (by the compiler) to have a display() method, so either of those reference types can be used to invoke display(). Remember that to the compiler, a PlayerPiece IS-A GameShape, so the compiler says, "I see that the declared type is PlayerPiece, and since PlayerPiece extends GameShape, that means PlayerPiece inherited the display() method. Therefore, PlayerPiece can be used to invoke the display() method."

Which methods can be invoked when the PlayerPiece object is being referred to using a reference declared as type Animatable? Only the animate() method. Of course the cool thing here is that any class from any inheritance tree can also implement Animatable, so that means if you have a method with an argument declared as type Animatable, you can pass in PlayerPiece objects, SpinningLogo objects, and anything else that's an instance of a class that implements Animatable. And you can use that parameter (of type Animatable) to invoke the animate() method, but not the display() method (which it might not even have), or anything other than what is known to the compiler based on the reference type. The compiler always knows, though, that you can invoke the methods of class Object on any object, so those are safe to call regardless of the reference—class or interface— used to refer to the object.

We've left out one big part of all this, which is that even though the compiler only knows about the declared reference type, the JVM at runtime knows what the object really is. And that means that even if the PlayerPiece object's display() method is called using a GameShape reference variable, if the PlayerPiece overrides the display() method, the JVM will invoke the PlayerPiece version! The JVM looks at the real object at the other end of the reference, "sees" that it has overridden the method of the declared reference variable type, and invokes the method of the object's actual class. But one other thing to keep in mind:

Polymorphic method invocations apply only to instance methods. You can always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the ONLY things that are dynamically selected based on the actual object (rather than the reference type) are instance methods. Not static methods. Not variables. Only overridden instance methods are dynamically invoked based on the real object's type.

Since this definition depends on a clear understanding of overriding, and the distinction between static methods and instance methods, we'll cover those next.

5.3 Explain the effect of modifiers on inheritance with respect to constructors, instance or static variables, and instance or static methods.

Access Modifier	Class	Subclass	Package	World
private	Yes	No	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes
Default	Yes	No	Yes	No

The static Modifier

The static modifier can be applied to variables, methods, and a block of code inside a method. The static elements of a class are visible to all the instances of the class. As a result, if one instances of the class makes a change to a static element, all the instances will see that change. Consider listing bellow, the variable instanceCounter is declared static in line 2, and another variable, counter, is not declared static in line 3. When an instance of the class StaticExample is created, both variables are incremented by one (lines 5 and 6). Each instance has its own copy of the variable counter, but they share the variable instanceCounter. A static variable belongs to the class, and not to a specific instance of the class, and therefore is initialized when the class is loaded. A static variable may be referenced by an instance of the class (lines 13 and 14) in which it is declared, or by the class name itself (line 15).

```
RunStaticExample.java
1. class StaticExample {
2.      static int instanceCounter = 0;
3.      int counter = 0;
4.      StaticExample() {
5.              instanceCounter++;
6.              counter++;
7.      }
8. }
9. class RunStaticExample {
10.     public static void main(String[] args) {
11.             StaticExample se1 = new StaticExample();
12.             StaticExample se2 = new StaticExample();
13.             System.out.println("Value of instanceCounter for se1: " +
                                                        se1.instanceCounter);
14.             System.out.println("Value of instanceCounter for se2: " +
                                                        se2.instanceCounter);
15.             System.out.println("Value of instanceCounter: " +
                                                  StaticExample.instanceCounter);
16.             System.out.println("Value of counter for se1: " + se1.counter);
17.             System.out.println("Value of counter for se2: " + se2.counter);
18.     }
19.}
```

The following is the output:

```
Value of instanceCounter for se1: 2
Value of instanceCounter for se2 2
Value of instanceCounter: 2
Value of counter for se1: 1
Value of counter for se2 1
```

The following line of code outside the StaticExample class will set the value of instanceCounter to 100 for all the instances of the class StaticExample:

```
StaticExample.instanceCounter = 100;
```

Just like a static variable, a static method also belongs to the class in which it is defined, and not to a specific instance of the class. Therefore, a static method can only access the static members of the class. In other words, a method declared static in a class cannot access the non-static variables and methods of the class. Because a static method does not belong to a particular instance of the class in which it is defined, it can be called even before a single instance of the class exists. For example, every Java application has a method main(…), which is the entry point for the application execution. It is executed without instantiating the class in which it exists. Also, a static method may not be overridden as non-static and vice versa.

A static method cannot access the non-static variables and methods of the class in which it is defined.

Also, a static method cannot be overridden as non-static. In addition to static variables and static methods, a class may have a static code block that does not belong to any method, but only to the class. For example, you may like to execute a task before the class is instantiated, or even before the method main(…) is called. In such a situation, the static code block will help because it will be executed when the class is loaded. Consider listing bellow, when you run the application RunStaticCodeExample, the static variable counter (line 2) is initialized, and the static code block (lines 3 to 6) are executed at the class StaticCodeExample load time. Then, the method main(…) is executed.

```
RunStaticCodeExample.java
1. class StaticCodeExample {
2.      static int counter=0;
3.      static {
4.              counter++;
5.              System.out.println("Static Code block: counter: " + counter);
6.      }
7.      StaticCodeExample() {
8.              System.out.println("Constructor: counter: " + counter);
9.      }
10.     static {
11.             System.out.println("This is another static block");
12.     }
13.}
14. public class RunStaticCodeExample {
15.     public static void main(String[] args) {
16.             StaticCodeExample sce = new StaticCodeExample();
17.             System.out.println("main: " + sce.counter);
18.     }
19.}
```

The output:

```
Static Code block: counter: 1
This is another static block
Constructor: counter: 1
main: counter: 1
```

This output demonstrates that the static code block is executed exactly once, and before the class constructor is executed—that is, at the time the class is loaded. It will never be executed again during the execution lifetime of the application. Note that all the static blocks will be executed in order before the class initialization regardless of where they are located in the class.

So, remember these points about the static modifier:

- The static elements (variables, methods, and code fragments) belong to the class and not to a particular instance of the class.
- Any change in a static variable of a class is visible to all the instances of the class.
- A static variable is initialized at class load time. Also, a static method and a static code fragment are executed at class load time.
- A static method of a class cannot access the non-static members of the class.

- You cannot declare the following elements as static: constructor, class (that is, the top-level class), interface, inner class (the top-level nested class can be declared static), inner class methods and instance variables, and local variables.
- It is easier to remember what you can declare static: top-level class members (methods and variables), the top-level nested class, and code fragments.

The static modifier cannot be applied to a top-level class or a class constructor. However, you can apply the final modifier to a class, which means the class cannot be extended. You may face the opposite situation, where you want the class to be extended before it can be instantiated. This situation is handled by the abstract modifier.

Modifiers: The Big Picture

Summary of Modifiers Used by Java Classes and Class Members

Modifier	Top-Level Class	Variable	Method	Constructor	Code Block
public	Yes	Yes	Yes	Yes	No
private	No	Yes	Yes	Yes	No
protected	No	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	Yes	N/A
final	Yes	Yes	Yes	No	No
static	No	Yes	Yes	No	Yes
abstract	Yes	No	Yes	No	No
native	No	No	Yes	No	No
transient	No	Yes	No	No	No
volatile	No	Yes	No	No	No
synchronized	No	No	Yes	No	Yes

5.4 Given a scenario, develop code that declares and/or invokes overridden or overloaded methods and code that declares and/or invokes superclass, overridden, or overloaded constructors.

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 overridden 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 is a one of the ways in which Java implements one of the key concepts of Object orientation, polymorphism. Polymorphism is a ten guinea made up word that is constructed from Ply meaning "many" and "morphism" implying meaning. Thus a overloading allows the same method name to have multiple meanings or uses. Overloading of methods is a compiler trick to allow you to use the same name to perform different actions depending on parameters. It takes advantage of the fact that Java resolves the actual method that gets called at runtime rather than compile time.

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.

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 grandaddy 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 overridden 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 gues 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.

5.5 Develop code that implements "is-a" and/or "has-a" relationships.

Inheritance is everywhere in Java. It's safe to say that it's almost (almost?) impossible to write even the tiniest Java program without using inheritance. In order to explore this topic we're going to use the instanceof operator. Remember that instanceof returns true if the reference variable being tested is of the type being compared to. This code:

```
class Test {
        public static void main(String [] args) {
                Test t1 = new Test();
                Test t2 = new Test();
                if (!t1.equals(t2))
                        System.out.println("they're not equal");
                if (t1 instanceof Object)
                        System.out.println("t1's an Object");
        }
}
```

Produces the output:

```
they're not equal
t1's an Object
```

Where did that equals method come from? The reference variable t1 is of type Test, and there's no equals method in the Test class. Or is there? The second if test asks whether t1 is an instance of class Object, and because it is (more on that soon), the if test succeeds.

Hold on…how can t1 be an instance of type Object, we just said it was of type Test? I'm sure you're way ahead of us here, but it turns out that every class in Java is a subclass of class Object, (except of course class Object itself). In other words, every class you'll ever use or ever write will inherit from class Object. You'll always have an equals method, a clone method, notify, wait, and others, available to use. Whenever you create a class, you automatically inherit all of class Object's methods.

Why? Let's look at that equals method for instance. Java's creators correctly assumed that it would be very common for Java programmers to want to compare instances of their classes to check for equality. If class Object didn't have an equals method, you'd have to write one yourself; you and every other Java programmer. That one equals method has been inherited billions of times. (To be fair, equals has also been overridden billions of times, but we're getting ahead of ourselves.)

For the exam you'll need to know that you can create inheritance relationships in Java by extending a class. It's also important to understand that the two most common reasons to use inheritance are

- To promote code reuse
- To use polymorphism

Let's start with reuse. A common design approach is to create a fairly generic version of a class with the intention of creating more specialized subclasses that inherit from it. For example:

```
class GameShape {
        public void displayShape() {
                System.out.println("displaying shape");
        }
        // more code
}

class PlayerPiece extends GameShape {
        public void movePiece() {
                System.out.println("moving game piece");
        }
        // more code
}

public class TestShapes {
```

```
        public static void main (String[] args) {
                PlayerPiece shape = new PlayerPiece();
                shape.displayShape();
                shape.movePiece();
        }
}
```

Outputs:

```
displaying shape
moving game piece
```

Notice that the PlayingPiece class inherits the generic display() method from the less-specialized class GameShape, and also adds its own method, movePiece(). Code reuse through inheritance means that methods with generic functionality (like display())—that could apply to a wide range of different kinds of shapes in a game—don't have to be reimplemented. That means all specialized subclasses of GameShape are guaranteed to have the capabilities of the more generic superclass. You don't want to have to rewrite the display() code in each of your specialized components of an online game.

But you knew that. You've experienced the pain of duplicate code when you make a change in one place and have to track down all the other places where that same (or very similar) code exists.

The second (and related) use of inheritance is to allow your classes to be accessed polymorphically—a capability provided by interfaces as well, but we'll get to that in a minute. Let's say that you have a GameLauncher class that wants to loop through a list of different kinds of GameShape objects, and invoke display() on each of them. At the time you write this class, you don't know every possible kind of GameShape subclass that anyone else will ever write. And you sure don't want to have to redo your code just because somebody decided to build a Dice shape six months later.

The beautiful thing about polymorphism ("many forms") is that you can treat any subclass of GameShape as a GameShape. In other words, you can write code in your GameLauncher class that says, "I don't care what kind of object you are as long as you inherit from (extend) GameShape. And as far as I'm concerned, if you extend GameShape then you've definitely got a display() method, so I know I can call it." Imagine we now have two specialized subclasses that extend the more generic GameShape class, PlayerPiece and TilePiece:

```
class GameShape {
        public void displayShape() {
                System.out.println("displaying shape");
        }
        // more code
}

class PlayerPiece extends GameShape {
        public void movePiece() {
                System.out.println("moving game piece");
        }
        // more code
}

class TilePiece extends GameShape {
        public void getAdjacent() {
                System.out.println("getting adjacent tiles");
        }
        // more code
}
```

Now imagine a test class has a method with a declared argument type of GameShape, that means it can take any kind of GameShape. In other words, any subclass of GameShape can be passed to a method with an argument of type GameShape. This code

```
public class TestShapes {
```

```
        public static void main (String[] args) {
                PlayerPiece player = new PlayerPiece();
                TilePiece tile = new TilePiece();
                doShapes(player);
                doShapes(tile);
        }
        public static void doShapes(GameShape shape) {
                shape.displayShape();
        }
}
```

Outputs:

```
displaying shape
displaying shape
```

The key point is that the doShapes() method is declared with a GameShape argument but can be passed any subtype (in this example, a subclass) of GameShape. The method can then invoke any method of GameShape, without any concern for the actual runtime class type of the object passed to the method. There are implications, though. The doShapes() method knows only that the objects are a type of GameShape, since that's how the parameter is declared. And using a reference variable declared as type GameShape—regardless of whether the variable is a method parameter, local variable, or instance variable—means that only the methods of GameShape can be invoked on it. The methods you can call on a reference are totally dependent on the declared type of the variable, no matter what the actual object is, that the reference is referring to. That means you can't use a GameShape variable to call, say, the getAdjacent() method even if the object passed in is of type TilePiece.

IS-A and HAS-A Relationships

For the exam you need to be able to look at code and determine whether the code demonstrates an IS-A or HAS-A relationship. The rules are simple, so this should be one of the few areas where answering the questions correctly is almost a no-brainer.

IS-A

In OO, the concept of IS-A is based on class inheritance or interface implementation. IS-A is a way of saying, "this thing is a type of that thing." For example, a Mustang is a type of horse, so in OO terms we can say, "Mustang IS-A Horse." Subaru IS-A Car. Broccoli IS-A Vegetable (not a very fun one, but it still counts). You express the IS-A relationship in Java through the keywords extends (for class inheritance) and implements (for interface implementation).

```
public class Car {
        // Cool Car code goes here
}
public class Subaru extends Car {
        // Important Subaru-specific stuff goes here
        // Don't forget Subaru inherits accessible Car members which
        // can include both methods and variables.
}
```

A Car is a type of Vehicle, so the inheritance tree might start from the Vehicle class as follows:

```
public class Vehicle { ... }
public class Car extends Vehicle { ... }
public class Subaru extends Car { ... }
```

In OO terms, you can say the following:

Vehicle is the superclass of Car.
Car is the subclass of Vehicle.
Car is the superclass of Subaru.

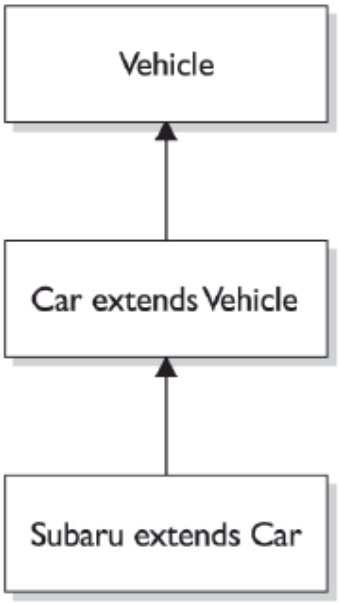
Subaru is the subclass of Vehicle.
Car inherits from Vehicle.
Subaru inherits from both Vehicle and Car.
Subaru is derived from Car.
Car is derived from Vehicle.
Subaru is derived from Vehicle.
Subaru is a subtype of both Vehicle and Car.

Returning to our IS-A relationship, the following statements are true:

"Car extends Vehicle" means "Car IS-A Vehicle."
"Subaru extends Car" means "Subaru IS-A Car."

And we can also say:

"Subaru IS-A Vehicle" because a class is said to be "a type of" anything further up in its inheritance tree. If the expression (Foo instanceof Bar) is true, then class Foo IS-A Bar, even if Foo doesn't directly extend Bar, but instead extends some other class that is a subclass of Bar. Figure 2-2 illustrates the inheritance tree for Vehicle, Car, and Subaru. The arrows move from the subclass to the superclass. In other words, a class' arrow points toward the class from which it extends.



HAS-A

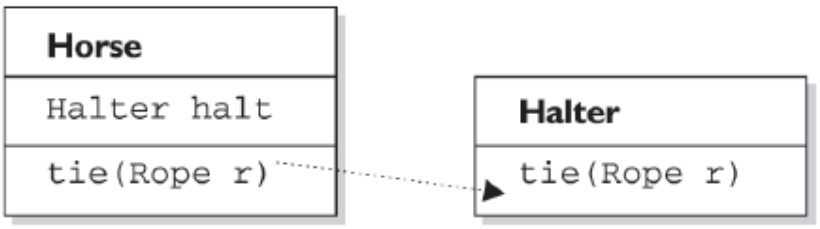
HAS-A relationships are based on usage, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B. For example, you can say the following,

A Horse IS-A Animal. A Horse HAS-A Halter.

The code might look like this:

```
public class Animal { }
        public class Horse extends Animal {
        private Halter myHalter;
}
```

In the preceding code, the Horse class has an instance variable of type Halter, so you can say that "Horse HAS-A Halter." In other words, Horse has a reference to a Halter. Horse code can use that Halter reference to invoke methods on the Halter, and get Halter behavior without having Halter-related code (methods) in the Horse class itself. Figure bellow illustrates the HAS-A relationship between Horse and Halter.



Horse class has a Halter, because Horse declares an instance variable of type Halter. When code invokes `tie()` on a Horse instance, the Horse invokes `tie()` on the Horse object's Halter instance variable.

HAS-A relationships allow you to design classes that follow good OO practices by not having monolithic classes that do a gazillion different things. Classes (and their resulting objects) should be specialists. As our friend Andrew says, "specialized classes can actually help reduce bugs." The more specialized the class, the more likely it is that you can reuse the class in other applications. If you put all the Halter-related code directly into the Horse class, you'll end up duplicating code in the Cow class, UnpaidIntern class, and any other class that might need Halter behavior. By keeping the Halter code in a separate, specialized Halter class, you have the chance to reuse the Halter class in multiple applications.

Users of the Horse class (that is, code that calls methods on a Horse instance), think that the Horse class has Halter behavior. The Horse class might have a tie(LeadRope rope) method, for example. Users of the Horse class should never have to know that when they invoke the tie() method, the Horse object turns around and delegates the call to its Halter class by invoking myHalter.tie(rope). The scenario just described might look like this:

```
public class Horse extends Animal {
        private Halter myHalter;
        public void tie(LeadRope rope) {
                myHalter.tie(rope); // Delegate tie behavior to the
                // Halter object
        }
}
public class Halter {
        public void tie(LeadRope aRope) {
                // Do the actual tie work here
        }
}
```

In OO, we don't want callers to worry about which class or which object is actually doing the real work. To make that happen, the Horse class hides implementation details from Horse users. Horse users ask the Horse object to do things (in this case, tie itself up), and the Horse will either do it or, as in this example, ask something else to do it. To the caller, though, it always appears that the Horse object takes care of itself. Users of a Horse should not even need to know that there is such a thing as a Halter class.