Classes

access, packages, and information hiding

Access

Access modifiers specify which elements of a class are accessible to code that is outside of that class. This allows the writer of the class to regulate how the class is used. Java provides four access modifiers to the programmer:

`public`
`private`
`protected`
package-private (default)

At the top-level – i.e. the outer class level – one can use either `public` or *package-private* modifiers. In other words, all classes of the type that we have dealt with may be either `public` or *package-private*. The following declarations are valid:

```
class A {
    ...
}

public class A {
    ...
}
```

The differences between these two declarations will be made clear below, when packages are discussed. No other modifier – i.e. `private` or `protected` – may be used to modify a top-level class. These modifiers can be used with nested or inner classes; however, we will not concern ourselves with those types of classes.

At the member level – i.e. variables, methods, constructors – one can use any of the modifiers. All of the following are valid:

```
class A {
    A(...) { ... }                  // this is package-private
    public A(...) { ... }
    private A(...) { ... }
    protected A(...) { ... }

    void f(...) { ... }             // this is package-private
    public void f(...) { ... }
    private void f(...) { ... }
    protected void f(...) { ... }

    int w;                          // this is package-private
    public int x;
    private int y;
    protected int z;
}
```

Note that *package-private* is not an explicit modifier, it is implied by the lack of a modifier. That is, we do not write *package-private* into the code, it is simply the default access level. So, when a class, variable, or method is not preceded by an access modifier, it is *package-private*.

Furthermore, each of the access levels can be used in conjunction with other Java Keywords – e.g. `static`, `final`, etc.:

```
class A {
    public static void f(...) { ... }
    static private void f(...) { ... }
    protected static void f(...) { ... }

    final private int x;
    public static final int y;
    protected static int z;
}
```

Notice that the keywords may occur in any order. The only requirement is that identifiers – i.e. names of variables and methods – must always be immediately preceded by the name of a type.

Let's consider the access modifiers in the context in which we have used them so far – i.e. classes within the same package. By default, classes are put into an un-named package. So, whenever writing a Java program, you are using packages – it is simply not explicit. Of course, you may override this by explicitly defining packages which will be discussed in the next section. The following classes – `A` and `B` – are in the same package by default (the un-named package):

```
class A {
    private String priv = "I'm private in A.";
    String default = "I'm not private.";

    public static void main(String [] args) {
        A aObj = new A();
        System.out.println(aObj.priv);          // prints "I'm private in A."
        System.out.println(aObj.default);       // prints "I'm not private."

        B bObj = new B();
        System.out.println(bObj.priv);          // not accessible in A!
        System.out.println(bObj.prot);          // prints "I'm protected."
        System.out.println(bObj.pub);           // prints "I'm public."
    }
}
```

```
class B {
    private String priv = "I'm private in B.";
    protected String prot = "I'm protected.";
    public String pub = "I'm public.";

    public static void main(String [] args) {
        A aObj = new A();
        System.out.println(aObj.priv);         // not accessible in B!
        System.out.println(aObj.default);      // prints "I'm not private."

        B bObj = new B();
        System.out.println(bObj.priv);         // prints "I'm private in B."
        System.out.println(bObj.prot);         // prints "I'm protected."
        System.out.println(bObj.pub);          // prints "I'm public."
    }
}
```

Notice that when two classes are in the same package, the `private` access modifier is the only one to affect the behavior of the classes. The `private` access modifier prevents access to any code defined outside the class in which the `private` member was declared. So, `private` members in `A` cannot be accessed by `B` and `private` members in `B` cannot be accessed by `A`. The same rules apply to the methods of a class.

Packages

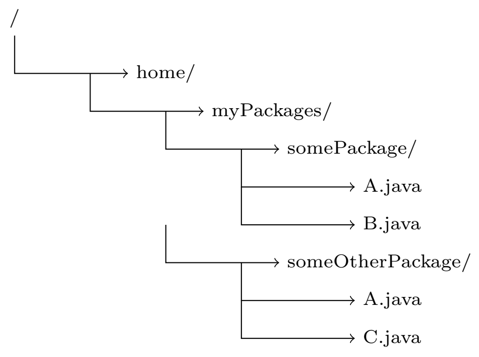
To really see the differences between each of the modifiers, it is necessary to discuss packages. Packages are provide a means of bundling a group of classes together. The primary benefits of packaging classes is to prevent naming clashes and to regulate access to code.

As mentioned above, when a class is not explicitly assigned to a package, it is automatically put into the un-named package. However, explicitly defining the package to which a class belongs simply requires a `package` statement:

```
package somePackage;

class A {
    public static void main(String [] args) {
        ...
    }
}
```

The first line of the code above puts `A` in the package `somePackage`. This now means that the name of `A` is actually `somePackage.A`, and in certain scenarios we must refer to `A` by its qualified name. Furthermore, this forces a directory structure on the way `A` is stored as a file. It implies that `A.java` must be located in a directory named `somePackage`. Consider the following directory structure:



In order to initiate the `main()` function of `A`, we must now use its qualified name: `somePackage.A`. However, this cannot be done anywhere. The call must be made in a way so that `A` is found. The way we usually initiate the `main()` function of a class is with the following command:

```
[user@notnotbc.org]$ java <classname>
```

However, now we are forced to be in a particular directory, or otherwise specify that particular directory. Namely, we have to be in the root of the package. So, from the directory `/home/myPackages`, we may execute the following command:

```
[user@notnotbc.org]$ pwd
/home/myPackages
[user@notnotbc.org]$ java somePackage.A
```

This command will look in the directory `somePackage` for a class called `A`. In order to execute this command from *any other directory*, it is possible to specify a ***classpath***. The classpath tells the JVM where to look for class files and by specifying a classpath, we add more locations to the default classpath. For instance, if located in the directory `somePackage`, the following would initiate the `main()` method of `A`:

```
[user@notnotbc.org]$ pwd
/home/myPackages/somePackage
[user@notnotbc.org]$ java -cp /home/myPackages somePackage.A
```

The command-line option `-cp` indicates that the next argument specifies the classpath. You may notice that given this directory structure, you may also package `A` as follows:

```
package myPackages.somePackage;

class A {
    public static void main(String [] args) {
        ...
    }
}
```

Note that the qualified name of `A` is now `myPackages.somePackage.A`. Each dot in the package name represents a level in the directory tree.

The fact that `A` now has a qualified name prevents it from every clashing with the name of another class. For instance, in the directory structure above, we have two classes named `A`. However, we do not get a name clash even if we use them in the same code. This is because they have different qualified names: `somePackage.A` and `someOtherPackage.A`. For instance, if both packages were imported into our code, we would access each class by it's qualified name:

```
import somePackage.*;         // imports all classes in somePackage
import someOtherPackage.*;    // imports all classes in someOtherPackage

class SomeClass {
    public static void main(String [] args) {
        somePackage.A spa = new somePackage.A();
        someOtherPackage.A sopa = new someOtherPackage.A();
            ...
    }
}
```

If we do not use the qualified name of `A` in each case, it is not clear which `A` we would like to use and so the compiler will prevent us from doing this.
Now, we can fully understand the default access modifier, *package-private*. *package-private* classes, and the variables and methods of those classes, may be accessed by any other class within the same package, but not by classes outside of that package. Furthermore, note that in the code above, `SomeClass` does not explicitly state which package it belongs to which implies it is in the un-named package. This means that both `somePackage.A` and `someOtherPackage.A` must be declared `public`:

```
package somePackage;

public class A {
    public static void main(String [] args) {
        ...
    }
}
```

```
package someOtherPackage;

public class A {
    public static void main(String [] args) {
        ...
    }
}
```

If those classes were not `public`, they would not be accessible outside of their packages. This is true even if any of the methods or variables in `somePackage.A` or `someOtherPackage.A` were `public`. In general, the most restrictive access level takes priority, so even if a class is `public` its `private`, `protected`, and *package-private* members will not be accessible as `public` members. The last access modifier, `protected`, allows only subclasses of a class to have access to that method, variable, or inner class. This modifier will prove more useful in a discussion of inheritance.

Information Hiding

The purpose of these access modifiers is to *hide information*. This is not necessarily because the information is sensitive, but because information hiding allow the designer of a class to be flexible about their implementation. In general, classes should be made as inaccessible as possible and then allow access as necessary. The reason is that as soon as software is published in the form of an API, anything in the code that is `public` becomes part of the API and is liable to be used by client code. This means that any changes that will be reflected in the API can result in malfunctions in the client code.

Consider for instance a class called `Line`, which represents a line with four values: the x- and y-coordinates of the two end-points of the line. And, suppose the class and its internals were to be made `public` so that they could be leveraged in client code.

```
package somePackage;

public class Line {
    public Line(int x1, int y1, int x2, int y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    public int x1, y1;
    public int x2, y2;
}
```

```
import somePackage.*;

class Client {
    public static void main(String [] args) {
        Line l = new Line(0,0,0,1);
        l.y2 = 10;
    }
}
```

Notice that client code using the `Line` class may access each of the member variables directly. This means that if at any point, the designer of the `Line` class decides to change the way the points are represented – i.e. in some way other than four `int` variables named `x1`, `y1`, `x2`, and `y2` – then the client code may suffer from a compile-time error. This can be seen in the code above. Imagine, for instance, that the names of the variables changed to: `x_one`, `y_one`, `x_two`, and `y_two`. This problem leads us to change the access level of the member variables and add methods to set and retrieve those values.

```
public class Line {
        ...

    public int getX(int which) throws Exception {
        if(which == 1) return x1;
        else if(which == 2) return x2;
        else throw new Exception("Only accepts 1 or 2");
    }

    public int getY(int which) throws Exception {
        if(which == 1) return y1;
        else if(which == 2) return y2;
        else throw new Exception("Only accepts 1 or 2");
    }

    public void setX(int which, int value) throws Exception {
        if(which, int value == 1) x1 = value;
        else if(which, int value == 2) x2 = value;
        else throw new Exception("Only accepts 1 or 2");
    }

    public void setY(int which, int value) throws Exception {
        if(which, int value == 1) y1 = value;
        else if(which, int value == 2) y2 = value;
        else throw new Exception("Only accepts 1 or 2");
    }

    private int x1, y1;
    private int x2, y2;
}
```

```
class Client {
    public static void main(String [] args) throws Exception {
        Line l = new Line(0,0,0,1);
        l.setY(2) = 10;
    }
}
```

Notice that now, the variables representing the points could have their names changed at any time without affecting the client code. However, notice that the parameter `which` is provided by the client and compared to hard-coded integers which specify the particular point in question. This leaks some implementation details which we should take some effort to cover up. Furthermore, the code could be cleaned up to create a class that's less bloated.

```
public class Line {
    public Line(x1, y1, x2, y2) {
        this.points[X][FIRST] = x1;
        this.points[Y][FIRST] = y1;
        this.points[X][SECOND] = x2;
        this.points[Y][SECOND] = y2;
    }

    public int get(int axis, int which) throws Exception {
        switch(axis) {
            case X:
                if(which == FIRST) return points[X][FIRST];
                else if(which == SECOND) return points[X][SECOND];
            case Y:
                if(which == FIRST) return points[Y][FIRST];
                else if(which == SECOND) return points[Y][SECOND];
            default:
                throw new Exception("Invalid value");
         }
    }

    public void set(char axis, int which, int value) throws Exception {
         switch(axis) {
            case X:
                if(which == FIRST) points[X][FIRST] = value;
                else if(which == SECOND) points[X][SECOND] = value;
                break;
            case Y:
                if(which == FIRST) points[Y][FIRST] = value;
                else if(which == SECOND) points[Y][SECOND] = value;
                break;
            default:
                throw new Exception("Invalid value");
         }
    }

    private int [][] points = new int [2][2];
    final public static int
        X = 0, Y = 1, FIRST = 0, SECOND = 1;
}
```

```
class Client {
    public static void main(String [] args) throws Exception {
        Line l = new Line(0,0,0,1);
        l.set(Line.Y, Line.SECOND, 10);
    }
}
```

Notice that the underlying structure that stores the data for the points is now a two-dimensional array where each coordinate is an element in that array. This, in and of itself, has no effect on our API – i.e. our public classes, methods, etc. – those may all be used just as before. However, the methods to manipulate and access the data have been generalized some more. This, of course, does have an effect on our API, but let's assume we wrote the generalized methods to begin with.

The generalization allows for fewer methods in our class and for the dimensionality of our `Line` class to grow in a way that's more scalable. For instance, imagine that our `Line` class had x-, y-, and z-coordinates and perhaps we stored more than just the end-points. The generalization comes at a price, though. It demands that the client code must now specify the point and the coordinate for that particular point as arguments. As mentioned earlier, this may leak some information about the internals of the class.

To mediate possible consequences, we may define constants which help the client access the data in the `Line` class. Notice the `final public static` fields in the `Line` class. These are supplied in the `Line` classes API so that a client can use them to be sure that their code maintains functionality even if the internals of the `Line` class have changed.

Let's consider the case were the data is re-structured once again. We may find that the points should have been encapusalted into a `Point` class as opposed to distinct `int` variables or an array.

```
public class Line {
    public Line(x1, y1, x2, y2) {
        points[FIRST] = new Point(x1, y1);
        points[SECOND] = new Point(x2, y2);
    }

    public int get(int axis, int which) throws Exception {
        switch(axis) {
            case X:
                if(which == FIRST) return points[FIRST].x;
                else if(which == SECOND) return points[SECOND].x;
            case Y:
                if(which == FIRST) return points[FIRST].y;
                else if(which == SECOND) return points[SECOND].y;
            default:
                throw new Exception("Invalid value");
          }
    }

    public void set(char axis, int which, int value) throws Exception {
         switch(axis) {
            case X:
                if(which == FIRST) points[FIRST].x = value;
                else if(which == SECOND) points[SECOND].x = value;
                break;
            case Y:
                if(which == FIRST) points[FIRST].y = value;
                else if(which == SECOND) points[SECOND].y = value;
                break;
            default:
                throw new Exception("Invalid value");
          }
    }

    private Point [] points = new Point[2];
    final public static int
        X = 0, Y = 1, FIRST = 0, SECOND = 1;
}

class Point {
    int x, y;
}
```

```
class Client {
    public static void main(String [] args) throws Exception {
        Line l = new Line(0,0,0,1);
        l.set(Line.Y, Line.SECOND, 10);
    }
}
```

Notice that the client code is unaffected by this change and, in fact, has no access to the `Point` class. However, say we wanted to allow the client to have access to the `Point`, that way our `Line` class could return `Point` objects and recieve them as paramaters from the client. This, of course, provides the client with some of the internals of the class. Though, the `Point` class is itself an abstraction and may hides its own implementation. This way, the composition doesn't reveal as much as it may seem. At any rate, it could be written as follows:

```
public class Line {
    public Line(x1, y1, x2, y2) { ... }
    public Line(Point p1, Point p2) {
        points[FIRST] = p1;
        points[SECOND] = p2;
    }

    public int get(int axis, int which) throws Exception { ... }
    public Point get(int which) throws Exception {
        switch(which) {
            case FIRST:
                return points[FIRST];
            case SECOND:
                return points[SECOND];
            default:
                throw new Exception("Invalid value");
        }
    }

    public void set(char axis, int which, int value) throws Exception { ... }
    public void set(int which, Point p) throws Exception {
        switch(which) {
            case FIRST:
                points[FIRST] = p;
                break;
            case SECOND:
                points[SECOND] = p;
                break;
            default:
                throw new Exception("Invalid value");
         }
    }

    private Point [] points = new Point[2];
    final public static int
        X = 0, Y = 1, FIRST = 0, SECOND = 1;
}
```

```
public class Point {
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    int x, y
}
```

```
class Client {
    public static void main(String [] args) throws Exception {
        Line l = new Line(0,0,0,1);
        l.set(Line.Y, Line.SECOND, 10);

        Point p = l.get(Line.SECOND);
        p.setX(10);
    }
}
```

Now, that the `Point` class is a part of the API, the `Line` class has the functionality to pass and recieve `Point` objects. This allows us to override the `get()` and `set()` methods in `Line` so that client code can now make use of the convenience of the `Point` class. However, notice that the code is backwards-compatible because the old methods are still there. This means we can add functionality without causing issues in client code.

While convenient, we still run into a potential problem given the way the code is written. Notice that the new `get()` method returns a reference to the `Point` object in our class. Furthermore, the `Point` object has a `set()` method. This means that anywhere in the client code where a call to `get()` happens, any changes made to the `Point` object returned will be visible in the `Line` object from which it came. This can prove to be *very* problematic and there are a number of ways to deal with this issue.

One way is to make the `Point` class *immutable*. This means it would not have any `set()` functions so that only particular classes could change it – namely, those in the same package – or, none at all. Additionally, the `get()` method may return a copy of the `Point` object.

```
    public class Line {
        public Point get(int which) throws Exception {
            Point p;
            switch(which) {
                case FIRST:
                    int x = points[FIRST].x;
                    int y = points[FIRST].y;
                    p = new Point(x, y);
                    return p;
                case SECOND:
                    int x = points[SECOND].x;
                    int y = points[SECOND].y;
                    p = new Point(x, y);
                    return p;
                default:
                    throw new Exception("Invalid value");
            }
        }
    }
```

As can be seen, an entirely different `Point` object has been created and the values were copied over into that new object. Any changes to the new object, will not result in changes in the `Point` object encapsulated within the `Line` object.

A mistaken approach to this problem would set the array or the `Point` objects themselves to `final`. However, `final` only means that *the reference is unchangeable* – i.e. the variable cannot refer to any other object. Therefore, the object to which the variable refers could very well be changed.