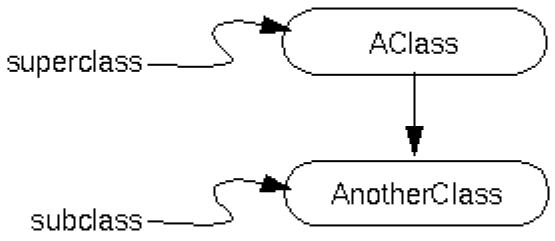
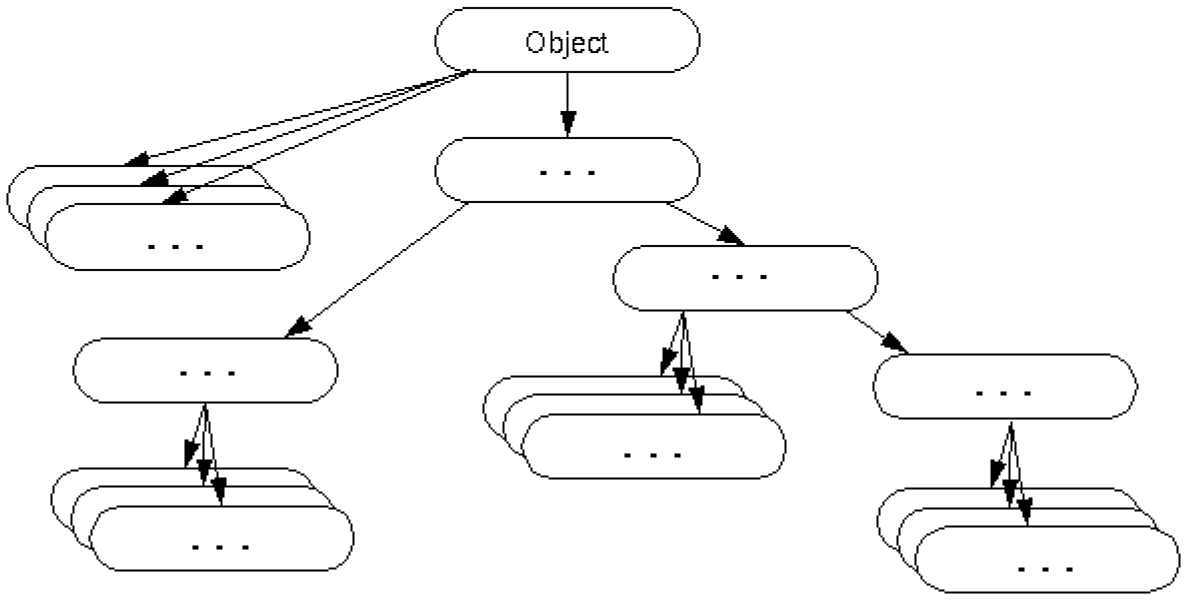
Subclasses, Superclasses, and Inheritance

To recap what you've seen before, classes can be derived from other classes. The derived class (the class that is derived from another class) is called a *subclass*. The class from which it's derived is called the *superclass*. The following figure illustrates these two types of classes:



In fact, in Java, all classes must be derived from some class. Which leads to the question "Where does it all begin?" The top-most class, the class from which all other classes are derived, is the `Object` class defined in `java.lang`. `Object` is the root of a hierarchy of classes, as illustrated in the following figure.



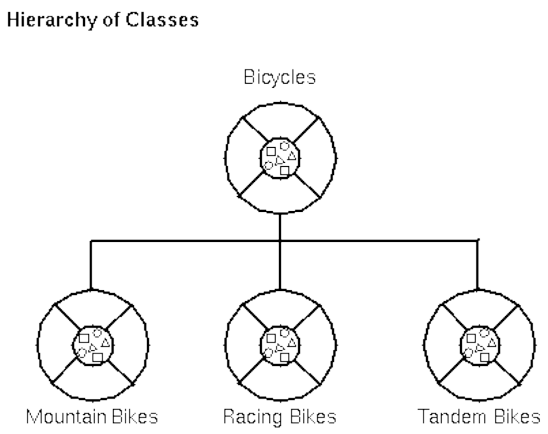
The subclass inherits state and behavior in the form of variables and methods from its superclass. The subclass can use just the items inherited from its superclass as is, or the subclass can modify or override it. So, as you drop down in the hierarchy, the classes become more and more specialized:

Definition: A subclass is a class that derives from another class. A subclass inherits state and behavior from all of its ancestors. The term superclass refers to a class's direct ancestor as well as all of its ascendant classes.

What Is Inheritance?

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a penny-farthing is, if I told you it was a bicycle, you would know that it had two wheels, handle bars, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all different kinds of bicycles. In object-oriented terminology, mountain bikes, racing bikes, and tandems are all *subclasses* of the bicycle class. Similarly, the bicycle class is the *superclass* of mountain bikes, racing bikes, and tandems.



Each subclass *inherits* state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviors: braking and changing pedaling speed, for example.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. What would be the point in that? Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also *override* inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could actually use those new gears.

You are not limited to just one layer of inheritance. The inheritance tree, or *class hierarchy*, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the further down in the hierarchy a class appears, the more specialized its behavior.

The Benefits of Inheritance

- Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.
- Programmers can implement superclasses called *abstract classes* that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

Creating Subclasses

To create a subclass of another class use the `extends` clause in your class declaration. (The Class Declaration explains all of the components of a class declaration in detail.) As a subclass, your class inherits member variables and methods from its superclass. Your class can choose to hide variables or override methods inherited from its superclass.

Writing Final Classes and Methods

Sometimes, for security or design reasons, you want to prevent your class from being subclassed. Or, you may just wish to prevent certain methods within your class from being overriden. In Java, you can achieve either of these goals by marking the class or the method as *final*.

Writing Abstract Classes and Methods

On the other hand, some classes are written for the sole purpose of being subclassed (and are not intended to ever be instantiated). These classes are called *abstract classes* and often contain *abstract methods*.

The Object Class

All objects in the Java environment inherit either directly or indirectly from the `Object` class. This section talks about the interesting methods in `Object`--methods that you may wish to invoke or override.

Creating Subclasses

You declare that a class is the subclass of another class within the Class Declaration. For example, suppose that you wanted to create a subclass named `SubClass` of another class named `SuperClass`. You would write:

```
class SubClass extends SuperClass {
    . . .
}
```

This declares that `SubClass` is the subclass of the `Superclass` class. It also implicitly declares that `SuperClass` is the superclass of `SubClass`. A subclass also inherits variables and methods from its superclass's superclass, and so on up the inheritance tree. To simplify our discussion, when this tutorial refers to a class's superclass it means the class's direct ancestor as well as all of its ascendant classes.

A Java class can have only one direct superclass. Java does not support multiple inheritance.

Creating a subclass can be as simple as including the `extends` clause in your class declaration. However, you usually have to make other provisions in your code when subclassing a class, such as overriding methods or providing implementation for abstract methods.

What Member Variables Does a Subclass Inherit?

Rule: A subclass inherits all of the member variables within its superclass that are accessible to that subclass (unless the member variable is hidden by the subclass).

The following list itemizes the member variables that are inherited by a subclass:

- Subclasses inherit those member variables declared as `public` or `protected`.
- Subclasses inherit those member variables declared with no access specifier as long as the subclass is in the same package as the superclass.
- Subclasses don't inherit a superclass's member variable if the subclass declares a member variable using the same name. The subclass's member variable is said to hide the member variable in the superclass.
- Subclasses don't inherit the superclass's `private` member variables.

Hiding Member Variables

As mentioned in the previous section, member variables defined in the subclass hide member variables of the same name in the superclass.

While this feature of the Java language is powerful and convenient, it can be a fruitful source of errors: hiding a member variable can be done deliberately or by accident. So, when naming your member variables be careful to only hide those member variables that you actually wish to hide.

One interesting feature of Java member variables is that a class can access a hidden member variable through its superclass. Consider this superclass and subclass pair:

```
class Super {
    Number aNumber;
}
class Sub extends Super {
    Float aNumber;
}
```

The `aNumber` variable in Sub hides `aNumber` in `Super`. But you can access `aNumber` from the superclass with:

```
super.aNumber
```

`super` is a Java language keyword that allows a method to refer to hidden variables and overriden methods of the superclass.

What Methods Does a Subclass Inherit?

The rule that specifies which methods get inherited by a subclass is similar to that for member variables.

Rule: A subclass inherits all of the methods within its superclass that are accessible to that subclass (unless the method is overriden by the subclass).

The following list itemizes the methods that are inherited by a subclass:

- Subclasses inherit those methods declared as `public` or `protected`.
- Subclasses inherit those superclass methods declared with no access specifier as long as the subclass is in the same package as the superclass.
- Subclasses don't inherit a superclass's method if the subclass declares a method using the same name. The method in the subclass is said to override the one in the superclass.
- Subclasses don't inherit the superclass's `private` methods.

A subclass can either completely override the implementation for an inherited method, or the subclass can enhance the method by adding functionality to it.

Overriding Methods

The ability of a subclass to override a method in its superclass allows a class to inherit from a superclass whose behavior is "close enough" and then supplement or modify the behavior of that superclass.

Writing Final Classes and Methods

Final Classes

You can declare that your class is final; that is, that your class cannot be subclassed. There are (at least) two reasons why you might want to do this: security reasons and design reasons.

Security: One mechanism that hackers use to subvert systems is to create subclasses of a class and then substitute their class for the original. The subclass looks and feels like the original class but does vastly different things possibly causing damage or getting into private information. To prevent this kind of subversion, you can declare your class to be final and prevent any subclasses from being created. The `String` class in the java.lang package is a final class for just this reason. The `String` class is so vital to the operation of the compiler and the interpreter that the Java system must guarantee that whenever a method or object uses a `String` they get exactly a `java.lang.String` and not some other string. This ensures that all strings have no strange, inconsistent, undesirable, or unpredictable properties.

If you try to compile a subclass of a final class, the compiler will print an error message and refuse to compile your program. In addition, the bytecode verifier ensures that the subversion is not taking place at the bytecode level by checking to make sure that a class is not a subclass of a final class.

Design: Another reason you may wish to declare a class as final are for object-oriented design reasons. You may think that your class is "perfect" or that, conceptually, your class should have no subclasses.

To specify that your class is a final class, use the keyword `final` before the `class` keyword in your class declaration. For example, if you wanted to declare your (perfect) `ChessAlgorithm` class as final, its declaration would look like this:

```
final class ChessAlgorithm {
    . . .
}
```

Any subsequent attempts to subclass `ChessAlgorithm` will result in a compiler error such as the following:

```
Chess.java:6: Can't subclass final classes: class ChessAlgorithm
class BetterChessAlgorithm extends ChessAlgorithm {
      ^
1 error
```

Final Methods

If creating a final class seems heavy handed for your needs, and you really just want to protect some of your class's methods from being overriden, you can use the `final` keyword in a method declaration to indicate to the compiler that the method cannot be overridden by subclasses.

You might wish to make a method final if the method has an implementation that should not be changed and is critical to the consistent state of the object. For example, instead of making your `ChessAlgorithm` class final, you might just want to make the `nextMove` method final:

```
class ChessAlgorithm {
    . . .
    final void nextMove(ChessPiece pieceMoved, BoardLocation
newLocation) {
    }
    . . .
}
```

Writing Abstract Classes and Methods

Abstract Classes

Sometimes, a class that you define represents an abstract concept and as such, should not be instantiated. Take, for example, food in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (my favorite), chocolate. Food represents the abstract concept of things that we can eat. It doesn't make sense for an instance of food to exist.

Similarly in object-oriented programming, you may want to model abstract concepts but you don't want to be able to create an instance of it. For example, the `Number` class in the `java.lang` package represents the abstract concept of numbers. It makes sense to model numbers in a program, but it doesn't make sense to create a generic number object. Instead, the `Number` class makes sense only as a superclass to classes like `Integer` and `Float` which implement specific kinds of numbers. Classes such as `Number`, which implement abstract concepts and should not be instantiated, are called *abstract classes*. An abstract class is a class that can only be subclassed--it cannot be instantiated.

To declare that your class is an abstract class, use the keyword `abstract` before the `class` keyword in your class declaration:

```
abstract class Number {
    . . .
}
```

If you attempt to instantiate an abstract class, the compiler will display an error similar to the following and refuse to compile your program:

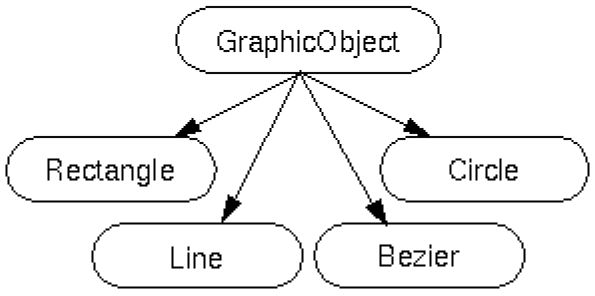
```
AbstractTest.java:6: class AbstractTest is an abstract class. It can't
be instantiated.
```

```
        new AbstractTest();
        ^
1 error
```

Abstract Methods

An abstract class may contain *abstract methods*, that is, methods with no implementation. In this way, an abstract class can define a complete programming interface, thereby providing its subclasses with the method declarations for all of the methods necessary to implement that programming interface. However, the abstract class can leave some or all of the implementation details of those methods up to its subclasses.

Let's look at an example of when you might want to create an abstract class with an abstract method in it. In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curvess, and so on. Each of these graphic objects share certain states (position, bounding box) and behavior (move, resize, draw). You can take advantage of these similarities and declare them all to inherit from the same parent object--`GraphicObject`.



However, the graphic objects are also substantially different in many ways: drawing a circle is quite different from drawing a rectangle. The graphics objects cannot share these types of states or behavior. On the other hand, all `GraphicObject`s must know how to draw themselves; they just differ in how they are drawn. This is a perfect situation for an abstract superclass.

First you would declare an abstract class, `GraphicObject`, to provide member variables and methods that were wholly shared by all subclasses, such as the current position and the `moveTo` method. `GraphicObject` also declares abstract methods for methods, such as `draw`, that need to be implemented by all subclasses, but are implemented in entirely different ways (no default implementation in the superclass makes sense). The `GraphicObject` class would look something like this:

```
abstract class GraphicObject {
    int x, y;
    . . .
    void moveTo(int newX, int newY) {
        . . .
    }
    abstract void draw();
}
```

Each non-abstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, would have to provide an implementation for the `draw` method.

```
class Circle extends GraphicObject {
    void draw() {
        . . .
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        . . .
    }
}
```

An abstract class is not required to have an abstract method in it. But any class that has an abstract method in it or that does not provide an implementation for any abstract methods declared in its superclasses *must* be declared as an abstract class.

The Object Class

The Object class sits at the top of the class hierarchy tree in the Java development environment. Every class in the Java system is a descendent (direct or indirect) of the `Object` class. The `Object` class defines the basic state and behavior that all objects must have, such as the ability to compare oneself to another object, to convert to a string, to wait on a condition variable, to notify other objects that a condition variable has changed, and to return the object's class.

The `equals` Method

Use the `equals` to compare two objects for equality. This method returns `true` if the objects are equal, false otherwise. Note that equality does not mean that the objects are the same object. Consider this code that tests two `Integer`s, `one` and `anotherOne`, for equality:

```
Integer one = new Integer(1), anotherOne = new Integer(1);

if (one.equals(anotherOne))
    System.out.println("objects are equal");
```

This code will display `objects are equal` even though `one` and `anotherOne` reference two different, distinct objects. They are considered equal because they contain the same integer value.

Your classes should override this method to provide an appropriate equality test. Your `equals` method should compare the contents of the objects to see if they are functionally equal and return `true` if they are.

The `getClass` Method

The `getClass` method is a final method (cannot be overridden) that returns a runtime representation of the class of this object. This method returns a `Class` object. You can query the `Class` object for a variety of information about the class, such as its name, its superclass, and the names of the interfaces that it implements. The following method gets and displays the class name of an object:

```
void PrintClassName(Object obj) {
    System.out.println("The Object's class is " +
obj.getClass().getName());
}
```

One handy use of the `getClass` method is to create a new instance of a class without knowing what the class is at compile time. This sample method creates a new instance of the same class as `obj` which can be any class that inherits from `Object` (which means that it could be any class):

```
Object createNewInstanceOf(Object obj) {
    return obj.getClass().newInstance();
}
```

The `toString` Method

`Object`'s `toString` method returns a `String` representation of the object. You can use `toString` to display an object. For example, you could display a `String` representation of the current `Thread` like this:

```
System.out.println(Thread.currentThread().toString());
```

The `String` representation for an object is entirely dependent on the object. The `String` representation of an `Integer` object is the integer value displayed as text. The `String` representation of a `Thread` object contains various attributes about the thread, such as its name and priority. For example, the previous of code above display the following:

```
Thread[main,5,main]
```

The `toString` method is very useful for debugging and it would behoove you to override this method in all your classes.

Object Methods Covered In Other Lessons or Sections

The `Object` class provides a method, `finalize` that cleans up an object before its garbage collected. This method's role during garbage collection is discussed in this lesson in Cleaning Up Unused Objects. Also, Writing a finalize Method shows you how to write override the `finalize` method to handle the finalization needs for you classes.

The `Object` class also provides five methods that are critical when writing multithreaded Java programs:

- `notify`
- `notifyAll`
- `wait` (three versions)

These methods help you ensure that your threads are synchronized and are covered in Threads of Control. Take particular note of the page titled Synchronizing Threads.