In inheritance, you can create a new class from an existing class. This is called *inheritance*.

In Java, inheritance is achieved with the `extends` keyword:

```java
public class NewClass extends OldClass
```

`NewClass` is called the *subclass*, *child class*, *extended class*, or *derived class*. `OldClass` is called the *superclass*, *parent class*, or *base class*.

The subclass will be comprised of all the members of the superclass and the new members specified in the new class.

The subclass `NewClass` is said to have an *is-an relationship* with the superclass `OldClass`.

If a class `C1` has as a member an object of the class `C2`, then `C1` is said to *have-a* `C2`.

A class can be derived from a derived class.
When instantiating an object of a subclass, Java will invoke the default constructor of the superclass, unless a constructor of the superclass is explicitly invoked using the keyword `super`. Invoking the constructor of the superclass must be the first statement in the constructor of the subclass.

```java
public class OldClass {
    OldClass() {
    }

    OldClass(int i) {
    }
}

public class NewClass extends OldClass {
    NewClass(int j) {
        // OldClass()
    }

    NewClass(int j) {
        super(j);
    }
}
```
If the superclass does not have a default constructor, writing a constructor of the subclass that does not invoke a specific constructor of the superclass will generate a compiler error.

```java
public class OldClass {
    OldClass(int i) {
    }
}

public class NewClass extends OldClass {
    NewClass(int j) {
    }          // Compiler error
}
```

If it is reasonable to create a default object of a class, one **should** to create a default constructor for the class so that derived classes do not need to invoke a constructor of the superclass explicitly.
Visibility

Derived classes, like any class, can access public data and methods members of the superclass.

Derived classes can access data and methods members of the superclass without their visibility defined, as long as the derived class is a member of the same package of the superclass.

Derived classes cannot access private data and methods members of the superclass.

Derived classes that are members of a separate package from the superclass can access protected data and methods members of the superclass.

The private and protected modifiers can be applied only to members of a class, they cannot be applied to the class itself.

Table 11.2 on page 438 of your text book lists the visibility rules. You can also find the information at:

http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
Overriding Methods

If a method member of a class is defined in a subclass with the same signature as the method in the parent class, the method in the derived class **overrides** the method in the parent class.

If a method member of a class is defined in a subclass with a **different signature** as the method in the parent class, the method in the derived class **overloads** the method in the parent class.

If a method member of a class is defined as **static**, the method is inherited by derived classes, but it **cannot** be overridden. Redefining it in the subclass **hides** the method in the superclass.

If a data member of a class is redefined in a subclass, the data member of the subclass hides the data member of the superclass.

The special syntax `@Override` placed before a method in a derived class makes the compiler check that the method exists in the superclass.
The **Object** class

All classes are implicitly derived from the **Object** class.

Among the inherited methods is `toString()`. This method will return the name of the class, followed by an at sign (@) and the address of the current object of the class.

The **Object** class also defines the method `clone()`, which we will cover when discussing *interfaces*.

### The `instanceof` keyword

The `instanceof` keyword allows you to verify if an object is an instance of a particular class:

```
if ( anObject instanceof aClass )
```

**Use with caution and only as last resort.** To quote Scott Meyers, *Effective C++*:

> Anytime you find yourself writing code of the form “if object is of type T1, then do something, but if it's of type T2, the do something else”, slap yourself.
Polymorphism

From the Greek "many forms". It indicates the ability of an object of a class to behave like an object of a superclass in the inheritance chain. If the class `Student` extends the class `Person`, then a `Student` object can be used both with instances of the `Student` type and the `Person` type.

```java
class Person {
    public void doPerson(Person p) { ... }
}
class Student extends Person {
    public void doStudent(Student s) { ... }
}
Person p = new Student(); // legal, Student is-a Person
Student s = new Person(); // illegal
Student s = new Student();
p.doPerson(s); // legal
s.doStudent(p); // illegal
printLastName(p);
printLastName(s);
void printLastName(Person p);
```
When an object invokes a member function, the JVM searches the object's class for an instance of that function. If the class does not have an accessible instance, it will search the parent of that class and, if needed, the parent of the parent, etc. The searches continues until the `Object` class is reached.

```java
class Object {}
class GGP {}
class GP extends GGP {}
class P extends GP {}
class C extends P {}

C c = new C();
c.f();

// If C.f exists use it
// else if P.f exists use it
// else if GP.f exists use it
// else if GGP.f exists use it
// else if Object.f exists use it
// else error
```

This is called *dynamic binding*.
A variable of a class can always be assigned to an instance of one of its subclasses, but a variable of a class cannot be assigned to an instance of one of its ancestors classes:

```java
class GP {}
class P extends GP {}
class C extends P {}

GP gp = new GP();
P gp = new C(); // legal
P p = new C(); // legal
P p = new GP(); // illegal
C c = new P(); // illegal
C c = new GP(); // illegal
```

To force the compiler to accept the assignment, you can use explicit casting:

```java
P p = (P)new GP(); // legal
C c = (C)new P(); // legal
```

If the superclass object is not an instance of the subclass, explicit casting will cause a runtime `ClassCastException`.

Explicit cast allows the developer to write generic functions.
The **final** keyword prevents the class from being extended. The `Math` class is an example.

```java
public final class P {}
public class C extends P {}  // illegal
```

The **final** keyword can be applied to a function, which will prevent derived class from overriding it.

```java
public class P {
    public final void doIt() {}
}

public class C extends {
    public void doIt() {}  // illegal
}
```
The **ArrayList** Class

The **ArrayList** class is included in the Java API and allows for the creation of arrays whose size can be dynamically changed. The class is a *generic class of type E*:

```java
ArrayList<String> cities = new ArrayList<String>();
// In JDK 7 this can be written as
ArrayList<String> cities = new ArrayList<>();

cities.add("Boston");
cities.add("New York");
cities.add("Newark");
if ( cities.contains("Washington") ) {}  
```

Check the Java API for more member functions