Inheritance in Java

Definitions: A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

- In Java **Object** is the top most class in class hierarchy, which has no superclass.
- Every class has one and only one direct superclass (single inheritance). This superclass may be derived from another class, and so on, and ultimately derived from the topmost class, Object.
- In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

- When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.
- A subclass inherits all the permitted *members* (fields, methods, and nested classes) from its superclass.
- Constructors are not members, so they are not inherited by subclasses.
- But the constructor of the superclass can be invoked from the subclass.

An Example of Inheritance

```
public class Bicycle {

   // the Bicycle class has three fields
   public int cadence;    public int gear;    public int speed;

   // the Bicycle class has one constructor
   public Bicycle(int startCadence, int startSpeed, int startGear) {
      gear = startGear;       cadence = startCadence;        speed = startSpeed;
   }
   // the Bicycle class has four methods
   public void setCadence(int newValue) {  cadence = newValue;    }

   public void setGear(int newValue) {    gear = newValue;    }

   public void applyBrake(int decrement) { speed -= decrement;    }

   public void speedUp(int increment) {  speed += increment;    }

}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {

   public int seatHeight;         // the MountainBike subclass adds one field
```

```
    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,  int startCadence,  int startSpeed,  int startGear) {
      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;
    }
    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {    seatHeight = newValue;    }
}
```

What You Can Do in a Subclass

- A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent.
- You can use the inherited members as is, replace them, hide them, or supplement them with new members.
- Hiding members : You can hide the either fields or methods of superclass
    - A field declared in the subclass with the same name as the one in the superclass (not recommended).
    - Write a new ***static*** method in the subclass that has the same signature as the one in the superclass (**static** method) , thus *hiding* it.

- **Overriding** superclass method : The inherited methods can be used directly as they are. Or write a new *instance* method in the subclass that has the same signature as the one in the superclass.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.
- A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.
- A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Casting Objects

For example, if we write   public MountainBike myBike = new MountainBike();

- **myBike** is a type of MountainBike which is derived from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.
- The reverse is not necessarily true: a Bicycle *may be* a MountainBike, but it isn't necessarily. Similarly, an Object *may be* a Bicycle or a MountainBike, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
```

then obj is both an Object and a MountainBike. (until such time as obj is assigned another object that is not a MountainBike). This is called implicit casting.

- If, on the other hand, we write `MountainBike myBike = obj;` we would get a compile-time error because obj is not known to the compiler to be a MountainBike. However, we can *tell* the compiler that we promise to assign a MountainBike to obj by *explicit casting:* `MountainBike myBike = (MountainBike)obj;`

Overriding Methods

- A subclass method with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass.
- Subclass **overrides** a method of a superclass whose behavior is "close enough" and then to modify behavior as needed.
- **In simple language** any method that you can override via inheritance is **virtual method.** In Java, all non-static methods are by default "virtual methods." Only methods marked with the keyword final, which cannot be overridden, along with private methods, which are not inherited, are non-virtual. In Java there is no **keyword** names "**virtual**".

Employee.java

```
public class Employee
{
  private String name;
  private String address;
  private int number;
  public Employee(String name, String address,
                  int number)
  {
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
  }
  public void mailCheck()
  {
    System.out.println("Mailing a check to " + this.name
     + " " + this.address);
  }
}
```

VirtualMethod.java

```
class Salary extends Employee
{
  private double salary;   //Annual salary
  public Salary(String name,
        String address,
        int number, double salary)
  {
     super(name, address, number);
     this.salary=salary;
  }
  @Override
  public void mailCheck()
  {
     System.out.println("Within mailCheck
          of Salary class ");
     System.out.println("Mailing check to "
            + " with salary " + salary);
  }
}
```

```
public class VirtualMethod
{
  public static void main(String [] args)
  {
    Salary s = new Salary("Kunal Sarkar", "Baranagar, WB", 3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
    System.out.println("Call mailCheck using Salary reference --");          s.mailCheck();
    System.out.println("\n Call mailCheck using Employee reference--");       e.mailCheck();
  }
}
```

Two Salary objects : One using a Salary reference s, and other using an Employee reference e.

Output

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to  with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to  with salary 2400.0
```

Explanation

- While invoking s.mailCheck() the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.
- When the compiler sees e.mailCheck(), the compiler sees the mailCheck() method in the Employee class for validation but at run time, the JVM invokes mailCheck() in the Salary class.

- From Java 1.5 onwards, an overriding method (method in sub class) in Java can return sub-class of return type of overridden method (super class method). So if your original method returns java.lang.Object then a method which overrides this in subclass can return object of Subclass. For example :

```
public class Shape{
    public Shape getShape()  {
          return new Shape();
    }
}
```

```
public class Circle extends Shape {
     @Override   public Circle getShape() {  return new Circle();  }
}
```

This subtype is called a ***covariant return type***.

- Use of **@Override** annotation during overriding instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the super classes, then it will generate an error.

The main reason @Override was created was to deal with simple (but nasty) typographical errors. For example, a method mistakenly declared as public int hashcode(){...} (notice all lower case) in subclass to override a method public int hashCode(){...} in super class.

It will however compile perfectly well. Such an error is easy to make, and difficult to catch, which is a dangerous combination. Using the @Override annotation prevents you from making such errors. You should be in the habit of using @Override whenever you override a superclass method, or implement an interface method.

It is not **mandatory** to use @Override when you override a method. In case you are using @Override annotation, and the method signature is not found at super class will results compilation error.

```
public class Animal {
    public void makeSound() {     }
}
```

```
class Cat extends Animal{
   @Override   public void makeSound(){
       System.out.println("myyyyyaaawwwwww");
   }
}
```

Using @override has two benefits:

- You can take advantage of the compiler checking to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does.
- It makes your code easier to understand because it is more obvious when methods are overwritten.

Hiding methods

Static Methods : If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

Implications of hiding a static method and overriding an instance method :

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Consider an example that contains two classes.

```
public class Animal {
   public void instanceMethod() {
      System.out.println("The instance
            method in Animal");
   }
   public static void classMethod() {
      System.out.println("The static method
            in Animal");
   }
}
```

```
public class Dog extends Animal {
   @override
   public void instanceMethod() {
      System.out.println("The instance method in Dog”);
    }
   // hides Animal’s method
    public static void classMethod() {
      System.out.println("The static method in Dog");
}
```

Here, Dog.classMethod() is said to hide Animal.classMethod(). Hiding does not work like overriding, because static methods are not polymorphic. So we call these methods following will happen:

```
public static void main(String[] args) {
     Animal.classMethod();      // prints The static method in Animal
     Dog.classMethod();         // prints The static method in Dog

     Animal a = new Animal();
     Animal b = new Dog();
     Dog c = new Dog();
     Animal d = null;
```

Calling static methods on instances rather than classes is a very **bad practice**, and should never be done.

```
     a.classMethod() ;     // Prints The static method in Animal because the declared type of a is Animal
     b.classMethod() ;     // Prints The static method in Animal because the declared type of b is Animal
     c.classMethod() ;     // Prints The static method in Dog because the declared type of c is Dog
     d.classMethod() ;     // Prints The static method in Animal because the declared type of d is Animal

     //  But if we call the instance method
     a.instanceMethod ;       // Prints The instance method in Animal
     b.instanceMethod ();     //  Prints The instance method in Dog – due to overriding

     c.instanceMethod ();   // Prints The instance method in Dog
     d.instanceMethod ():   // throws NullPointerException
   }
}
```

Note: **super**, like **this**, can't be used in a static context. We can call Animal.classMethod() from Dog.classMethod(), though, but the two methods have in fact nothing in common, except their name and signature. Hiding is used to differentiate from overriding. But method hiding is not really useful, whereas method overiding is one af the key parts of OO design.

Interface Methods

- Default methods and abstract methods in interfaces are inherited like instance methods.
- When the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict.
- These rules are driven by the following **two principles**:

1. Instance methods are preferred over interface default methods.

   Consider the following classes and interfaces:

```
public class Horse {
   public String identifyMyself() {    return "I am a horse.";    }
}
public interface Flyer {
   default public String identifyMyself() {    return "I am able to fly.";     }
}
public interface Mythical {
   default public String identifyMyself() {     return "I am a mythical creature.";    }
```

```
    }
    public class Pegasus extends Horse implements Flyer, Mythical {
       public static void main(String... args) {
          Pegasus myApp = new Pegasus();
          System.out.println(myApp.identifyMyself());
       }
    }
```

The method Pegasus.identifyMyself returns the string I am a horse, i.e it is calling instance method.

2. Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor. Consider the following interfaces and classes:

```
    public interface Animal {
       default public String identifyMyself() {      return "I am an animal.";    }
    }
    public interface EggLayer extends Animal {
       default public String identifyMyself() {     return "I am able to lay eggs.";    }
    }
    public interface FireBreather extends Animal { }
    public class Dragon implements EggLayer, FireBreather {
       public static void main (String... args) {
          Dragon myApp = new Dragon();
          System.out.println(myApp.identifyMyself());
       }
    }
```

It is ignored as it is overridden.

The method Dragon.identifyMyself returns the string I am able to lay eggs.

If two or more independently defined default methods conflict, or a **default** method conflicts with an **abstract** method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.

Consider the example about computer-controlled cars that can now fly. You have two interfaces (OperateCar and FlyCar) that provide default implementations for the same method, (startEngine):

```
public interface OperateCar {    // ...
    default public int startEngine(EncryptedKey key) {      // Implementation     }
}
public interface FlyCar {     // ...
   default public int startEngine(EncryptedKey key) {       // Implementation    }
}
```

A class that implements both OperateCar and FlyCar must override the method startEngine. You could invoke any of the of the default implementations with the super keyword.

```
public class FlyingCar implements OperateCar, FlyCar {
   // ...
   public int startEngine(EncryptedKey key) {
      FlyCar.super.startEngine(key);
      OperateCar.super.startEngine(key);
   }
```

The name preceding **super** (in this example, FlyCar or OperateCar) must refer to a direct superinterface that defines or inherits a default for the invoked method.

```
    }
```

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

```
public interface Mammal {   String identifyMyself();   }
public class Horse {
   public String identifyMyself() {     return "I am a horse.";   }
}
public class Mustang extends Horse implements Mammal {
   public static void main(String... args) {
      Mustang myApp = new Mustang();
      System.out.println(myApp.identifyMyself());
   }
}
```

Here identifyMyself() method of Horse class override the same method of interface. So separate implementation not provided.

Output : **I am a horse.** The class Mustang inherits the method identifyMyself from the class Horse, which overrides the abstract method of the same name in the interface Mammal.

Note: **Static methods in interfaces are never inherited.**

Access specifier in overridden methods of subclass

The access specifier for an overriding method can allow **more**, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

Summary

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

Defining a Method with the Same Signature as a Superclass's Method		
	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Note: In a subclass, you can **overload** the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are **new methods**, unique to the subclass.

Polymorphism

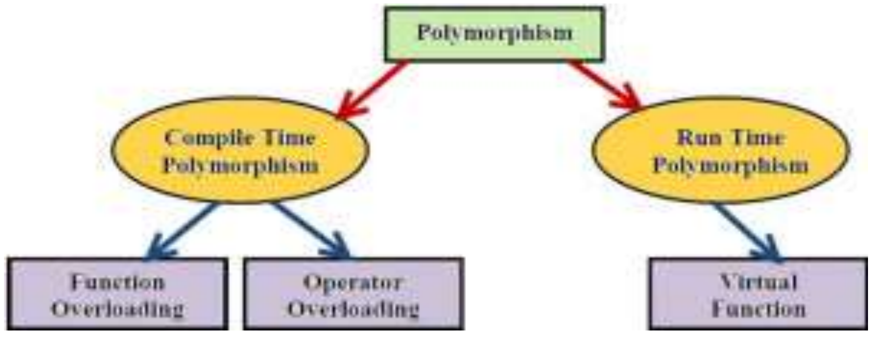
- The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages.

Our universe exhibits many examples of entities that can change form:

- A butterfly morphs from larva to pupa to imago, its adult form.
- On Earth, the normal state of water is liquid, but water changes to a solid when frozen, and to a gas when heated to its boiling point.

This ability to change form is known as *polymorphism*. It is nothing but the ability to take more than one form.

- In OOP, polymorphism means a type can point to different object at different time. In other words, the actual object to which a reference type refers, can be determined at runtime.
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.
- Polymorphism in OOP is closely associated with the principle of inheritance.
- It means one interface, many possible implementations.



In Java it is implemented using overloading and overriding.

- **Two types of Polymorphism:**

1. **Static Polymorphism:** It is achieved through method overloading and operator overloading. It is always faster. It is also called as compile time polymorphism. At the compilation time java knows which method is call by checking the arguments, so it is also known as early binding or static binding.

   - **Method Overloading:** It is nothing but the ability of one function performs different tasks. These functions must differ by the data types. To call function the same function name is used for various instances.

```
class Children //create class
{
    void student()   //declare method without parameters
    {
        System.out.println("Students are playing");
    }
    //declare same method with parameter
    void student(int rollno, String name)
    {
        System.out.println("Roll No: "+rollno+"\nName: "+name);
    }
    public static void main(String[] args)//main method
    {
        Children ch = new Children();//Create object of class
        ch.student();//without parameters method call
        ch.student(1, "Priya");//with parameters method call
        ch.student(2, "Pranjal");
    }
}
```

```
Output
Students are playing
Roll No: 1
Name: Priya
Roll No: 2
Name: Pranjal
```

- **Operator Overloading :** Java does not support operator overloading.

2. **Dynamic Polymorphism :** It is also called as **run-time polymorphism**. In this case java compiler does not know which method is invoked at compilation time. Just JVM decides which method is invoked at the run-time. Method **overriding** is example of run-time polymorphism. In this case overridden method is invoking through the super class reference variable.

```
class Children      //parent class
{
    public void speak()   //define method
    {
        System.out.println("Children speak in Hindi");
    }
}
```

```
Subclass
class Student extends Children
{
    public void speak()     //override method
    {
        System.out.println("Students can
            speak in English");
    }
}
```

```
class Test
{
    public static void main(String[] args)     //main method
    { //create object of Children and Student class
        Children c1 = new Children();
        Student s = new Student();
        //call method speak
        c1.speak();
        s.speak();
    }
}
```

```
Output
Children speak in Hindi
Students can speak in English
```

Check earlier examples and discussion on override method

Suppose that we have the following interface and classes:

```
public interface Animal {
   public void move();
}
public class Bird implements Animal {
   public void move() {
      System.out.print("Flying...");
   }
}
public class Fish implements Animal {
   public void move() {
      System.out.print("Swimming...");
   }
}
public class Dog implements Animal {
   public void move() {
      System.out.print("Running...");
   }
}
```

Now we come to a more interesting example to see the power of polymorphism.
Suppose that we have a trainer who teaches animals. We create the Trainer class as follows:

```
public class Trainer {
   public void teach(Animal anim) {
      anim.move();
   }
}
```

Notice that the teach() method accepts any kind of Animal. Thus we can pass any objects which are sub types of the Animal type. anim of type Animal references 3 different kinds of object: Dog, Bird and Fish.

For example:

```
Trainer trainer = new Trainer();
Dog dog = new Dog();

Bird bird = new Bird();
Fish fish = new Fish();

trainer.teach(dog);
trainer.teach(bird);
trainer.teach(fish);
```

A reference type can take different objects (many forms). This is the simplest form of polymorphism.

```
Outputs:
Running...
Flying...
Swimming...
```

The teach() method can accept 'many forms' of Animal: Dog, Bird, Fish,… as long as they are sub types of the Animal interface.
In the teach() method is invoked on the Animal reference. And depending on the actual object type, the appropriate method is called.

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as ***virtual method invocation*** and demonstrates an aspect of the important polymorphism features in the Java language.

Why is Polymorphism?

Polymorphism is a robust feature of OOP. It increases the reusability, flexibility and extensibility of code. Take the above example for instance:

- Reusability: the teach() method can be re-used for different kinds of objects as long as they are sub types of the Animal interface.
- Flexibility: the actual object can be determined at runtime which allows the code run more flexibly.
- Extensibility: when we want to add a new kind of Animal, e.g. Snake, we just pass an object of Snake into the teach() method without any modification.

Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if **their types are different**. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through **super**.

Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword **super**. You can also use super to refer to a hidden field (although hiding fields is discouraged). Consider this class, Superclass:

```
public class Superclass {
   public void printMethod() {   System.out.println("Printed in Superclass.");   }
}
```

Here is a subclass, called Subclass, that overrides printMethod():

```
public class Subclass extends Superclass {

   // overrides printMethod in Superclass
   public void printMethod() {      super.printMethod();    System.out.println("Printed in Subclass");    }
   public static void main(String[] args) {
      Subclass s = new Subclass();        s.printMethod();
   }
}
```

printMethod() refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to printMethod() inherited from Superclass, Subclass must use a qualified name, using **super** as shown. Compiling and executing Subclass prints the following:

```
Printed in Superclass.
Printed in Subclass
```

Subclass Constructors

Use the super keyword to invoke a superclass's constructor. MountainBike is a subclass of Bicycle. Here is the MountainBike (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

```
public MountainBike(int startHeight,  int startCadence,  int startSpeed,  int startGear) {
     super(startCadence, startSpeed, startGear);
     seatHeight = startHeight;
}
```

Invocation of a superclass constructor must be the **first line** in the subclass constructor.
The syntax for calling a superclass constructor is   super();   or: super(parameter list);

Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a **no-argument constructor**, you will get a **compile-time error**. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object. In fact, this is the case. It is called ***constructor chaining***, and you need to be aware of it when there is a long line of class descent.

Object as a Superclass (Not in Syllabus)

Every class is a descendant, direct or indirect, of the **Object** class. Every class you use or write inherits the instance methods of Object. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from Object that are discussed in this section are:

- protected Object **clone**() throws CloneNotSupportedException :   Creates and returns a copy of this object.
- public boolean **equals**(Object obj) :   Indicates whether some other object is "equal to" this one.
- protected void **finalize**() throws Throwable
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- public final Class **getClass**() :   Returns the runtime class of an object.
- public int **hashCode**() :   Returns a hash code value for the object.
- public String **toString**() :   Returns a string representation of the object.

The notify, notifyAll, and wait methods of Object all play a part in **synchronizing** the activities of independently running threads in a program. There are five of these methods:

- public final void notify()
- public final void notifyAll()
- public final void wait()
- public final void wait(long timeout)
- public final void wait(long timeout, int nanos)

The clone() Method

If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:
aCloneableObject.clone();

Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as

protected Object clone() throws CloneNotSupportedException
or:
public Object clone() throws CloneNotSupportedException

if you are going to write a clone() method to override the one in Object.

If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.

For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object *and* ObjExternal. Then the original object references ObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent.

The equals() Method

The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided by Object tests whether the object *references* are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Book class that overrides equals():

```
public class Book {
   ...
   public boolean equals(Object obj) {
      if (obj instanceof Book)       return ISBN.equals((Book)obj.getISBN());
      else           return false;
   }
}
```

Consider this code that tests two instances of the Book class for equality:

```
// Swing Tutorial, 2nd edition
Book firstBook  = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {     System.out.println("objects are equal");  }
else {    System.out.println("objects are not equal");  }
```

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.
You should always override the equals() method if the identity operator is not appropriate for your class.
Note: If you override equals(), you must override hashCode() as well.

The finalize() Method

The Object class provides a callback method, finalize(), that *may be* invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.
The finalize() method *may be* called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.

The getClass() Method

You cannot override getClass.
The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, 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().getSimpleName());
}
```

The Class class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.

The hashCode() Method

The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.

By definition, if two objects are equal, their hash code *must also* be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

The toString() Method

You should always consider overriding the toString() method in your classes.

The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

```
System.out.println(firstBook.toString());
```