Interfaces

Interfaces

- There are a number of situations when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. *interfaces* are such contracts.
- In the Java programming language, an *interface* is a reference type.
- Similar to a class, an *interface* can contain *only* constants, method signatures, default methods, static methods, and nested types.
- Method bodies exist only for default methods and static methods.
- Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Interfaces

```
public interface OperateCar {

   // constant declarations, if any

   // method signatures

   // An enum with values RIGHT, LEFT
   int turn(Direction direction,
            double radius,
            double startSpeed,
            double endSpeed);
   int changeLanes(Direction direction,
                   double startSpeed,
                   double endSpeed);
   int signalTurn(Direction direction,
                  boolean signalOn);
   int getRadarFront(double distanceToCar,
                     double speedOfCar);
   int getRadarRear(double distanceToCar,
                    double speedOfCar);
         ......
   // more method signatures
}
```

* Note that the method signatures have no braces and are terminated with a semicolon.

Interfaces

- To use an interface, you write a class that *implements* the interface.
- When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface.

```
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:
    int signalTurn(Direction direction, boolean signalOn) {
       // code to turn BMW's LEFT turn indicator lights on
       // code to turn BMW's LEFT turn indicator lights off
       // code to turn BMW's RIGHT turn indicator lights on
       // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes not
    // visible to clients of the interface
}
```

Defining an Interface

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

- The public access specifier indicates that the interface can be used by any class in any package.
- If you do not specify that the interface is public, then your interface is accessible only to classes defined in the same package as the interface.

Defining an Interface

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

- An interface can extend other interfaces, just as a class subclass or extend another class.
- However, whereas a class can extend only one other class, an interface can extend any number of interfaces.
- The interface declaration includes a comma-separated list of all the interfaces that it extends.

The Interface Body

- The interface body can contain abstract methods, default methods, and static methods.
  - An abstract method within an interface is followed by a semicolon, but no braces (an abstract method has no implementation).
  - Default methods are defined with the default modifier, and static methods with the static keyword.
  - All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.
- In addition, an interface can contain constant declarations.
- All constant values defined in an interface are implicitly public, static, and final. You can omit these modifiers.

Implementing an Interface

- To declare a class that implements an interface, you include an implements clause in the class declaration.
- Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.
- By convention, the implements clause follows the extends clause, if there is one.

Implementing an Interface

```
public interface Relatable {

    // this (object calling isLargerThan)
    // and other must be instances of
    // the same class returns 1, 0, -1
    // if this is greater than,
    // equal to, or less than other
    public int isLargerThan(Relatable other);
}
```

Any class can implement Relatable if there is some way to compare the relative "size" of objects instantiated from the class.

- For strings, it could be number of characters;
- for books, it could be number of pages;
- for students, it could be weight; and so forth.
- For planar geometric objects, area would be a good choice.

Implementing the Relatable Interface

```
public class RectanglePlus
    implements Relatable {
    public int width = 0; public int height = 0;   public Point origin;

    // four constructors
    public RectanglePlus() {
        origin = new Point(0, 0);
    }

    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);  width = w; height = h;
    }
    public RectanglePlus(Point p, int w, int h) {
        origin = p;   width = w;  height = h;
    }
```

Implementing the Relatable Interface

```
    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x; origin.y = y;
    }

    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }

    // a method required to implement the Relatable interface
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect
            = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1;
        else
            return 0;
    }
}
```

Using Interface as an Type

- When you define a new interface, you are defining a new reference data type.
- You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

Using Interface as an Type

```
public Object findLargest(Object object1, Object object2) {
      Relatable obj1 = (Relatable)object1;
      Relatable obj2 = (Relatable)object2;
      if ((obj1).isLargerThan(obj2) > 0)
         return object1;
      else
         return object2;
 }
```

By casting object1 to a Relatable type, it can invoke the isLargerThan method.

Evolving Interfaces

- Consider an interface that you have developed called DoIt:

```
public interface DoIt {
    void doSomething(int i, double x);
     int doSomethingElse(String s);
}
```

- Suppose that, at a later time, you want to add a third method to DoIt, so that the interface now becomes:

```
public interface DoIt {
    void doSomething(int i, double x);
     int doSomethingElse(String s);
     boolean didItWork(int i, double x, String s);
}
```

- All classes that implement the old DoIt interface will break because they no longer implement the old interface.

Evolving Interfaces

- One Solution:

```
public interface DoItPlus extends DoIt {
    boolean didItWork(int i, double x, String s);
}
```

- Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

- Alternatively, you can define your new methods as default methods. The following example defines a default method named didItWork:

```
public interface DoIt {
    void doSomething(int i, double x);
    int doSomethingElse(String s);
    default boolean didItWork(int i, double x, String s) {
        // Method body
    }
}
```

Default Methods

- Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.
- When Extending Interfaces That Contain Default Methods
  - Not mention the default method at all, which lets your extended interface inherit the default method.
  - Redeclare the default method, which makes it abstract.
  - Redefine the default method, which overrides it.
- In addition to default methods, you can define static methods in interfaces.
- All method declarations in an interface, including static methods, are implicitly public, so you can omit the public modifier.

Summary of Interfaces

- An interface declaration can contain method signatures, default methods, static methods and constant definitions. The only methods that have implementations are default and static methods.
- A class that implements an interface must implement all the methods declared in the interface.
- An interface name can be used anywhere a type can be used.

Inheritance

- 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*).
- Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance).
- In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

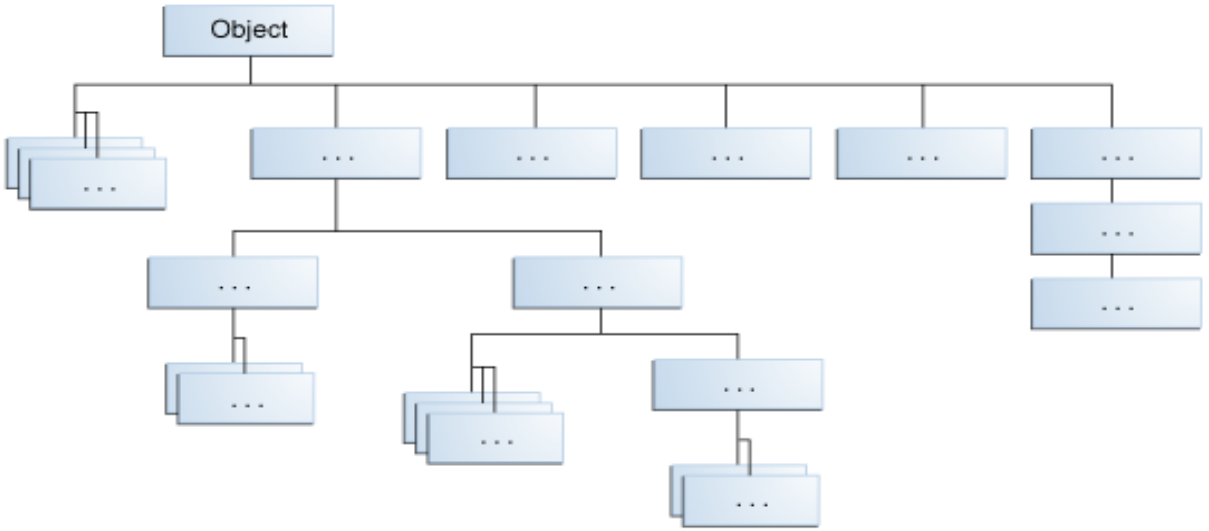
Inheritance

- Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object.
- Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to 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.

Inheritance

- A subclass inherits all the *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.

The Java Platform Class Hierarchy



- At the top of the hierarchy, Object is the most general of all classes.
- Classes near the bottom of the hierarchy provide more specialized behavior.

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;
    }

}
```

An Example of Inheritance

```
public class MountainBike extends Bicycle {

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

    // 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.

What You Can Do in a Subclass

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.

What You Can Do in a Subclass

- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- 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.

Private Members in a Superclass

- 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

- MountainBike is descended 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 Objects

- *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 . This is called *implicit casting (upcasting)*.

- 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.

Casting Objects

- However, we can *tell* the compiler that we promise to assign a MountainBike to obj by *explicit casting (downcasting):*

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike.

If obj is not a MountainBike at runtime, an exception will be thrown.

- You can make a logical test as to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast.For example:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Multiple Inheritance of State, Implementation, and Type

- One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot.
- In addition, you can instantiate a class to create an object, which you cannot do with interfaces.
- An object stores its state in fields, which are defined in classes.
- One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of *multiple inheritance of state*, which is the ability to inherit fields from multiple classes.

Multiple Inheritance of State

- For example, suppose that you are able to define a new class that extends multiple classes.
- When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses.
- What if methods or constructors from different superclasses instantiate the same field?
- Which method or constructor will take precedence?
- Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

Multiple Inheritance of Implementation

- *Multiple inheritance of implementation* is the ability to inherit method definitions from multiple classes.
- Problems arise with this type of multiple inheritance, such as name conflicts and ambiguity.
- When compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke.
- In addition, a programmer can unwittingly introduce a name conflict by adding a new method to a superclass.

Multiple Inheritance of Type

- The Java programming language supports *multiple inheritance of type*, which is the ability of a class to implement more than one interface.
- An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements.
- This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface.

Overriding and Hiding Methods

- An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.
- The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed.
- The overriding method has the same name, number and type of parameters, and return type as the method that it overrides.
- An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a *covariant return type*.

Covariant Return Types

```
class Grain {
     public String toString() { return "Grain"; }}
class Wheat extends Grain {
     public String toString() { return "Wheat"; }}
class Mill {
     Grain process() {  return new Grain(); }}
class WheatMill extends Mill {
     Wheat process() { return new Wheat(); }}

public class CovariantReturn {
    public static void main(String[] args) {
          Mill m = new Mill();
          Grain g = m.process();
          System.out.println(g);
          m = new WheatMill();
          g = m.process();
          System.out.println(g);
    }
}
```

```
/* Output:
Grain
Wheat
*///:~
```

Overriding and Hiding Methods

- When overriding a method, you might want to use the @Override annotation that 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 superclasses, then it will generate an error.

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.
- The distinction between hiding a static method and overriding an instance method has important implications:
  - 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.

Example

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

The Cat class overrides the instance method in Animal and hides the static method in Animal.

Static Methods

- The output from this program is as follows:

  The static method in Animal

  The instance method in Cat

- The version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

Interface Methods

- Default methods and abstract methods in interfaces are inherited like instance methods.
- However, 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.

Interface Methods

Instance methods are preferred over interface default methods.

```
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.

Interface Methods

Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

```
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());
    }
}
```

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

Interface Methods

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.

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

Interface Methods

A class that implements both OperateCar and FlyCar must override the method startEngine. You could invoke any 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 super interface that defines or inherits a default for the invoked method.

Interface Methods

Inherited instance methods from classes can override abstract interface methods.

```
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());
    }
}
```

The method Mustang.identifyMyself returns the string I am a horse.

Note: Static methods in interfaces are never inherited.

Modifiers

- 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 of Overriding and Hiding

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

Polymorphism

- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

```
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;
    }

    public void printDescription(){
        System.out.println("\nBike is " + "in gear " + this.gear
            + " with a cadence of " + this.cadence +
            " and travelling at a speed of " + this.speed + ". ");
    }
}
```

Polymorphism

```
public class MountainBike extends Bicycle {
    private String suspension;
    public MountainBike(
                int startCadence,
                int startSpeed,
                int startGear,
                String suspensionType){
        super(startCadence,
              startSpeed,
              startGear);
        this.setSuspension(suspensionType);
    }
    public String getSuspension(){
      return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }
    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

Polymorphism

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;
    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
                    int newTireWidth){
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }
    public int getTireWidth(){
      return this.tireWidth;
    }
    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }
   public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}
```

Polymorphism

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;
    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);
    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}
```

The following is the output from the test program:

```
Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.
```

- 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.
- The fact that an object variable can refer to multiple actual types is called polymorphism.

Dynamic Binding

- This behavior is referred to as *virtual method invocation* and demonstrates an aspect of the important polymorphism features in the Java language.
- Automaticaly selecting the appropriate method at runtime is called dynamic binding.
  - If the method is private, static, final, or a constructor, then the compiler knows exactly which method to call. This is called static binding.
  - Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at runtime.
- Dynamic binding has a very important property. It makes programs extensible without the need for modifying existing code.

Pitfall: “Overriding” Private Methods

```
//: polymorphism/PrivateOverride.java
// Trying to override a private method.
package polymorphism;
import static net.mindview.util.Print.*;

public class PrivateOverride {
  private void f() { print("private f()"); }
  public static void main(String[] args) {
    PrivateOverride po = new Derived();
    po.f();
  }
}

class Derived extends PrivateOverride {
  public void f() { print("public f()"); }
} /* Output:
private f()
*///:~
```

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.

Pitfall: Fields and Static Methods

```
//: polymorphism/FieldAccess.java
// Direct field access is determined at compile time.

class Super {
  public int field = 0;
  public int getField() { return field; }
}

class Sub extends Super {
  public int field = 1;
  public int getField() { return field; }
  public int getSuperField() { return super.field;}
}

public class FieldAccess {
  public static void main(String[] args) {
    Super sup = new Sub(); // Upcast
    System.out.println("sup.field = " + sup.field +
      ", sup.getField() = " + sup.getField());
    Sub sub = new Sub();
    System.out.println("sub.field = " +
      sub.field + ", sub.getField() = " +
      sub.getField() +
      ", sub.getSuperField() = " +
      sub.getSuperField());
  }
} /* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
*///:~
```

Pitfall: Fields and Static Methods

```
//: polymorphism/StaticPolymorphism.java
// Static methods are not polymorphic.
class StaticSuper {
  public static String staticGet() {
    return "Base staticGet()";
  }
  public String dynamicGet() {
    return "Base dynamicGet()";
  }
}
class StaticSub extends StaticSuper {
  public static String staticGet() {
    return "Derived staticGet()";
  }
  public String dynamicGet() {
    return "Derived dynamicGet()";
  }
}
public class StaticPolymorphism {
  public static void main(String[] args) {
    StaticSuper sup = new StaticSub(); // Upcast
    System.out.println(sup.staticGet());
    System.out.println(sup.dynamicGet());
  }
} /* Output:
Base staticGet()
Derived dynamicGet()
*///:~
```

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).

Superclass

```
public class Superclass {

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

Subclass

```
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();
    }
}
```

- Output:
  Printed in Superclass.
  Printed in Subclass

Subclass Constructors

```
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);
- With super(), the superclass no-argument constructor is called.
  With super(parameter list), the superclass constructor with a matching parameter list is called.

Subclass Constructors

- 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

- The Object class, in the java.lang package, sits at the top of the class hierarchy tree.
- The Object class is the ultimate ancestor. 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.

Object as a Superclass

- You can use a variable of type Object to refer to objects of any type:

```
Object obj = new Employee("Harry Hacker", 35000);
```

- To do anything specific with the value, you need to have some knowledge about the original type and apply a cast:

```
Employee e = (Employee) obj;
```

- In Java, only the primitive types are not objects.
- All array types, no matter whether they are arrays of objects or arrays of primitive types, are class types that extend the Object class.

```
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```

The toString Method

public String toString()

- Returns a string representation of the object. The result should be a concise but informative representation that is easy for a person to read.
- When concatenating a string with other types of data, the toString() method is automatically invoked .
  - ✓ For example, System.out.println(“student:=" + s1) ;
- The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.

The toString Method

```
public class TestToString {
        public static void main(String[] args) {
                Student s1 = new Student( );
                System.out.println("s1= " + s1);
        }
}

class Student{
}

// Output
s1= Student@1db9742
```

It is recommended that all subclasses override this method.

The toString Method

```
public class TestToString {
        public static void main(String[] args) {
                Student s1 = new Student( );
                System.out.println("s1= " + s1);
        }
}

class Student{
    public String toString(){
                return "Wei Liu";
        }

}
// Output
s1= Wei Liu
```

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 .

The equals() Method-Example

```
class Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      // a quick test to see if the objects are identical
      if (this == otherObject) return true;
      // must return false if the explicit parameter is null
      if (otherObject == null) return false;
      // if the classes don't match, they can't be equal
      if (getClass() != otherObject.getClass())
         return false;
      // now we know otherObject is a non-null Employee
      Employee other = (Employee) otherObject;
      // test whether the fields have identical values
      return name.equals(other.name)
         && salary == other.salary
         && hireDay.equals(other.hireDay);
   }
}
```

The equals() Method-Example

abstractClasses/Person.java

```
 1  package abstractClasses;
 2
 3  public abstract class Person
 4  {
 5     public abstract String getDescription();
 6     private String name;
 7
 8     public Person(String n)
 9     {
10        name = n;
11     }
12
13     public String getName()
14     {
15        return name;
16     }
17  }
```

The equals() Method-Example

abstractClasses/Employee.java

```
1  package abstractClasses;
2
3  import java.util.Date;
4  import java.util.GregorianCalendar;
5
6  public class Employee extends Person
7  {
8     private double salary;
9     private Date hireDay;
10
11    public Employee(String n, double s, int year, int month, int day)
12    {
13       super(n);
14       salary = s;
15       GregorianCalendar calendar = new GregorianCalendar(year, month
- 1, day);
16       hireDay = calendar.getTime();
17    }
18    public double getSalary()
19    {
20       return salary;
21    }
22
23    public Date getHireDay()
24    {
25       return hireDay;
26    }
27
28    public String getDescription()
29    {
30       return String.format("an employee with a salary of $%.2f",
salary);
31    }
32
33    public void raiseSalary(double byPercent)
34    {
35       double raise = salary * byPercent / 100;
36       salary += raise;
37    }
38 }
```

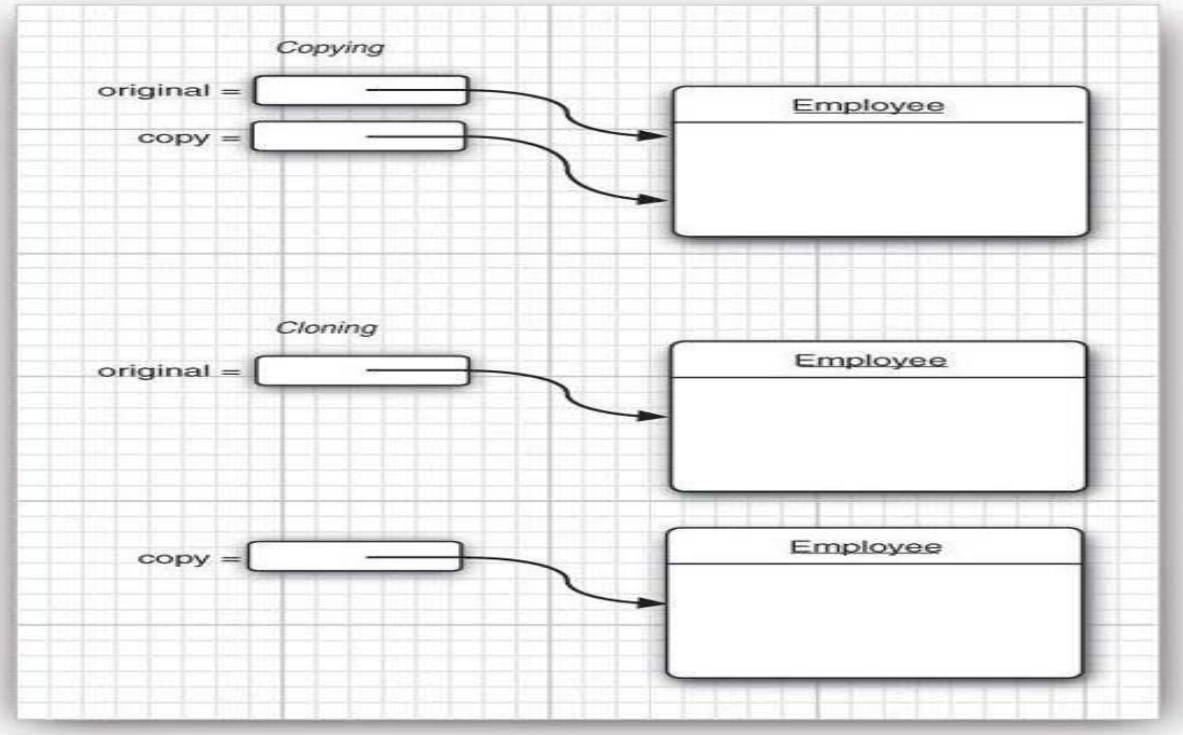
clone( ) Method

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed
```

If you would like copy to be a new object that begins its life being identical to original but whose state can diverge over time, use the clone method.

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

Copying and Cloning



Subtleties of clone() Method

- First, note that the clone method is ***protected*** in the Object class.
- This means that it is *protected* for subclasses as well.
- Hence, it cannot be called from within an Object of another class and package.
- To use the clone method, you must override in your subclass and upgrade visibility to *public*.

More Issues

- Also, any class that uses clone must implement the *Cloneable* interface.
  - This is a bit different from other interfaces that we've seen.
  - There are no methods; rather, it is used just as a marker of your intent.
  - The method that needs to be implemented is inherited from Object.
- Finally, clone throws a CloneNotSupportedException.
  - This is thrown if your class is not marked Cloneable.
  - This is all a little odd but you must handle this in subclass.

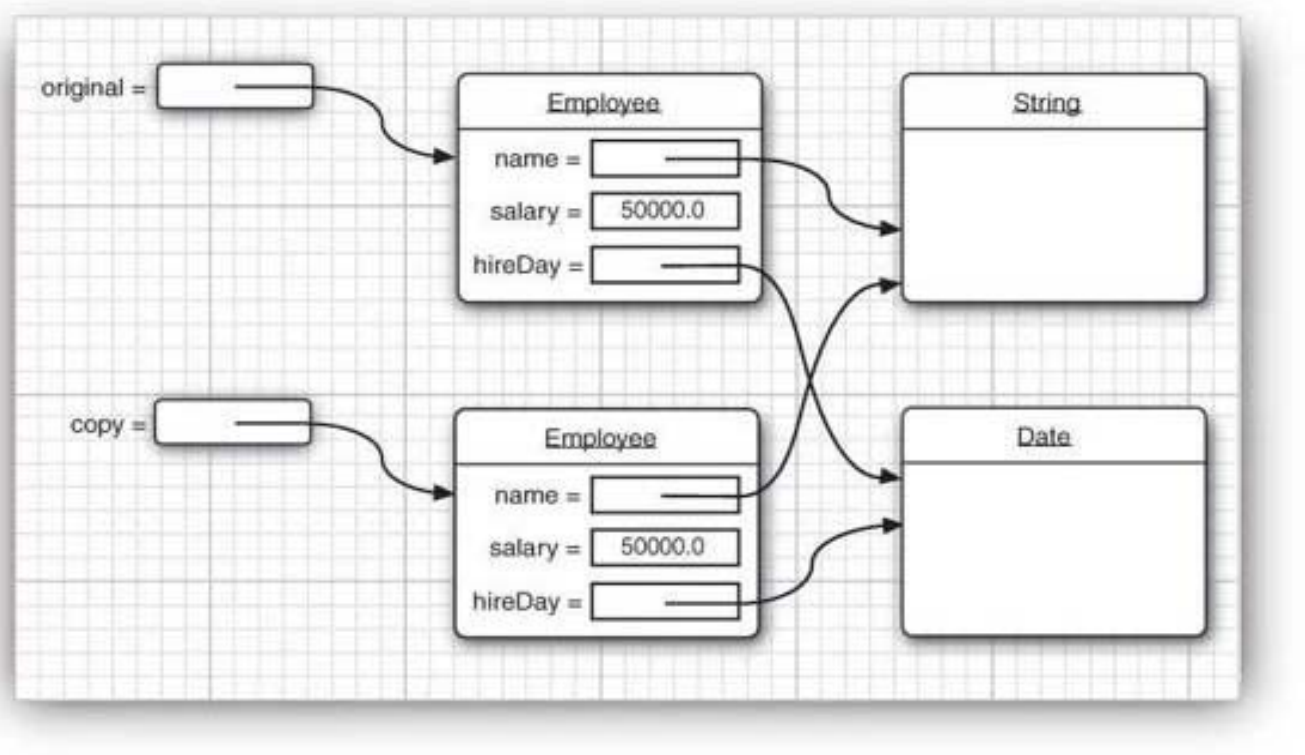
Steps for Cloning

- To reiterate, if you would like objects of class C to support cloning, do the following:
  - implement the Cloneable interface
  - override the clone method with public access privileges
  - call super.clone()
  - Handle CloneNotSupported Exception.
- This will get you default cloning, but more subtleties still lurk.

Shallow Copies

- We haven’t yet said what the default clone() method does.
- By default, clone makes a *shallow copy* of all data fields in a class.
- *Shallow copy* means that all data fields are copied in regular way. It doesn’t clone objects that are referenced inside other objects.
- This is not what you typically want. Must override clone to explicitly clone objects.

Shallow Copies



Immutable Objects

- A special class of Objects are called *immutable* because their state cannot be changed once set. A common example is String.
- Immutable object simplify programming in certain instances, such as when writing thread safe code.
- They also simplify cloning, since an object that cannot be changed doesn’t really need to be deep-copied.

Deep Copies

- For *deep copies* that recurse through the objects, you have to do some more work.
- super.clone() is first called to clone the first level of data fields.
- Returned cloned object’s object fields are then accessed one by one and clone method is called for each.

Example

```
class Employee implements Cloneable
{
   . . .
   public Employee clone() throws CloneNotSupportedException
   {
      // call Object.clone()
      Employee cloned = (Employee) super.clone();
      // clone mutable fields
      cloned.hireDay = (Date) hireDay.clone();
      return cloned;
   }
}
```

Additional clone() Properties

- Note that the following are typical, but not strictly required:
  - x.clone() != x;
  - x.clone().getClass() == x.getClass();
  - x.clone().equals(x);

- Finally, though no one really cares, Object does not support clone().

finalize() Method

- Called as final step when Object is no longer used, just before garbage collection
- Object version does nothing
- Since java has automatic garbage collection, finalize() does not need to be overridden to reclaim memory.
- Can be used to reclaim other resources – close streams, database connections, threads.
- However, it is strongly recommended *not* to rely on this for scarce resources.
- Be explicit and create own dispose method.

Writing Final Classes and Methods

- You can declare some or all of a class's methods *final*.
- You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses.
- The Object class does this—a number of its methods are final.
- Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.

Abstract Methods and Classes

- An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.
- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- If a class includes abstract methods, then the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject {
    // declare fields
     // declare nonabstract methods
     abstract void draw();
 }
```

Abstract Methods and Classes

- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.
- Methods in an *interface* that are not declared as default or static are *implicitly* abstract, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.)

Abstract Classes Compared to Interfaces

- Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation.
- However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods.
- With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public.
- In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

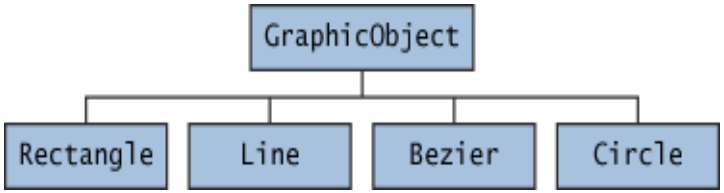
The Scenario of Using Abstract Classes

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields.
- This enables you to define methods that can access and modify the state of the object to which they belong.

The Scenario of Using Interface

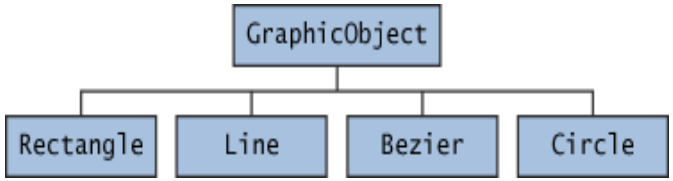
- You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

An Abstract Class Example




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

An Abstract Class Example




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

When an Abstract Class Implements an Interface

- It was noted that a class that implements an interface must implement all of the interface's methods.
- It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract.

```
abstract class X implements Y {
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
}
```

- In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

Class Members

- An abstract class may have static fields and static methods.
- You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with any other class.

Summary of Inheritance

- Except for the Object class, a class has exactly one direct superclass. A class inherits fields and methods from all its superclasses, whether direct or indirect.
- A subclass can override methods that it inherits, or it can hide fields or methods that it inherits. (Note that hiding fields is generally bad programming practice.)
- The table in Overriding and Hiding Methods section shows the effect of declaring a method with the same signature as a method in the superclass.

Summary of Inheritance

- The Object class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from Object include toString(), equals(), clone(), and getClass().
- You can prevent a class from being subclassed by using the final keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.
- An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods—methods that are declared but not implemented. Subclasses then provide the implementations for the abstract methods.