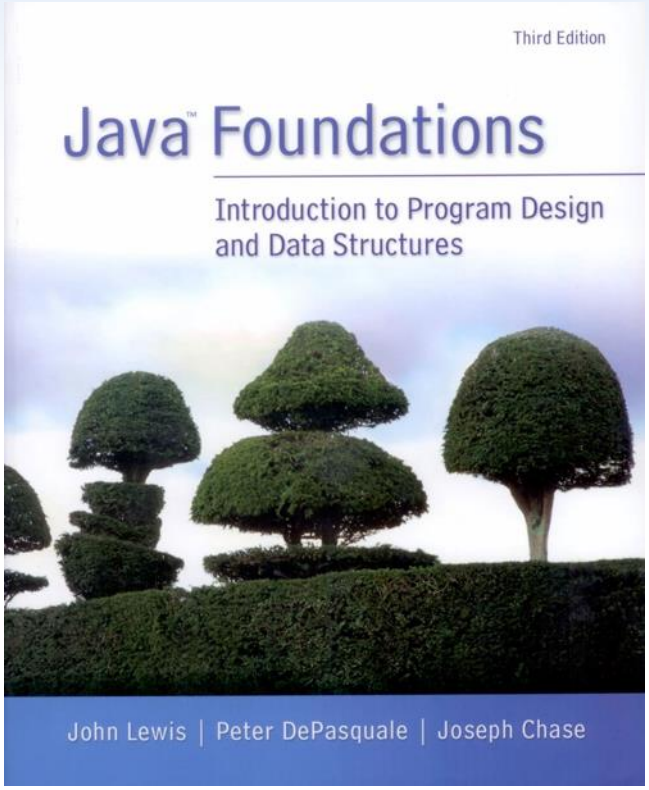


Chapter 9

Polymorphism

Chapter Scope

- The role of polymorphism
- Dynamic binding
- Using inheritance for polymorphism
- Exploring Java interfaces in more detail
- Using interfaces for polymorphism
- Polymorphic design

Binding

- Consider the following method invocation:

```
obj.doIt();
```

- At some point, this invocation is *bound* to the definition of the method that it invokes
- If this binding occurred at compile time, then that line of code would call the same method every time
- But Java defers method binding until run time; this is called *dynamic binding* or *late binding*

Polymorphism

- The term *polymorphism* literally means “having many forms”
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

Polymorphism

- Suppose we create the following reference variable

```
Occupation job;
```

- Java allows this reference to point to an `Occupation` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance

- For example, if the `Holiday` class is the parent of `Christmas`, then a `Holiday` reference could be used to point to a `Christmas` object

```
Holiday special = new Christmas();
```

References and Inheritance

- Assigning a child object to a parent reference is considered to be a widening conversion, and can be performed by simple assignment
- Assigning an parent object to a child reference can be done also, but it is considered a narrowing conversion and must be done with a cast
- The widening conversion is the most useful

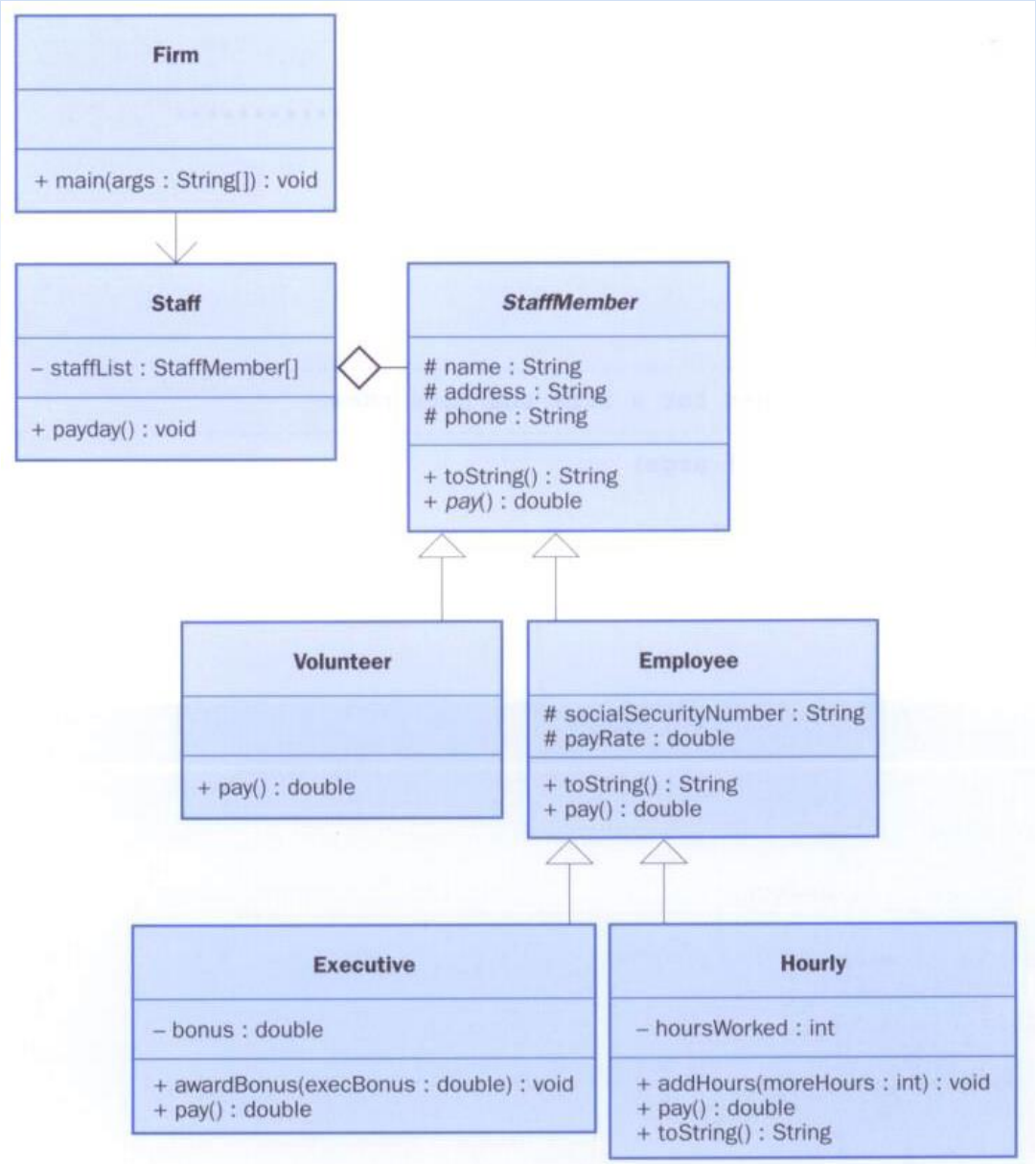
Polymorphism via Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Mammal` class has a method called `move`, and the `Horse` class overrides it
- Now consider the following invocation

```
pet.move();
```

- If `pet` refers to a `Mammal` object, it invokes the `Mammal` version of `move`; if it refers to a `Horse` object, it invokes the `Horse` version

- Let's look at an example that pays a set of employees using a polymorphic method



```
//********************************************************************
//  Firm.java       Java Foundations
//
//  Demonstrates polymorphism via inheritance.
//********************************************************************

public class Firm
{
   //-----------------------------------------------------------------
   //  Creates a staff of employees for a firm and pays them.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Staff personnel = new Staff();

      personnel.payday();
   }
}
```

```
//********************************************************************
//  Staff.java       Java Foundations
//
//  Represents the personnel staff of a particular business.
//********************************************************************

public class Staff
{
   private StaffMember[] staffList;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the list of staff members.
   //-----------------------------------------------------------------
   public Staff()
   {
      staffList = new StaffMember[6];

      staffList[0] = new Executive("Tony", "123 Main Line",
         "555-0469", "123-45-6789", 2423.07);

      staffList[1] = new Employee("Paulie", "456 Off Line",
         "555-0101", "987-65-4321", 1246.15);
      staffList[2] = new Employee("Vito", "789 Off Rocker",
         "555-0000", "010-20-3040", 1169.23);

      staffList[3] = new Hourly("Michael", "678 Fifth Ave.",
         "555-0690", "958-47-3625", 10.55);
```

```
      staffList[4] = new Volunteer("Adrianna", "987 Babe Blvd.",
         "555-8374");
      staffList[5] = new Volunteer("Benny", "321 Dud Lane",
         "555-7282");

      ((Executive)staffList[0]).awardBonus(500.00);

      ((Hourly)staffList[3]).addHours(40);
   }

   //-----------------------------------------------------------------
   //  Pays all staff members.
   //-----------------------------------------------------------------
   public void payday()
   {
      double amount;

      for (int count=0; count < staffList.length; count++)
      {
         System.out.println (staffList[count]);

         amount = staffList[count].pay();  // polymorphic

         if (amount == 0.0)
            System.out.println("Thanks!");
         else
            System.out.println("Paid: " + amount);

         System.out.println("-----------------------------------");
      }
   }
}
```

```
//********************************************************************
//  StaffMember.java       Java Foundations
//
//  Represents a generic staff member.
//********************************************************************

abstract public class StaffMember
{
   protected String name;
   protected String address;
   protected String phone;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this staff member using the specified
   //  information.
   //-----------------------------------------------------------------
   public StaffMember(String eName, String eAddress, String ePhone)
   {
      name = eName;
      address = eAddress;
      phone = ePhone;
   }
```

```
   //-----------------------------------------------------------------
   //  Returns a string including the basic employee information.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = "Name: " + name + "\n";

      result += "Address: " + address + "\n";
      result += "Phone: " + phone;

      return result;
   }

   //-----------------------------------------------------------------
   //  Derived classes must define the pay method for each type of
   //  employee.
   //-----------------------------------------------------------------
   public abstract double pay();
}
```

```
//********************************************************************
//  Volunteer.java       Java Foundations
//
//  Represents a staff member that works as a volunteer.
//********************************************************************

public class Volunteer extends StaffMember
{
   //-----------------------------------------------------------------
   //  Constructor: Sets up this volunteer using the specified
   //  information.
   //-----------------------------------------------------------------
   public Volunteer(String eName, String eAddress, String ePhone)
   {
      super(eName, eAddress, ePhone);
   }

   //-----------------------------------------------------------------
   //  Returns a zero pay value for this volunteer.
   //-----------------------------------------------------------------
   public double pay()
   {
      return 0.0;
   }
}
```

```
//********************************************************************
//  Employee.java       Java Foundations
//
//  Represents a general paid employee.
//********************************************************************

public class Employee extends StaffMember
{
   protected String socialSecurityNumber;
   protected double payRate;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this employee with the specified
   //  information.
   //-----------------------------------------------------------------
   public Employee(String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate)
   {
      super(eName, eAddress, ePhone);

      socialSecurityNumber = socSecNumber;
      payRate = rate;
   }
```

```
   //-----------------------------------------------------------------
   //  Returns information about an employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nSocial Security Number: " + socialSecurityNumber;

      return result;
   }

   //-----------------------------------------------------------------
   //  Returns the pay rate for this employee.
   //-----------------------------------------------------------------
   public double pay()
   {
      return payRate;
   }
}
```

```
//********************************************************************
//  Executive.java       Java Foundations
//
//  Represents an executive staff member, who can earn a bonus.
//********************************************************************

public class Executive extends Employee
{
   private double bonus;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this executive with the specified
   //  information.
   //-----------------------------------------------------------------
   public Executive(String eName, String eAddress, String ePhone,
                    String socSecNumber, double rate)
   {
      super(eName, eAddress, ePhone, socSecNumber, rate);

      bonus = 0;  // bonus has yet to be awarded
   }

   //-----------------------------------------------------------------
   //  Awards the specified bonus to this executive.
   //-----------------------------------------------------------------
   public void awardBonus(double execBonus)
   {
      bonus = execBonus;
   }
```

```
   //-----------------------------------------------------------------
   //  Computes and returns the pay for an executive, which is the
   //  regular employee payment plus a one-time bonus.
   //-----------------------------------------------------------------
   public double pay()
   {
      double payment = super.pay() + bonus;

      bonus = 0;

      return payment;
   }
}
```

```
//********************************************************************
//  Hourly.java       Java Foundations
//
//  Represents an employee that gets paid by the hour.
//********************************************************************

public class Hourly extends Employee
{
   private int hoursWorked;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this hourly employee using the specified
   //  information.
   //-----------------------------------------------------------------
   public Hourly(String eName, String eAddress, String ePhone,
                 String socSecNumber, double rate)
   {
      super(eName, eAddress, ePhone, socSecNumber, rate);

      hoursWorked = 0;
   }
```

```
//-----------------------------------------------------------------
//  Adds the specified number of hours to this employee's
//  accumulated hours.
//-----------------------------------------------------------------
public void addHours(int moreHours)
{
   hoursWorked += moreHours;
}

//-----------------------------------------------------------------
//  Computes and returns the pay for this hourly employee.
//-----------------------------------------------------------------
public double pay()
{
   double payment = payRate * hoursWorked;

   hoursWorked = 0;

   return payment;
}
```

```
   //-----------------------------------------------------------------
   //  Returns information about this hourly employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nCurrent hours: " + hoursWorked;

      return result;
   }
}
```

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2(float value, char ch);
   public boolean doTheOther(int num);
}
```

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class states that it implements an interface, it must define all methods in the interface

Interfaces

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }

   // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Interfaces

- A class that implements an interface can implement other methods as well
- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants

```
//********************************************************************
//  Encryptable.java       Java Foundations
//
//  Represents the interface for an object that can be encrypted
//  and decrypted.
//********************************************************************

public interface Encryptable
{
   public void encrypt();
   public String decrypt();
}
```

```
//********************************************************************
//  Secret.java       Java Foundations
//
//  Represents a secret message that can be encrypted and decrypted.
//********************************************************************

import java.util.Random;

public class Secret implements Encryptable
{
   private String message;
   private boolean encrypted;
   private int shift;
   private Random generator;

   //-----------------------------------------------------------------
   //  Constructor: Stores the original message and establishes
   //  a value for the encryption shift.
   //-----------------------------------------------------------------
   public Secret(String msg)
   {
      message = msg;
      encrypted = false;
      generator = new Random();
      shift = generator.nextInt(10) + 5;
   }
```

```
//-----------------------------------------------------------------
//  Encrypts this secret using a Caesar cipher. Has no effect if
//  this secret is already encrypted.
//-----------------------------------------------------------------
public void encrypt()
{
   if (!encrypted)
   {
      String masked = "";
      for (int index=0; index < message.length(); index++)
         masked = masked + (char)(message.charAt(index)+shift);
      message = masked;
      encrypted = true;
   }
}
```

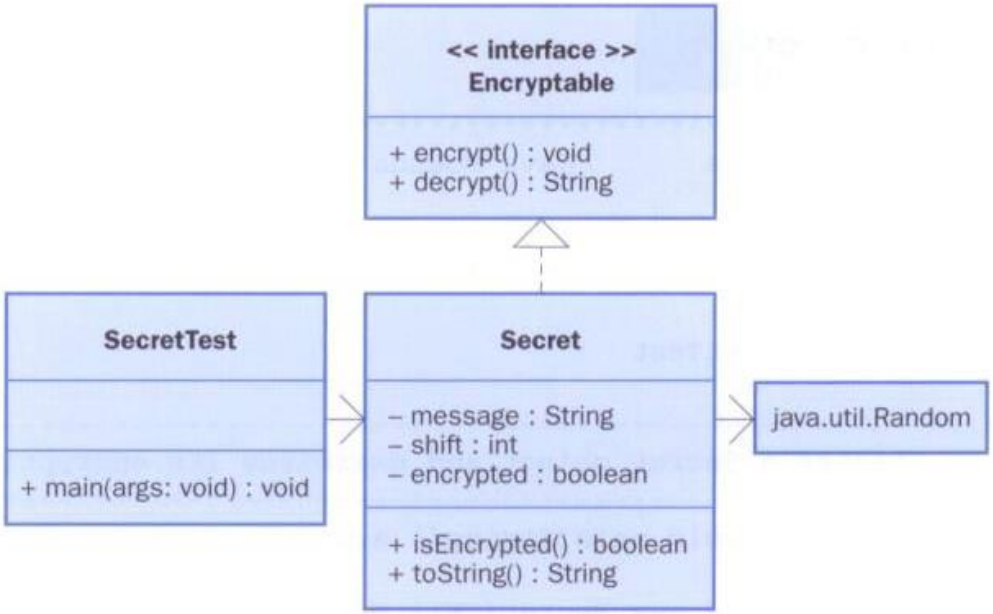
```
//-----------------------------------------------------------------
//  Decrypts and returns this secret. Has no effect if this
//  secret is not currently encrypted.
//-----------------------------------------------------------------
public String decrypt()
{
   if (encrypted)
   {
      String unmasked = "";
      for (int index=0; index < message.length(); index++)
         unmasked = unmasked + (char)(message.charAt(index)-shift);
      message = unmasked;
      encrypted = false;
   }

   return message;
}
```

```
   //-----------------------------------------------------------------
   //  Returns true if this secret is currently encrypted.
   //-----------------------------------------------------------------
   public boolean isEncrypted()
   {
      return encrypted;
   }

   //-----------------------------------------------------------------
   //  Returns this secret (may be encrypted).
   //-----------------------------------------------------------------
   public String toString()
   {
      return message;
   }
}
```

```
//********************************************************************
//  SecretTest.java       Java Foundations
//
//  Demonstrates the use of a formal interface.
//********************************************************************

public class SecretTest
{
   //-----------------------------------------------------------------
   //  Creates a Secret object and exercises its encryption.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Secret hush = new Secret("Wil Wheaton is my hero!");
      System.out.println(hush);

      hush.encrypt();
      System.out.println(hush);

      hush.decrypt();
      System.out.println(hush);
   }
}
```

Interfaces

- In UML, a dotted arrow is used to show that a class implements an interface
- The designation `<<interface>>` is used to indicate an interface



Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements Interface1, Interface2
{
   // all methods of both interfaces
}
```

Interfaces

- The Java API contains many helpful interfaces
- The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects
- We discussed the `compareTo` method of the `String` class in Chapter 4
- The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order

The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
if (obj1.compareTo(obj2) < 0)
   System.out.println("obj1 is less than obj2");
```

- The value returned from `compareTo` should be negative is `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`
- When you design a class that implements the `Comparable` interface, it should follow this intent

The Comparable Interface

- It's up to the programmer to determine what makes one object less than another
- For example, you may define the `compareTo` method of an `Employee` class to order employees by name (alphabetically) or by employee number
- The implementation of the method can be as straightforward or as complex as needed for the situation

The Iterator Interface

- As we discussed in Chapter 4, an iterator is an object that provides a means of processing a collection of objects one at a time
- An iterator is created formally by implementing the `Iterator` interface, which contains three methods
- The `hasNext` method returns a boolean result – true if there are items left to process
- The `next` method returns the next object in the iteration
- The `remove` method removes the object most recently returned by the `next` method

The Iterator Interface

- By implementing the `Iterator` interface, a class formally establishes that objects of that type are iterators
- The programmer must decide how best to implement the iterator functions
- Once established, the for-each version of the `for` loop can be used to process the items in the iterator

Interfaces

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways
- Which brings us back to polymorphism

References and Interfaces

- Suppose we have an interface called `Speaker`:

```
public interface Speaker
{
    public void speak();
    public void announce(String str);
}
```

- The interface name can now be used as the type of a reference variable:

```
Speaker current;
```

- The variable `current` can now point to any object of any class that implements `Speaker`

Polymorphism via Interfaces

- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing:

```
current.speak();
```

- This is analogous to the technique for polymorphism using inheritance

Polymorphism via Interfaces

- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```

Event Processing

- Polymorphism plays an important role in the development of a Java graphical user interface
- As we've seen, we establish a relationship between a component and a listener:

```
JButton button = new JButton();
button.addActionListener(new MyListener());
```

- Note that the `addActionListener` method is accepting a `MyListener` object as a parameter
- We can pass any object that implements the `ActionListener` interface to the `addActionListener` method

Event Processing

- The source code for the `addActionListener` method accepts a parameter of type `ActionListener` (the interface)
- Because of polymorphism, any object that implements that interface is compatible with the parameter reference variable
- The component can call the `actionPerformed` method because of the relationship between the listener class and the interface
- Extending an adapter class to create a listener represents the same situation; the adapter class implements the appropriate interface already