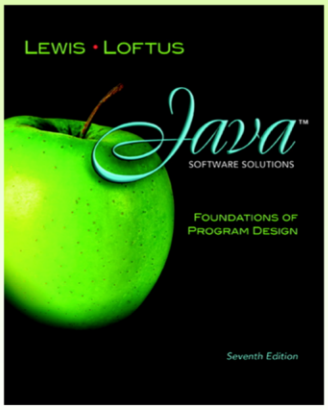
Chapter 10
Polymorphism



Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

Polymorphism

- Polymorphism is an object-oriented concept that allows us to create versatile software designs
- Chapter 10 focuses on:
  - defining polymorphism and its benefits
  - using inheritance to create polymorphic references
  - using interfaces to create polymorphic references
  - using polymorphism to implement sorting and searching algorithms
  - additional GUI components

Outline

- **Polymorphism and Late Binding**
- **Polymorphism via Inheritance**
- **Polymorphism via Interfaces**
- **Sorting**
- **Searching**
- **Event Processing Revisited**
- **File Choosers and Color Choosers**
- **Sliders**

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 called through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

-When we use classes from inheritance hierarchies, we can make use of a concept called **polymorphism**

-Polymorphism can be broadly defined as something that can change (e.g. its form or behavior)

-In object-oriented programming, it describes how reference variables can “change behavior”

-Practically speaking, we’ll see how reference variables can call different method implementations of the same name!

-We’ll see this is accomplished because an object reference variable can actually point to **different** types of objects within a hierarchy

-Before we see this in action, let’s first review an object reference variable

-Recall that an object reference variable stores an address to some memory location where the object lives

-Recall that the **new** operator allocates memory and returns the address where it is allocated

-This is “where the actual object lives”; the reference variable stores this address to remember where it is

-Think of a reference variable as a “pointer” since its contents (the address) **points** to the object in memory

-When we declare a reference variable we **normally** assign it the address of an object of the same variable type

-For example, below we assign the address of a Random object to a Random object reference variable:

```
Random r = new Random();
```

Polymorphism

- Suppose we create the following reference variable:

```
Occupation job;
```

- This reference can 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

-Returning to our example, below we assign the address of a Random object to a Random object reference variable:

```
Random r = new Random();
```

-This is how we've been declaring and assigning object reference variables up until now

-When we work with hierarchies, however, we can actually assign an address of a child object to a parent reference variable!

-For example, if we had a parent class named Shape and a child class named Circle, we could do the following

```
Shape s = new Circle();
```

-Here we are assigning the address of a child (Circle) in the parent (Shape) reference variable

-(Not the opposite of assigning a parent to a child is possible, but only useful in specific situations)

-We'll see that this special relationship above between parent and child classes makes polymorphism possible

Binding

- Consider the following method invocation:

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


-Returning to our example, we assign a child object to a base class reference variable:

```
Shape s = new Circle();
```

-Suppose we have a method in the parent (Shape) class named doIt():

```
public void doIt()
{
    System.out.println("I'm a shape");
}
```

-Suppose also, we have a method in our child (Circle) class with the same name

```
public void doIt()
{
    System.out.println("I'm a circle");
}
```

-Recall that when we have the same method name in the child as the parent, we are **overriding** the method

Binding

- 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
- However, Java defers method binding until run time -- this is called *dynamic binding* or *late binding*

-Now, let's suppose we performed the following in a main method and examined the output as shown

```
Shape s = new Circle();
s.doIt();
Output: "I'm a circle"
```

-When the s.doIt statement is executed, the doIt method from the Circle class is executed, NOT the Shape class!
-This is an example of polymorphism since the Shape variable "changed" its behavior in the program
-More specifically, the Shape reference variable called a different doIt method instead of its own
-In this way, we say that this reference variable is **polymorphic** (ability to change)
-Officially, the term used to describe which doIt method is actually called (parent or child) is called **binding**
-Note that the Shape reference variable could change what it is pointing to as the program is running
-It might later store the address of a Triangle object for example
-For this reason, the decision as to which doIt to call can only be made while the **program is running**
-This decision is made **dynamically**, at the exact time the call is made while the program is running (**run-time**)
-For this reason, we call this type of binding, **dynamic binding or late binding**

Outline

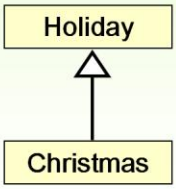
- Late Binding
- **Polymorphism via Inheritance**
- Polymorphism via Interfaces
- Sorting
- Searching
- Event Processing Revisited
- File Choosers and Color Choosers
- Sliders

-We can see polymorphism in action both when using classes in a hierarchy (e.g. via Inheritance)

-Later, we'll see polymorphism in action both when using interfaces in a hierarchy (e.g. via Interfaces)

References and Inheritance

- An object reference can refer to an object of any class related to it by inheritance
- For example, if `Holiday` is the superclass of `Christmas`, then a `Holiday` reference could be used to refer to a `Christmas` object




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


-In addition to our Shape and Circle example, let's look at another example of polymorphism via Inheritance

References and Inheritance

- These type compatibility rules are just an extension of the is-a relationship established by inheritance
- Assigning a `Christmas` object to a `Holiday` reference is fine because Christmas is-a holiday
- Assigning a child object to a parent reference can be performed by simple assignment
- Assigning an parent object to a child reference can be done also, but must be done with a cast
- After all, Christmas is a holiday but not all holidays are Christmas

Polymorphism via Inheritance

- Now suppose the `Holiday` class has a method called `celebrate`, and `Christmas` overrides it
- What method is invoked by the following?

```
day.celebrate();
```

- The type of the object being referenced, not the reference type, determines which method is invoked
- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes that version

Polymorphism via Inheritance

- Note that the compiler restricts invocations based on the type of the reference
- So if `Christmas` had a method called `getTree` that `Holiday` didn't have, the following would cause a compiler error:

```
day.getTree();  // compiler error
```

- Remember, the compiler doesn't "know" which type of holiday is being referenced
- A cast can be used to allow the call:

```
((Christmas)day).getTree();
```

Quick Check

If `MusicPlayer` is the parent of `CDPlayer`, are the following assignments valid?

```
MusicPlayer mplayer = new CDPlayer();
```

```
CDPlayer cdplayer = new MusicPlayer();
```

Quick Check

If `MusicPlayer` is the parent of `CDPlayer`, are the following assignments valid?

```
MusicPlayer mplayer = new CDPlayer();
```

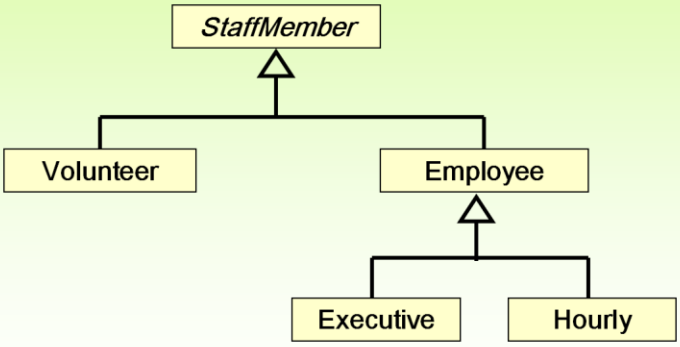
Yes, because a `CDPlayer` is-a `MusicPlayer`

```
CDPlayer cdplayer = new MusicPlayer();
```

No, you'd have to use a cast (and you shouldn't knowingly assign a super class object to a subclass reference)

Polymorphism via Inheritance

- Consider the following class hierarchy:



-Let's consider another example of polymorphism via inherited classes

-This examples has an abstract method in the parent class

-We'll see that polymorphism works the same with abstract methods

Polymorphism via Inheritance

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

- See `Firm.java`
- See `Staff.java`
- See `StaffMember.java`
- See `Volunteer.java`
- See `Employee.java`
- See `Executive.java`
- See `Hourly.java`

-As in our other examples, this demonstrates polymorphism by using parent references to store child objects

-When an overridden method (pay()) is called, dynamic binding determines which child method it should call

-Note also, that this overridden method is defined to be abstract in the parent (StaffMember) class

-Dynamic binding and polymorphism work the same way with abstract methods

-In fact, this is typically how we utilize polymorphism, with parent classes containing abstract methods!

```
//********************************************************************
//  Firm.java       Author: Lewis/Loftus
//
//  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();
   }
}
```

Output

```
Name: Sam
Address: 123 Main Line
Phone: 555-0469
Social Security Number: 123-45-6789
Paid: 2923.07
-----------------------------------
Name: Carla
Address: 456 Off Line
Phone: 555-0101
Social Security Number: 987-65-4321
Paid: 1246.15
-----------------------------------
Name: Woody
Address: 789 Off Rocker
Phone: 555-0000
Social Security Number: 010-20-3040
Paid: 1169.23
-----------------------------------
```

Output (continued)

```
Name: Diane
Address: 678 Fifth Ave.
Phone: 555-0690
Social Security Number: 958-47-3625
Current hours: 40
Paid: 422.0
-----------------------------------
Name: Norm
Address: 987 Suds Blvd.
Phone: 555-8374
Thanks!
-----------------------------------
Name: Cliff
Address: 321 Duds Lane
Phone: 555-7282
Thanks!
-----------------------------------
```

```
//********************************************************************
//  Staff.java       Author: Lewis/Loftus
//
//  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];
```

continue

-Note this class contains an array of parent reference variables

-The parent, in this example, is the StaffMember

continue

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

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

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

      staffList[4] = new Volunteer ("Norm", "987 Suds Blvd.",
         "555-8374");
      staffList[5] = new Volunteer ("Cliff", "321 Duds Lane",
         "555-7282");

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

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

continue

-Note how you can explicitly cast one of the parent reference variables

((Executive)staffList[0]).awardBonus(…)

-A more readable version of the statement above is:

Executive exec = (Executive)staffList[0];
exec.awardBonus(…);

-This type-casting is necessary because the awardBonus method is **only** in the Executive class (not the parent)
-If we tried the following, we'd get an error because this method doesn't exist in the parent (StaffMember) class

staffList[0].awardBonus(…);

continue

```
   //-----------------------------------------------------------------
   //  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 ("-----------------------------------");
      }
   }
}
```


-Note how the loop just goes through the staff members and calls the abstract method pay()

-This is where dynamic binding and polymorphism is happening!

-Each staffList reference variable is storing an address to a different type of staff member

-Depending on which type of staff member is being stored in each parent reference, a different pay() method is called!

```
//********************************************************************
//  StaffMember.java       Author: Lewis/Loftus
//
//  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;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  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       Author: Lewis/Loftus
//
//  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;
   }
}
```


-Note how we can pass arguments into our parent class from the child class constructor using **super**

-Remember **super** is a reference to the immediate parent of the current class

super(eName, eAddress, ePhone);

-Note how we are overriding the abstract pay method here to describe how a **Volunteer** gets paid

```
//********************************************************************
//  Employee.java       Author: Lewis/Loftus
//
//  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;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  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;
   }
}
```


-Note how we are overriding the abstract pay method here to describe how an **Employee** gets paid

```
//********************************************************************
//  Executive.java       Author: Lewis/Loftus
//
//  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
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  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;
   }
}
```


-Note how we are overriding the abstract pay method here to describe how an **Executive** gets paid

```
//********************************************************************
//  Hourly.java       Author: Lewis/Loftus
//
//  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;
   }
```

continue

continue

```
//-----------------------------------------------------------------
//  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;
}
```

continue

-Note how we are overriding the abstract pay method here to describe how an **Hourly** gets paid

continue

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

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

      return result;
   }
}
```