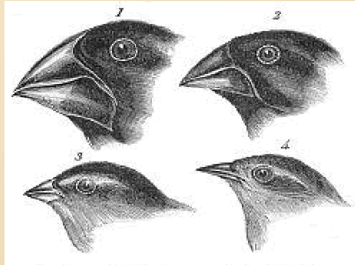
Introduction to Computer Science
Shimon Schocken
IDC Herzliya

Lectures 13-1, 13-2

Polymorphism



Polymorphism

Java is a "dynamic language":

- Run-time object types
- Virtual method invocation (aka "late binding" / "dynamic binding")

Implication: Polymorphism

- The behavior obtained by invoking `x.m()` can take a different form according to the run-time type of object x
- Therefore, objects belonging to different types can respond to a method call of the same name, each according to a different type-specific implementation
- The calling program does not have to know the object type in advance; The exact behavior is determined in run-time.

What are the problems to which polymorphism is the solution?

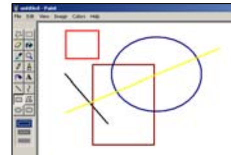
Quite often we have to represent a collection of objects of different types that have to have a similar but different behavior. Examples:

Payroll application:

- Different Worker types: fixed salary, hourly-workers, volunteers, …
- Common behavior: we have to pay each Worker according to his sub-type

Computer game:

- Different Fighter types: boxers, ninjas, shooters, …
- Common behavior: every Fighter hits in some sub-type specific way



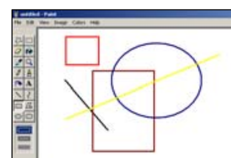
Paintbrush application:

- Different Figure types: lines, rectangles, circles, …
- Common behavior: every figure draws itself in some sub-type specific way

A polymorphic design approach

1. Design a base class, or an interface
2. Implement each sub-type as:
   - A class that extends the base-class, or
   - A class that implements the interface
3. Represent the common behavior as a method at the base-class or at the interface level
4. Have each sub-class implement this method in a sub-type specific way
5. This design allows you to invoke the same method on any object, knowing that the object will know how "to handle itself".

- This is the essence of polymorphism.



Example: zoo

```
abstract class Animal {
  abstract public String sound();
}

class Dog extends Animal {
  public String sound() {
    return "Woof";
  }
}

class Pig extends Animal {
  public String sound() {
    return "Oink";
  }
}

class Mouse extends Animal {
  public String sound() {
    return "Squeak";
  }
  public String complain() {
    return "Ouch!";
  }
}
```

```
class AnimalDemo {
  public static void main(String[] args) {
    Animal[] zoo = new Animal[4];
    zoo[0] = new Pig();
    zoo[1] = new Pig();
    zoo[2] = new Mouse();
    zoo[3] = new Dog();

    for (int i = 0; i < zoo.length; i++){
      Animal a = zoo[i];
      System.out.println(a.sound());
      if (a instanceof Mouse) {
        Mouse m = (Mouse) a;
        System.out.println(m.complain());
      }
    }
  }
}
```

Can store any sub-type of Animal

Polymorphic method invocation

```
C:\WINDOWS\system32\cmd.exe
D:\demo>java AnimalDemo
Oink
Oink
Squeak
Ouch!
Woof
```


Outline

- Polymorphism

Examples:

- Fighting army
- Payroll
- Paintbrush

Related topics:

- Polymorphic method calls
- Abstract class vs. interface
- Heterogeneous collections
- Generic sorters

Abstract class vs. interface

```
abstract class Animal {
  abstract public String sound();
}

class Dog extends Animal {
  public String sound() {
    return "Woof";
  }
}
...
}
```

Has the same effect as:

```
interface Animal {
  String sound();
}

class Dog implements Animal {
  public String sound() {
    return "Woof";
  }
}
...
}
```

An abstract class with abstract methods only is the same as an interface

- Use abstract classes when you want to declare some data and behavior at the base-class level and then have each sub-class declare its own "share" of data and behavior separately
- Use interfaces when you want to enforce design obligations that other classes must obey.

Example: zoo, take 2

```
Interface Animal {
  String sound();
}

class Dog implements Animal {
  public String sound() {
    return "Woof";
  }
}

class Pig implements Animal {
  public String sound() {
    return "Oink";
  }
}

class Mouse implements Animal {
  public String sound() {
    return "Squeak";
  }
}
```

Same as before

```
class AnimalDemo {
  public static void main(String[] args) {
    Animal[] zoo = new Animal[4];
    zoo[0] = new Pig();
    zoo[1] = new Pig();
    zoo[2] = new Mouse();
    zoo[3] = new Dog();

    for (int i = 0; i < zoo.length; i++){
      Animal a = zoo[i];
      System.out.println(a.sound());
      if (a instanceof Mouse) {
        Mouse m = (Mouse) a;
        System.out.println(m.complain());
      }
    }
  }
}
```

An array of objects that implement the Animal interface.

Polymorphic method invocation

C:\WINDOWS\system32\cmd.exe

```
D:\demo>java AnimalDemo
Oink
Oink
Squeak
Ouch!
Woof
```

Outline

- Polymorphism

Examples:

- Fighting army
- Payroll
- Paintbrush

Related topics:

- Polymorphic method calls
- Abstract class vs. interface
- Heterogeneous collections
- Generic sorters

A fighting army

C:\WINDOWS\system32\cmd.exe

```
D:\demo>java FightingArmy
Soldier 0:  trach! trach! trach!
Soldier 1:  trach! trach!
Soldier 2:  left punch! right punch!
Soldier 3:  trach! trach!
Soldier 4:  left punch! right punch! left punch!
Soldier 5:  trach!
Soldier 6:  left punch!
Soldier 7:  left punch!
Soldier 8:  left punch! right punch!
Soldier 9:  trach! trach! trach!
```

Kung Fu Fighter

```
public interface Fighter {
  public void hit();
}
```

```
public class KungFuFighter implements Fighter {

  public void hit() {
     System.out.print("trach! ");
  }

}
```

Boxer Fighter

```
public interface Fighter {
  public void hit();
}
```

```
// Represents a left- or right-handed boxer.
public class Boxer implements Fighter {

  private boolean nextPunchLeft;

  // Constructs either a left- or a right-handed Boxer
  public Boxer(boolean leftHanded) {
    nextPunchLeft = leftHanded;
  }

  public void hit() {
    System.out.print(nextPunchLeft ?
                     "left punch! " : "right punch! ");
    nextPunchLeft = !nextPunchLeft;
  }

}
```

A fighting army

```
import java.util.Random;
public class FightingArmy {
  public static void main(String[] args) {

    // Creates and populates an army of 10 fighters
    Fighter[] soldiers = new Fighter[10];
    for ( int i = 0 ; i < 10 ; i++ )
      if ( Math.random() > 0.5 )
        soldiers[i] = new Boxer(true);
      else
        soldiers[i] = new KungFuFighter();

    // For each fighter, prints his number and
    // generates a random series of at most 4 hits
    for ( int i = 0 ; i < 10 ; i++ ) {
      System.out.print("Soldier " + i + ":  ");

      int nHits = 1 + (new Random()).nextInt(3);
      for (int k = 0 ; k < nHits ; k++)
        soldiers[i].hit();
        System.out.println();
    }
  }
}
```

Polymorphic method invocation

<u>OO terminology</u>

Method calls, e.g. `x.m()`, are sometimes referred to as "sending a message `m()` to the object `x`"

Different objects respond to the same message in different ways, depending on their type.

```
C:\WINDOWS\system32\cmd.exe
D:\demo>java FightingArmy
Soldier 0:  trach! trach! trach!
Soldier 1:  trach! trach!
Soldier 2:  left punch! right punch!
Soldier 3:  trach! trach!
Soldier 4:  left punch! right punch! left punch!
Soldier 5:  trach!
Soldier 6:  left punch!
Soldier 7:  left punch!
Soldier 8:  left punch! right punch!
Soldier 9:  trach! trach! trach!
```


Outline

- Polymorphism

<u>Examples:</u>

- Fighting army
- ➡ Payroll
- Paintbrush

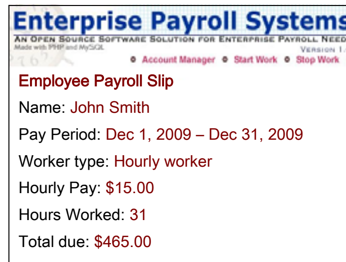
<u>Related topics:</u>

- Polymorphic method calls
- Abstract class vs. interface
- Heterogeneous collections
- Generic sorters

Payroll application

Our company employees various types of workers. We have employees, who are paid a monthly salary, we have hourly workers, who we pay according to the hours they actually worked, and we have volunteers, who don't get paid. We also have executives. The executives are employees, meaning that they get a monthly salary. But, they may also get a monthly bonus, that reflects their achievements during the month.

We need a payroll system that, each month, will pay each worker his or her due."



Design considerations

Identifying entities:

Our company employees various types of workers. We have employees, who are paid a monthly salary, we have hourly workers, who we pay according to the hours they actually worked, and we have volunteers, who don't get paid. We also have executives. The executives are employees, meaning that they get a monthly salary. But, they may also get a monthly bonus, that reflects their achievements during the month.

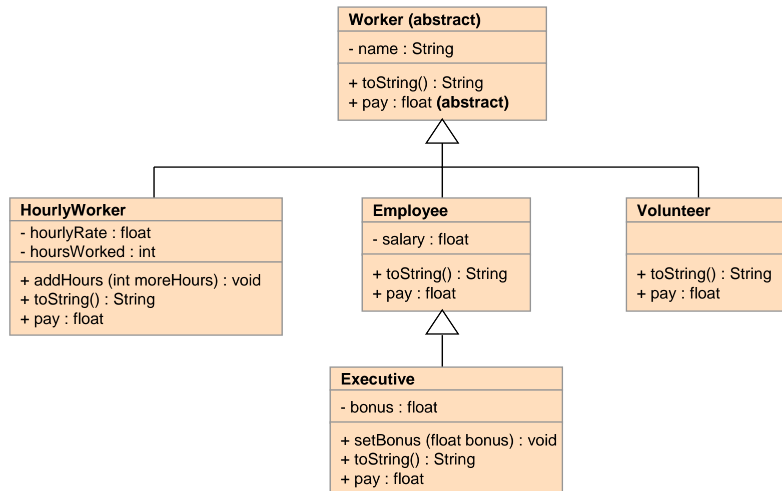
Observations:

- Employee is a worker
- Hourly-worker is a worker
- Volunteer is a worker
- Executive is an employee

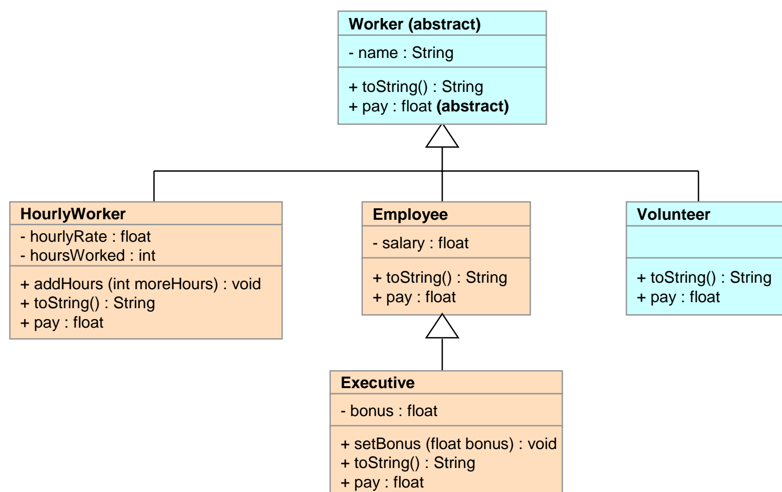
Design decision:

It makes sense to put as much common data and functionality in a Worker class, and derive specific worker sub-classes from it.

Payroll class diagram



Payroll class diagram



Sub-classing Worker: Volunteer

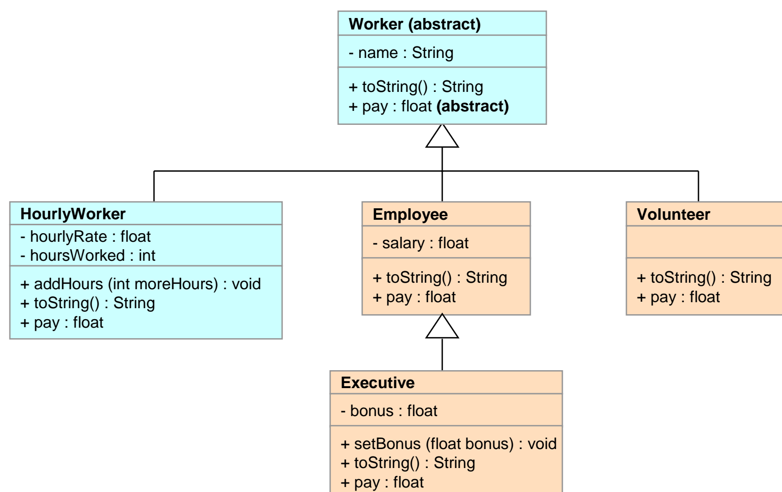
base-class

```
// Represents a generic worker.
abstract public class Worker {

  // Worker's data:
  private String name;
  // Address, telephone, email, etc.,
  // omitted to save clutter.

  // Constructs a worker
  public Worker (String name) {
    this.name = name;
  }

  public String toString() {
    return "Name: " + name;
  }

  // Pays this worker.
  public abstract float pay();
}
```

sub-class

```
//  Represents a volunteer worker.
public class Volunteer extends Worker {

  // Constructs a new volunteer.
  public Volunteer (String name) {
    super (name);
  }

  // Returns information about this volunteer.
   public String toString() {
      String result = super.toString();
      result += "\n" + "Volunteer, no payment";
      return result;
   }

  // Volunteers receive no payment.
  public float pay() {
    return 0;
  }
}
```


Payroll class diagram



Sub-classing Worker: HourlyWorker

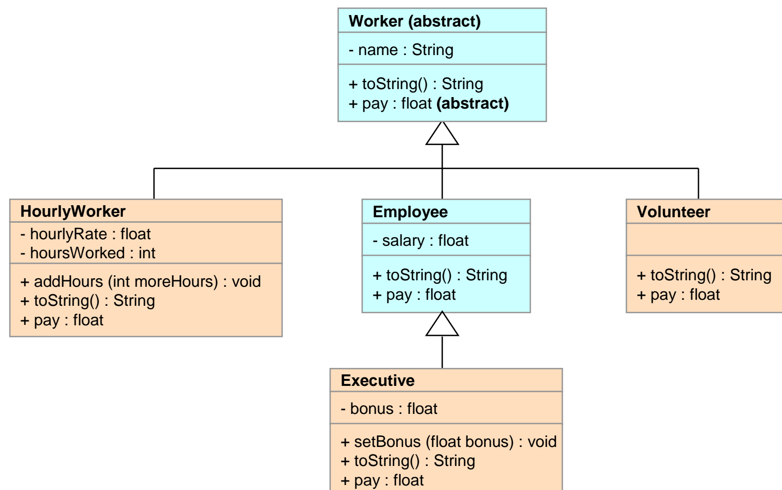
base-class

```
// Represents a generic worker.
abstract public class Worker {

  // Worker's data:
  private String name;
  // Address, telephone, email, etc.,
  // omitted to save clutter.

  // Constructs a worker
  public Worker (String name) {
    this.name = name;
  }

  public String toString() {
    return "Name: " + name;
  }

  // Pays this worker.
  public abstract float pay();
}
```

sub-class

```
public class HourlyWorker extends Worker {
   private float hourlyRate;
   private int hoursWorked;

   // Constructs a new hourly worker
   public HourlyWorker (String name,float hourlyRate) {
     super (name);
     this.hourlyRate = hourlyRate;
     this.hoursWorked = 0;
   }

   public void addHours (int hours) {
      hoursWorked += hours;
   }

   public String toString() {
      String result = super.toString();
      result += "\n" + "Current hours: " + hoursWorked;
      result += "\n" + "Hourly rate: " + hourlyRate;
      return result;
   }

   public float pay() {
     float payment = hoursWorked * hourlyRate;
     hoursWorked = 0;
     return payment;
   }
}
```


Payroll class diagram



Sub-classing Worker: Employee

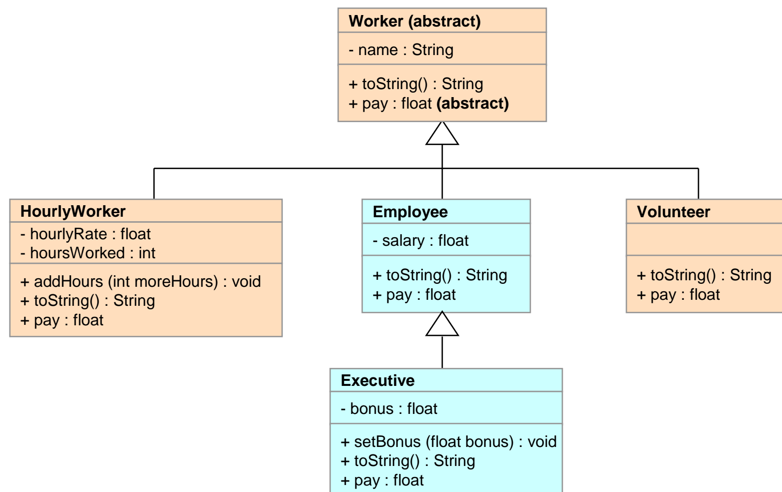
base-class

```
// Represents a generic worker.
abstract public class Worker {

  // Worker's data:
  private String name;
  // Address, telephone, email, etc.,
  // omitted to save clutter.

  // Constructs a worker
  public Worker (String name) {
    this.name = name;
  }

  public String toString() {
    return "Name: " + name;
  }

  // Pays this worker.
  public abstract float pay();
}
```

sub-class

```
//  Represents an employee worker.
public class Employee extends Worker {

  private float salary;

  // Constructs an employee
  public Employee (String name, float salary) {
    super (name);
    this.salary = salary;
  }

  // Returns information about this employee.
  public String toString() {
    String result = super.toString();
    result += "\n"+ "Monthly salary: " + salary;
    return result;
  }

  //  Monthly payment of this employee.
  public float pay() {
    return salary;
  }
}
```


Payroll class diagram



Sub-classing Employee: Executive

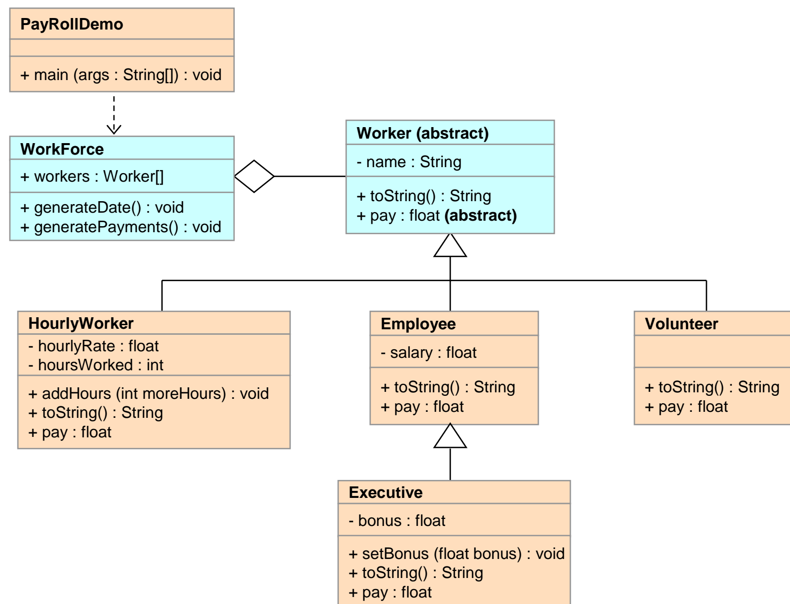
base-class

```java
public class Employee
      extends Worker {

  private float salary;

  // Constructs an employee
  public Employee (String name,
                   float salary) {
    super (name);
    this.salary = salary;
  }

  public String toString() {
    String result = super.toString();
    result += "\n"+ "Monthly salary: "
              + salary;
    return result;
  }

  // Monthly payment of this employee.
  public float pay() {
    return salary;
  }
}
```

sub-class

```java
public class Executive extends Employee {
   private float bonus;

   //  Constructs a new Executive.
   public Executive (String name, float salary) {
      super(name, salary);
      bonus = 0;
   }

   // Awards a bonus to this executive.
   public void setBonus (float bonus) {
       this.bonus = bonus;
   }

  public String toString() {
      String result = super.toString();
      result += "\n" + "Bonus: " + bonus;
      return result;
   }

   // Monthly payment of this executive
   public float pay() {
      float payment = super.pay() + bonus;
      bonus = 0;
      return payment;
   }
}
```


Payroll class diagram (complete)



WorkForce: a collection of Worker objects

```
//  Represents workers and their payments
public class WorkForce {

  private Worker[] workers;

  public WorkForce () {
   //  Constructs a demo array of 6 workers.
    workers = new Worker[6];
    workers[0] = new Executive("Jane", 7500);
    workers[1] = new Employee ("Carla", 3000);
    workers[2] = new Employee ("Woody", 2500);
    workers[3] = new HourlyWorker ("Diane", 10);
    workers[4] = new Volunteer ("Norm");
    workers[5] = new Volunteer ("Cliff");
  }

  // Generate some demo work data
  public void generateData () {
    ((Executive)workers[0]).setBonus(500);
    ((HourlyWorker)workers[3]).addHours(40);
    ((HourlyWorker)workers[3]).addHours(10);
  }

  //  Pays all the workers
  public void generatePayments() // Next slide.
}
```

The construction of different workers depends on their types: different sub-types have different constructors.

Paying the workers, polymorphically

```
public class PayRollDemo {
  public static void main (String[] args) {
    WorkForce workForce = new WorkForce();
    workForce.generateData();
    workForce.generatePayments();
  }
}
```

```
// Code continues from previous slide

//  Pays all the workers
public void generatePayments() {
  for (int j=0 ; j<workers.length; j++) {
    // Print the worker's data
    System.out.println (workers[j]);
    // Compute and print the monthly payment
    System.out.println ("Pay due: " + workers[j].pay());
    System.out.println ();
  }
}}
```

Polymorphic method invocation

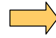
```
C:\WINDOWS\system32\cmd.exe

D:\demo\payroll>java PayRollDemo
Name: Jane
Monthly salary: 7500.0
Bonus: 500.0
Pay due: 8000.0

Name: Carla
Monthly salary: 3000.0
Pay due: 3000.0

Name: Woody
Monthly salary: 2500.0
Pay due: 2500.0

Name: Diane
Current hours: 50
Hourly rate: 10.0
Pay due: 500.0

Name: Norm
Volunteer, no payment
Pay due: 0.0

Name: Cliff
Volunteer, no payment
Pay due: 0.0
```

Outline

- Polymorphism

Examples:

- Fighting army
- Payroll
- Paintbrush



Related topics:

- Polymorphic method calls
- Abstract class vs. interface
- Heterogeneous collections
- Generic sorters

Heterogeneous collections

A heterogeneous collection is a class that can contain objects of arbitrary types

Popular heterogeneous collection classes in Java:

- `java.util.Vector`
- `Java.util.ArrayList`

array

```
Animal[] zoo = new Animal[4];
zoo[0] = new Pig();
zoo[1] = new Pig();
zoo[2] = new Mouse();
zoo[3] = new Dog();
```

Vector

- A `Vector` holds an ordered collection of objects (of any type)
- A growable, flexible, and untyped version of an array
- You can add / remove objects using an index, or not
- The size of the Vector grows and shrinks as needed.

Vector

```
Vector zoo = new Vector();
zoo.addElement(new Pig());
zoo.addElement(new Pig());
zoo.addElement(new Mouse());
zoo.addElement(new Dog());
zoo.addElement(17);
zoo.addElement("It's raining");

zoo.remove(2);
zoo.insertElementAt(new Dog(),2)
```

Heterogeneous collections are not type safe

```
Vector zoo = new Vector();
zoo.addElement(new Pig());
zoo.addElement(new Pig());
zoo.addElement(new Mouse());
zoo.addElement(new Dog());
zoo.addElement(17);
zoo.addElement("It's raining");


Object obj = zoo.elementAt(j);
if obj instanceOf Animal
  Animal a = (Animal) obj;
// Now a can be used as an animal
```

`Vector` (like other heterogeneous collection classes) is type unsafe

Before using an item taken from a `Vector`, you must check its type and then cast accordingly.

Typed collections

Java allows to create typed collections, using the syntax

CollectionName < TypeName >

```
Vector<Animal> zoo = new Vector();
zoo.addElement(new Pig());
zoo.addElement(new Pig());
zoo.addElement(new Mouse());
zoo.addElement(new Dog());
zoo.addElement(17);                // Will not compile
zoo.addElement("It's raining");    // Will not compile

// With a typed collection, there is no need to check and cast:
Animal a = zoo.elementAt(j);
```

Outline

- Polymorphism

Examples:

- Fighting army
- Payroll
- Paintbrush

Related topics:

- Polymorphic method calls
- Abstract class vs. interface
- Heterogeneous collections
- Generic sorters

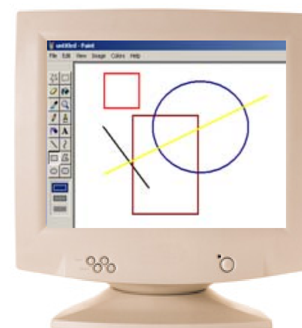
Paintbrush application

The task: Build a program that allows users to create and manage simple pictures. Each picture is made of generic geometrical figures like rectangle, circle, triangle, etc.
The system should allow:

- Creating a new picture
- Adding geometric shapes to the picture
- Deleting figures
- Moving figures
- Resizing figures
- Etc.

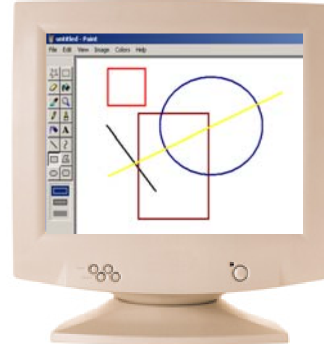
It should be possible to take the picture and

- Store it in a file
- Ship it to another computer
- Display it on any given screen.



Picture API

```
public class Picture {
  public Picture()
  public void addFigure(Figure figure)
  public void deleteFigure(Figure figure)
  public void draw()   // the entire picture
  public void erase()  // the entire picture
  // Other Picture-level methods.
}
```



The PaintBrush application GUI:

We assume that, one way or another, the user defines shapes

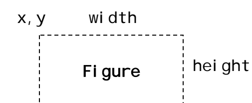
When the user is done defining each shape, we add this shape to the this picture.

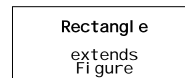
Figure API

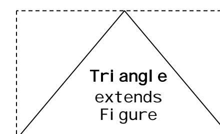
```
public abstract class Figure {
  public Figure(int x, int y, int width, int height)
  public Color getLineColor ()
  public void setLineColor (Color c)
  public abstract void draw ()
   // More Figure methods
}

public class Triangle extends Figure {
  // Constructs an equilateral triangle of a given size
  public Triangle (int x, int y, float size)
  public void draw()

public class Circle extends Figure {
  // Constructs a circle of a given radius
  public Circle(int x, int y, float radius)
  public void draw()

// One such class for every generic figure.
```

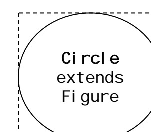
Figure is an abstract class:

It provides a template for deriving sub-classes that represent boxed geometric shapes







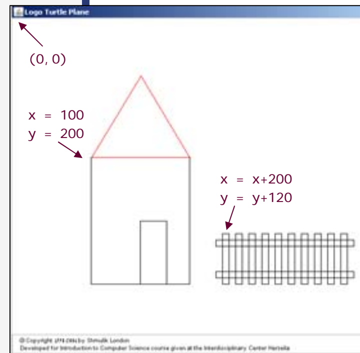


Client code example

```
public class PaintBrushDemo {

  public static void main(String[] args) {
    Picture picture = new Picture();

    // Build a stick house
    int x = 100; int y = 200;
    Rectangle wall = new Rectangle(x, y, 150, 200);
    Rectangle door = new Rectangle(x+75, y+100, 40, 100);
    Triangle roof = new Triangle(x, y, 150);
    roof.setLineColor(Color.red);

    picture.addFigure(wall);
    picture.addFigure(door);
    picture.addFigure(roof);

    // Build a fence
    x = x + 200; y = y + 120;
    for (int i=0; i<10; i++) {
      picture.addFigure(new Rectangle(x, y, 10, 80));
      x += 20;
    }
    // Draw the 2 horizontal beams (omitted)

    picture.draw();
  }
}
```

The client works with intuitive PaintBrush abstractions:

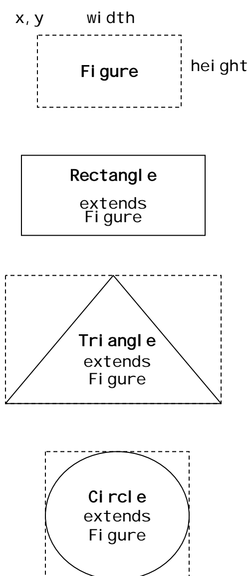
A Picture, to which he adds figures like Rectangle, Triangle, etc.



Behind the scene: The abstract Figure class

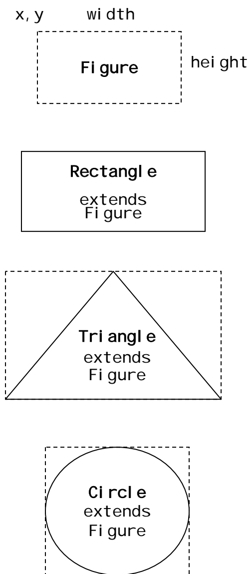
```
import java.awt.*;

public abstract class Figure {

  // The top-left corner of this figure's box
  protected int x,y;

  // The dimensions of this figure's box
  protected int width, height;

  // Default color of this figure's outline
  private Color lineColor = Color.black;

  public Figure(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
  }

  public Color getLineColor() {return lineColor;}

  public void setLineColor(Color c) {lineColor = c;}

  // More Figure methods

}
```

An abstract class, Provides a template for deriving sub-classes that represent boxed geometrical shapes



The abstract Figure class (cont.)

```
public abstract class Figure {

  protected int x, y, width, height;

  // Continued from previous slide ...

  // Draws this figure
  public abstract void draw();

  // Checks if (x,y) is in this figure's box
  public boolean contains(int x, int y) {
    return x >= this.x && x <= this.x + width &&
    y >= this.y && y <= this.y + height;
  }

  // More Figure methods

}
```



- `Figure` is the abstract parent class of all the figure sub-types
- It defines the data and behavior that each boxed geometric figure must have.

Sub-classing Figure: Rectangle

base

```
public abstract class Figure {

  protected int x, y, width, height;
  private Color lineColor = Color.black;

  public Figure(int x, int y,
                int width, int height)

  public Color getLineColor()

  public void setLineColor(Color c)

  public abstract void draw();

  public boolean contains(int x, int y)

  // Other Figure methods

}
```

sub

```
import turtle.Turtle;

public class Rectangle extends Figure {

  public Rectangle(int x, int y,
                   int width, int height) {
    super(x, y, width, height);
  }

  public void draw() {
    Turtle painter = new Turtle();
    painter.setLineColor(getLineColor());
    painter.setLocation(x, y);
    painter.setAngle(0);
    painter.tailDown();
    painter.moveForwards(width);
    painter.turnRight(90);
    painter.moveForwards(height);
    painter.turnRight(90);
    painter.moveForwards(width);
    painter.turnRight(90);
    painter.moveForwards(height);
    painter.hide();
  }
}
```

Sub-classing Figure: Triangle

base

```
public abstract class Figure {

  protected int x, y, width, height;
  private Color lineColor = Color.black;

  public Figure(int x, int y,
                int width, int height)

  public Color getLineColor()

  public void setLineColor(Color c)

  public abstract void draw();

  public boolean contains(int x, int y)

  // Other Figure methods

}
```

sub

```
import turtle.Turtle;

public class Triangle extends Figure {

  // Constructs an equilateral triangle.
  // x,y are the top-left corner of the box
  public Triangle(int x, int y, int size) {
    super(x,y,size,(int)Math.sqrt(3/4)*size);
  }

  public void draw() {
    Turtle painter = new Turtle();
    painter.setLineColor(getLineColor());
    painter.setLocation(x + width,
                        y + height);
    painter.tailDown();
    painter.turnLeft(30);
    painter.moveForwards(width);
    painter.turnLeft(120);
    painter.moveForwards(width);
    painter.turnLeft(120);
    painter.moveForwards(width);
    painter.hide();
  }
}
```


The Picture class

```
import java.util.Vector;

public class Picture {

  private Vector figures;

  public Picture() {
    this.figures = new Vector();
  }

  public void addFigure(Figure figure) {
    figures.addElement(figure);
  }

  public void draw() {
    for (int i = 0; i < figures.size(); i++) {
        ((Figure) figures.elementAt(i)).draw();
    }
  }
}
```

Polymorphic method invocation

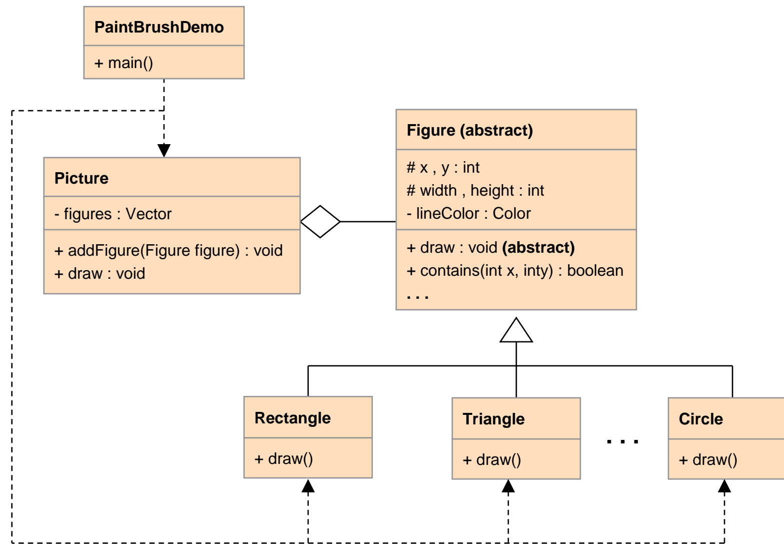
PaintBrush clients create pictures by:

- Constructing a picture
- Constructing various figures
- Adding the figures to the picture

Thus, it makes sense to implement picture as a heterogeneous collection.

Note the casting – we are dealing with a type unsafe collection (Vector)

The PaintBrush application: class diagram



Outline

- Polymorphism

Examples:

- Fighting army
- Payroll
- Paintbrush

Related topics:

- Polymorphic method calls
- Abstract class vs. interface
- Heterogeneous collections
- Generic sorters

Revisiting interfaces: `Comparable`

implements

```
public class Date implements Comparable {
  private int year, month, day;

  public Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }

  public String toString() {
    return day + "/" + month + "/" + year;
  }

  public boolean gt(Comparable other) {
    Date d = (Date) other;
    if (year != d.year) return year > d.year;
    if (month != d.month) return month > d.month;
    return day > d.day;
  }

 // lt and equals implementation are similar

}
```

interface

```
public interface Comparable {
  boolean gt (Comparable other);
  boolean lt (Comparable other);
  boolean equals (Comparable other);
}
```

The Java class library features a `comparable` interface.

In this example we create a `comparable` interface of our own.

Generic sorter

```
public class Sorter {
  public static void selectionSort (Comparable[] a) {
    for (int j=0 ; j<a.length-1 ; j++) {
      int min = j;
      for (int k=j+1 ;  k<a.length ; k++) {
        if ( a[min].gt(a[k]) )
          min = k;
        if (min != j) {
          Comparable temp = a[min];
          a[min] = a[j];
          a[j] = temp;
        }
      }
    }
  }
}
```

Instead of sorting an array of a specific data type, we are willing to sort any array of `Comparable` objects

Instead of using > we use `gt`, since we know that `Comparable` objects must implement it.

When declaring a new variable that holds a comparable value, we cast it as `Comparable`

Implication: `Sorter.selectionSort` can now be used to sort objects that come from any class that implements `Comparable`

Generic sorter

```
public class DateSortDemo{
  public static void main (String[] args) {
    Date[] dates = new Date[4];
    dates[0] = new Date(17, 2, 2009);
    dates[1] = new Date(1, 4, 1954);
    dates[2] = new Date(20, 2, 2009);
    dates[3] = new Date(3, 11, 1967);
    Sorter.selectionSort(dates);
    for ( int j =0 ; j < 4 ; j++ )
      System.out.println(dates[j]);
  }
}
```