Learning the Java Language

2.3 Objects and Classes in Java
• Importing Classes and Packages
• Creating and Extending Classes
• Implementing Interfaces
• Declaring Variables
• Implementing Methods
• The Applet Class
• How it works?
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Spot extends Applet implements MouseListener{
    private java.awt.Point clickPoint = null;
    private static final int RADIUS = 7;

    public void init() {
        addMouseListener(this);
    }
    public void paint(Graphics g) {
        g.drawRect(0, 0, getSize().width - 1, 
            getSize().height - 1);
        if (clickPoint != null)
            g.fillOval(clickPoint.x - RADIUS, 
                clickPoint.y - RADIUS, 
                RADIUS * 2, RADIUS * 2);
    }
    public void mousePressed(MouseEvent event) {
        clickPoint = event.getPoint();
        repaint();
    }
    public void mouseClicked(MouseEvent event) {} 
    public void mouseReleased(MouseEvent event) {} 
    public void mouseEntered(MouseEvent event) {} 
    public void mouseExited(MouseEvent event) {} 
}
A. Importing Classes and Packages

In Java, every class is in a package
- similar to file in a directory

There may be packages inside a package
- similar to a directory can have subdirectories
• It is common that classes need other classes to help their operation. E.g. `System.out.println(...)`
• For those classes or packages that are often used, we `import` them before the program run
• It makes the program easier to read and write

They are the same

```java
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends java.applet.Applet {
    public void paint(java.awt.Graphics gph) {
        gph.drawString(“Hello world!”, 50, 25);
    }
}
```
B. Creating and Extending Classes

It declares a new class **Spot** which is publicly accessible and is a subclass of **Applet** and agree to implement the interface **MouseListener**.

```java
public class Spot extends Applet implements MouseListener {
    private java.awt.Point clickPoint = null;
    private static final int RADIUS = 7;
}
```
In fact, a class can be created with a number of options

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Class is publicly accessible</td>
<td>non-public</td>
</tr>
<tr>
<td>abstract</td>
<td>Class cannot be instantiated</td>
<td>non-abstract</td>
</tr>
<tr>
<td>final</td>
<td>Class cannot be subclassed</td>
<td>non-final</td>
</tr>
<tr>
<td>class <strong>NameOfClass</strong></td>
<td><strong>Name of the Class</strong></td>
<td></td>
</tr>
<tr>
<td>extends <strong>Super</strong></td>
<td>Superclass of the class</td>
<td>No superclass</td>
</tr>
<tr>
<td>implement <strong>interface</strong></td>
<td>Interfaces implemented by the class</td>
<td>Implement no interface</td>
</tr>
</tbody>
</table>

{  
  **ClassBody**  
}

- Only `class **NameOfClass**` is compulsory, others are optional
- If not specified, default is assumed
• public
  – By default, a class can be used only by other classes in the same package
  – If declared, it means the class is accessible by classes of any package

• abstract
  – Some classes are just abstract concepts that cannot instantiate an object from them
  – E.g. the class Food is an abstract class. It refers to something that we can eat
  – We don’t have objects of class Food. But we have objects of class Carrot, Chicken, Ham, etc.

• final
  – If declare, the current class will not have subclass
  – Usually use for security reasons, e.g. avoid hacker subvert system
• extends `super`
  - If declared, suggested that `super` is the superclass of the current class

    ```
    public class Spot extends Applet implements MouseListener {
        private java.awt.Point clickPoint = null;
        private static final int RADIUS = 7;
    }
    ```

• **Applet** is a very useful class for writing applet
• It handles things required to put the applet onto the browser
• Inherit **Applet** allows the subclass to have the same ability
• User can further improve the ability by overriding some of the inherited methods
Remark 1: Inheritance and Overriding

- Subclass **inherits** superclass variables or method if they are declared as public or protected (but not private)
- The subclass maintains the same capability as its superclass
- **Simplify** the creation of new classes

```java
class Parent {
    public int aNum = 2;
    public int increment (int bNum) {
        return (bNum + 1);
    }
}

class Child extends Parent {
    int res = aNum;
    res = increment(res);
}
```

*aNum and increment() used in Child are the members of Parent*

*res = 3 at the end*
• However, if the subclass further declares the superclass’s variable or method, this variable or method is said to have been overridden

• Overriding allows the subclass to enhance its capability to be even better than its superclass

• Overridden variables and method can still be accessible through, e.g. Super.aNum, which refers to aNum in Parent class

```java
import java.io.*;

class Parent {
    public int aNum = 2;
    public int increment(int bNum) {
        return (bNum + 1);
    }
}

public class Child extends Parent {
    int aNum;
    public Child() {
        this.aNum = increment(super.aNum);
    }

    public static void main(String[] args) {
        Child t = new Child();
        System.out.println("Result = "+t.aNum);
    }
}
```
• Remark 2: Constructor
  – In the previous program, the constructor of Child is defined
  – Usually the constructor of a class performs initialization when the class is instantiated, e.g. by the new method
  – The constructor of a class is identified by its name. It has the same name as the Class name
  – The constructor of the class Spot is hidden. It is provided by the Java runtime which, in fact, doesn’t do anything

• Remark 3: Keywords this and super
  – The keyword this refers to the current object
  – The keyword super refers to the parent class
Exercise: Build the above program using Java Workshop 2.0. Show the result.

Modify the statement

```java
this.aNum = increment( super.aNum);
```

to

```java
Child.aNum = increment( Parent.aNum);
```

Are they the same? Why?
C. Implementing Interfaces

Example 1: ATM

- When the bank installs the ATM, it has no idea who the customer is and how the customer uses the machine.
- To standardize the usage, the ATM built with a standard interface that every user, if he wants to use the ATM, must follow.
- They must implement the procedures that the ATM requires them to do, e.g. key in the withdraw amount.
Example: Implementing Interfaces for Mouse Event Generator

- When Java designed the *Mouse Event Generator*, it did not know who need the mouse events.
- It gives the definitions of an interface and claims that it will send the events to that interface.
- Any program that needs the mouse events should implement the interface by implementing some methods.
• An **interface** is a named collection of method definitions (without implementation) or constant variables
• To implement an interface **gives an access point** for other objects to your object
• To implement an interface means that the object has agreed with a **protocol**, e.g. agree to **implement some methods**
• Interface is **useful for inter-objects access** since it is not restricted due to the class or package boundary
public class Spot extends Applet implements MouseListener {
    private java.awt.Point clickPoint = null;
    private static final int RADIUS = 7;
    
    addMouseListener(this);
    
    public void mousePressed(MouseEvent event) {
        clickPoint = event.getPoint();
        repaint();
    }
    public void mouseClicked(MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
    public void mouseEntered(MouseEvent event) { }
    public void mouseExited(MouseEvent event) { }
}
Exercise: Build the class Spot using Java Workshop 2.0. Show the result.

Remove the statement

```java
public void mouseExited(MouseEvent event) { }
```

What is the error message? Why?
D. Declaring Variables

```java
public class Spot extends Applet implements MouseListener {
    private java.awt.Point clickPoint = null;
    private static final int RADIUS = 7;
    public void init() {} 
    public void paint(Graphics g) {} 
    public void mousePressed(MouseEvent event) {} 
    public void mouseClicked(MouseEvent event) {} 
    public void mouseReleased(MouseEvent event) {} 
    public void mouseEntered(MouseEvent event) {} 
    public void mouseExited(MouseEvent event) {} 
}
```

Declare the variable members
private static final int RADIUS = 7;

Declare the access level

**Private:** Only the current class can access variable. Just like a personal secret. You even don’t want your parents and your friends know it

**Protected:** The class itself, its subclass and classes in the same package can access the variable. Just like a family secret

**Public:** Any class of any parentage in any package has access to this variable. No secret at all

**Package:** This level of access assumes that classes in the same package are trusted friends. They can have the access to this variable. It is like that you extend to your closest friends but wouldn’t trust even to your family
private static final int RADIUS = 7;

Indicates that it is constant

- It’s a compile-time error if your program ever tries to change a final variable
- By convention, the name of constant values are spelled in uppercase letters
- It is a good practice to declare the constant variable to be final. It saves memory for its implementation

Type and Name

- It’s a must for all variables

Initialization

- To initialize the variable
private static final int RADIUS = 7;

They are different copies of x

myX.x = 1
yourX.x = 2

class Xclass {
    int x;  // instance variable
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}

Program

Xclass myX = new Xclass();
myX.x = 1;
Xclass yourX = new Xclass();
yourX.x = 2;
System.out.println("myX.x = " + myX.x());
System.out.println("yourX.x = " + yourX.x());
class Xclass {
    static int x;
    // class variable
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}

Xclass myX = new Xclass();
myX.x = 1;
Xclass yourX = new Xclass();
yourX.x = 2;
System.out.println("myX.x = " + myX.x());
System.out.println("yourX.x = " + yourX.x());

They refer to the shared copy of x

• **Instance variable**: a new copy of variable is generated for each instance of the class
• **Class variable**: only one copy of the variable shared by various instances of the class

**Result**

myX.x = 2
yourX.x = 2
• Similarly, method can also be divided into class method or instance method using the keyword `static`.

```java
class Xclass {
    static int x;
        // class variable
    static public int x() {
        return x;
    } // class method
    static public void setX(int newX) {
        x = newX;
    } // class method
}
```

• **Class method** implies that all instances of the class share a copy of this method.

• **Instance method** implies that every instance has its own copy of this method.

• **Class method can only access class variables**.

• **Class methods and variables** give saving in memory

• **Instance methods and variables** give good integrity of different instances of a class.
Exercise: Modify the following statement of the class Spot

```java
public void int () {
```
to

```java
public static void int () {
```

What is the error message? Why?
E. Implementing Methods

```java
public class Spot extends Applet implements MouseListener {
    public void paint(Graphic g) {
        // Method body
    }
}
```
• There are many other options:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessLevel</td>
<td>Access level for this method</td>
</tr>
<tr>
<td>static</td>
<td>This is a class method</td>
</tr>
<tr>
<td>abstract</td>
<td>This method is not implemented</td>
</tr>
<tr>
<td>final</td>
<td>Method cannot be overridden</td>
</tr>
<tr>
<td>native</td>
<td>Method implemented in another language</td>
</tr>
<tr>
<td>synchronized</td>
<td>Method requires a monitor to run</td>
</tr>
<tr>
<td><strong>returnType</strong></td>
<td>The return type and method name</td>
</tr>
<tr>
<td><strong>methodName</strong></td>
<td></td>
</tr>
<tr>
<td>(paramlist)</td>
<td>The list of argument</td>
</tr>
<tr>
<td>throws exceptions</td>
<td>The exceptions thrown by this method</td>
</tr>
</tbody>
</table>
Two approaches for passing parameters to a method:

- **Pass-by-value**
- **Pass-by-reference**

Sometimes cause confusion

```java
public class Pen {
    int redValue, greenValue, blueValue;
    public void getRGBColor(int red, int green, int blue) {
        red = redValue;
        green = greenValue;
        blue = blueValue;
    }
}
```

```java
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red= "+r+", green = "+g+", blue= "+b);
```

We expect that the system will print the updated values of r, g, b in Pen class

The result is that the values of r, g, and b are still -1
The following program will follow our expectation:

```java
//define another class
public class RGBColor {
    public int r, g, b;
}

public class Pen {
    int redValue, greenValue, blueValue;
    public void getRGBColor(RGBColor aColor) {
        aColor.r = redValue;
        aColor.g = greenValue;
        aColor.b = blueValue;
    }
}
```

The system will print the updated values of r, g, b in Pen class.

```java
RGBColor penColor = new RGBColor();
pen.getRGBColor(penColor);
System.out.println("red= " + penColor.r +
    "green = " + penColor.g +
    "blue= " + penColor.b);
```
F. The Applet Class

• Every applet is implemented by creating a subclass of the Applet class

• The Applet class inherits from the Abstract Windows Toolkit (AWT) class

• That’s why one can create user interfaces in applets

```
java.lang.Object
  \downarrow
java.awt.Component
  \downarrow
java.awt.Container
  \downarrow
java.applet.Applet
```
The Life Cycle of an Applet

- **Initialize** - when the applet is created; once per applet
- **Start** - after the applet is initialized or revisited
- **Stop** - when user goes to other page, a started applet must stop
- **Final Cleanup** - when the applet is destroyed; once per applet

An applet must contain the following methods to indicate the action the applet will do in each case:

```java
public class Simple extends Applet {
    public void init() { . . . }
    public void start() { . . . }
    public void stop() { . . . }
    public void destroy() { . . . }
}
```
Methods for Drawing

- **Applets** can override two display methods for **drawing** objects
  - **paint** - the basic display method. Draw the applet’s representation within a browser window
  - **update** - a method used with **paint** to improve drawing performance

- **Applets** inherit their paint and update methods from the **Applet** class, which inherits them from the **AWT Component** class
For example:

```java
public class Spot extends Applet implements MouseListener {
    public void init() {
        addMouseListener(this);
    }
    public void paint(Graphic g) {
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
        if (clickPoint != null)
            g.fillOval(clickPoint.x - RADIUS,
                       clickPoint.y - RADIUS,
                       RADIUS * 2, RADIUS * 2);
    }
}
```

When the applet is first created, a Mouse Listener is added.

Whenever the screen is updated, first draw a rectangle.

If the mouse has clicked at (x, y), draw a filled oval at this position with diameter 2*RADIUS.
**Exercise:** Add the `start()` and `stop()` methods to the class `Spot` such that it will make a **click sound** whenever the applet maximizes and minimizes

**Hint:** A click sound can be generated by using the method

```java
Toolkit.getDefaultToolkit().beep()
```

when the class `java.awt.Toolkit` has been imported
G. How it works?

```java
public class Spot extends Applet implements MouseListener {
    private Point clickPoint = null;
    private static final int RADIUS = 7;
    public void init() {
        addMouseListener(this);
    }
    public void paint(Graphic g) {
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
        if (clickPoint != null)
            g.fillOval(clickPoint.x - RADIUS,
                       clickPoint.y - RADIUS,
                       RADIUS * 2, RADIUS * 2);
    }
    public void mousePressed(MouseEvent event) {
        clickPoint = event.getPoint();
        repaint(); // will call paint()
    }
}
```

When the applet is first created, a Mouse Listener is added.

Whenever the screen is updated, first draw a rectangle, and draw an oval at position (x,y) with diameter 2*RADIUS.

When the Mouse is clicked, save the position and update the screen.
• The **Spot** can be executed in a Web browser using the `<APPLET>` tag of the HTML

Using a text editor, create a file named `Spot.html` in the same directory that contains `Spot.class`. This HTML file should contain the following text:

```html
<HTML>
<HEAD>
<TITLE> The Spot Example </TITLE>
</HEAD>
<BODY>

Click on the area below and you will see the spot!!!

<APPLET CODE="Spot.class" WIDTH=500 HEIGHT=600>
</APPLET>
</BODY>
</HTML>
```