Overloading

- Each method has a *signature*: its name together with the number and types of its parameters

<table>
<thead>
<tr>
<th>Methods</th>
<th>Signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>String toString()</td>
<td>()</td>
</tr>
<tr>
<td>void move(int dx, int dy)</td>
<td>(int, int)</td>
</tr>
<tr>
<td>void paint(Graphics g)</td>
<td>(Graphics)</td>
</tr>
</tbody>
</table>

- Two methods can have the same name if they have different signatures. They are *overloaded*.

Overloading Example

```java
public class Point {
    protected double x, y;

    public Point() {
        x = 0.0; y = 0.0;
    }

    public Point(double x, double y) {
        this.x = x; this.y = y;
    }

    /**
     * calculate the distance between this point and the other point
     */
    public double distance(Point other) {
        double dx = this.x - other.x;
        double dy = this.y - other.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    Overloading Example (cont'd)

    /**
     * calculate the distance between this point and (x,y)
     */
    public double distance(double x, double y) {
        double dx = this.x - x;
        double dy = this.y - y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    /**
     * calculate the distance between this point and the origin
     */
    public double distance() {
        return Math.sqrt(x * x + y * y);
    }

    // other methods
}
```

When to Overload

When there is a general, non-discriminative description of the functionality that can fit all the overloaded methods.

```java
public class StringBuffer {
    String append(String str) { ... }
    String append(boolean b) { ... }
    String append(char c) { ... }
    String append(int i) { ... }
    String append(long l) { ... }
    String append(float f) { ... }
    String append(double d) { ... }
    // ...
}
```
When to Overload (cont'd)

When all the overloaded methods offer exactly the same functionality, and some of them provide default values for some of the parameters.

```java
public class String {
    public String substring(int i, int j) {
        // base method: return substring [i .. j-1]
    }
    public String substring(int i) {
        // provide default argument
        return substring(i, length - 1);
    }
    // ...
}
```

Inheritance and Extended Classes

- *Extended classes* are also known as *subclasses*.
- Inheritance models the *is-a* relationship.
- If class E is an extended class of class B, then any object of E can *act-as* an object of B.
- Only single inheritance is allowed among classes.
- All public and protected members of a super class are accessible in the extended classes.
- All protected members are also accessible within the package.

Constructors of Extended Classes

- The constructor of the super class can be invoked.
- `super(...)` must be the first statement.
- If the super constructor is not invoked explicitly, by default the no-arg `super()` is invoked implicitly.
- You can also invoke another constructor of the same class.

```java
public class ColoredPoint extends Point {
    public Color color;
    public ColoredPoint(double x, double y, Color color) {
        super(x, y);
        this.color = color;
    }
    public ColoredPoint(double x, double y) {
        this(x, y, Color.black); // default value of color
    }
    public ColoredPoint() {
        color = Color.black;
    }
}
```
Default Constructor of Extended Classes

Default no-arg constructor is provided:

```java
public class Extended extends Super {
    public Extended() {
        super();
    }
    // methods and fields
}
```

Execution Order of Constructors

```java
public class Super {
    int x = ...;  // executed first

    public Super() {
        x = ...;    // executed second
    }
    // ...
}

public class Extended extends Super {
    int y = ...;  // executed third

    public Extended() {
        super();
        y = ...;   // executed fourth
    }
    // ...
}
```

Overriding and Hiding

Overloading

More than one methods have the same name but different signatures

Overriding

Replacing the implementation of a methods in the superclass with one of your own.
- You can only override a method with the same signature.
- You can only override non-static methods.

Hiding

Fields and static methods can not be overridden. They can only be hidden.
- Hidden fields and static methods are still accessible via references to the superclass.
- A static method can be only be hidden by another static method.
- A static variable may be hidden by an instance variable.

Overriding and Hiding (cont'd)

Which implementation is used?

- When invoking a non-static method, the actual class of the object determines. (run-time)
- When accessing a field, the declared type determines. (compile time)
- When invoking a static method, the declared type determines. (compile time)
Object Reference: super

Keyword super is an reference to the current object but acts as an instance of its superclass.

Consider the equals() method in ColoredPoint

```java
public boolean equals(Object other) {
    if (other != null && other instanceof ColorPoint) {
        ColorPoint p = (ColorPoint) other;
        return (super.equals(p) && p.color.equals(this.color));
    } else {
        return false;
    }
}
```

Type Conversion --- Implicit

Java allows two kinds of implicit type conversions:

Numeric variables
Any numeric types can be converted to another numeric type with larger range, e.g. `char ==> int`, `int ==> long`, `int ==> float`, `float ==> double`.

Object reference
An object reference of class C can be converted to a reference of a superclass of C.

Type Conversion --- Explicit Cast

Numeric variables
Any numeric types can be explicitly cast to any other numeric type. May lose bits, precision.

Object reference
Cast an object reference of a class to a reference of any other class is:
- syntactically allowed; but
- runtime checked.

Cast Object References

class Student { ... }
class Undergraduate extends Student { ... }
class Graduate extends Student { ... }

Student student1, student2;
student1 = new Undergraduate(); // ok
student2 = new Graduate(); // ok

Graduate student3;
student3 = student2; // compilation error
student3 = (Graduate) student2; // explicit cast, ok
student3 = (Graduate) student1; // compilation ok // run-time exception
Graphical User Interfaces (GUI)

Abstract Windows Toolkit (AWT): `java.awt`
- **GUI elements:**
  - **Primitive**
    - Button, Label, Checkbox, Scrollbar, etc.
  - **Container**
    - Panel, Frame, Dialog, etc.
- **Layout managers:**
  - FlowLayout, BorderLayout, etc.
- **Supporting classes:**
  - **Event handling**
    - `java.awt.event` package
  - **Graphics**
    - Color, Font, Graphics, etc.
  - **Geometry**
    - Point, Rectangle, Dimension, etc.
  - **Imaging**
    - Image class and `java.awt.image` package

The Component Hierarchy

The Swing Components

Layout Managers

- The layout of the elements in a container is handled by the *layout manager* associated with the container.
- Relative positions of the elements are specified, not their absolute coordinates.
- The positions and sizes of the element will be automatically adjusted when the window is resized.
The Layout Manager Hierarchy

Buttons and Flow Layout

Layout elements in horizontal rows.

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

public class Flow extends Applet {
    public Flow () {
        setLayout(new FlowLayout());
        add(new Button("Java"));
        add(new Button("C++"));
        add(new Button("Perl"));
        add(new Button("Ada"));
        add(new Button("Smalltalk"));
        add(new Button("Eiffel"));
    }
}
```

Border Layout
Border Layout (cont'd)

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

public class Border extends Applet {
    public Border () {
        setLayout(new BorderLayout());
        add(new Button("North"), BorderLayout.NORTH);
        add(new Button("South"), BorderLayout.SOUTH);
        add(new Button("East"), BorderLayout.EAST);
        add(new Button("West"), BorderLayout.WEST);
        add(new Button("Center"), BorderLayout.CENTER);
    }
}
```

Grid Layout

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

public class Grid extends Applet {
    public void init () {
        int row = 0, col = 0;
        String att = getParameter("row");
        if (att != null) row = Integer.parseInt(att);
        att = getParameter("col");
        if (att != null) col = Integer.parseInt(att);
        if (row == 0 && col == 0) { row = 3; col = 2; }
        setLayout(new GridLayout(row, col));
        add(new Button("Java"));
        add(new Button("C++"));
        add(new Button("Perl"));
        add(new Button("Ada"));
        add(new Button("Smalltalk"));
        add(new Button("Eiffel"));
    }
}
```
public class NestedPanels extends Applet {
    protected Label messageBar;
    protected Choice choice;

    public NestedPanels () {
        // set up the center panel
        Panel center = new Panel();
        center.setLayout(new BorderLayout());
        center.add(new Button("south"), BorderLayout.SOUTH);
        center.add(new Button("north"), BorderLayout.NORTH);
        center.add(new Button("east"), BorderLayout.EAST);
        center.add(new Button("west"), BorderLayout.WEST);
        center.add(new Button("center"), BorderLayout.CENTER);

        // set up the south panel
        Panel south = new Panel();
        south.setLayout(new FlowLayout());
        south.add(new Button("Help"));
        choice = new Choice();
        choice.addItem("one");
        choice.addItem("two");
        choice.addItem("three");
        choice.addItem("four");
        choice.addItem("five");
        south.add(choice);
        messageBar = new Label("This is a message bar.");
        south.add(messageBar);

        // set up the outer panel
        setLayout(new BorderLayout());
        add(new Button("North"), BorderLayout.NORTH);
        add(new Button("East"), BorderLayout.EAST);
        add(new Button("West"), BorderLayout.WEST);
        add(south, BorderLayout.SOUTH);
        add(center, BorderLayout.CENTER);
    }
}
**Event Handling**

- Event source: buttons, checkboxes, choices
- Event listener: any class interested in handling certain events
- A listener must
  - implement an appropriate *listener* interface;
  - inform the source that it is interested in handling a certain type of events.
- A listener may listen to several sources and different types of events.
- The source may also be the listener.
- Listeners can be full-fledged classes or *inner* classes.

**The Event Object and Listener Classes**

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Listener Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>ActionListener</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>ItemListener</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>MouseListener, MouseMotionListener, MouseMotionAdapter</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>KeyListener, KeyAdapter</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>WindowListener, WindowAdapter</td>
</tr>
</tbody>
</table>

- *XyzListener* are interfaces.
- *XyzAdapter* are classes that implement the corresponding listener interfaces.

**Nested Panels, Handling Events**

```java
import java.awt.*;
import java.awt.event.*;

public class NestedPanels2 extends NestedPanels implements ActionListener, ItemListener {

    public NestedPanels2() {
        super(); // create all the components
        // register item listener
        choice.addItemListener(this);
        // register action listener
        registerButtonHandler(this);
    }

    public void itemStateChanged(ItemEvent event) {
        if (event.getStateChange() == ItemEvent.SELECTED) {
            messageBar.setText("Choice selected: " + event.getItem());
        }
    }

    public void actionPerformed(ActionEvent event) {
        Button source = (Button) event.getSource();
        messageBar.setText("Button pushed: " + source.getLabel());
    }
}
```

**Event Handling Methods**

```java
public void itemStateChanged(ItemEvent event) {
    if (event.getStateChange() == ItemEvent.SELECTED) {
        messageBar.setText("Choice selected: " + event.getItem());
    }
}

public void actionPerformed(ActionEvent event) {
    Button source = (Button) event.getSource();
    messageBar.setText("Button pushed: " + source.getLabel());
}
```
Register The Listener

```java
protected
void registerButtonHandler(Component comp) {
    if (comp != null) {
        if (comp instanceof Button) {
            Button button = (Button) comp;
            button.addActionListener(this);
        } else if (comp instanceof Container) {
            Container container = (Container) comp;
            int n = container.getComponentCount();
            for (int i = 0; i < n; i++)
                registerButtonHandler(
                    container.getComponent(i));
        }
    }
}
```

Event Handling Using Inner Class

```java
import java.awt.*;
import java.awt.event.*;

public class NestedPanels3 extends NestedPanels {
    public NestedPanels3() {
        super();
        ChoiceEventHandler cHandler =
            new ChoiceEventHandler();
        choice.addItemListener(cHandler);
        ButtonEventHandler bHandler =
            new ButtonEventHandler();
        bHandler.registerButtonHandler(this);
    }
}
```

Inner Class Listener

Inner classes
- classes that reside inside other (full-fledged) classes.
- Intended to be small.
- serve as helpers to the enclosing class.

```java
class ChoiceEventHandler implements ItemListener {
    public void itemStateChanged(ItemEvent event) {
        if (event.getStateChange() ==
            ItemEvent.SELECTED) {
            messageBar.setText("Choice selected: " +
                event.getItem());
        }
    }
}
```

Inner Class Listener (cont'd)

```java
class ButtonEventHandler implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        Button source = (Button) event.getSource();
        messageBar.setText("Button pushed: " +
            source.getLabel());
    }
}
```
protected void
registerButtonHandler(Component comp) {
    if (comp != null) {
        if (comp instanceof Button) {
            Button button = (Button) comp;
            button.addActionListener(this);
        } else if (comp instanceof Container) {
            Container container = (Container) comp;
            int n = container.getComponentCount();
            for (int i = 0; i < n; i++)
                registerButtonHandler(
                    container.getComponent(i));
        }
    }
}