Object-Oriented Java Programming

Why Java?

- Simple
- Platform Independent
- Safe
- Multi-Threaded
- Garbage Collected

Objects, Classes, and Methods

- **Object**
  Something that occupies memory and has a state

- **Class**
  Pattern (type) of object

- **Method**
  Function and procedure to access object state

Constructors Overloading

```java
public class Point {
    ...}

public class Rectangle {
    private int width = 0;
    private int height = 0;
    private Point origin;

    public Rectangle() {
        origin = new Point(0, 0);
    }

    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
    ...
}
```

Access Control

- **Public:**
- **Private:**
- **Protected:**
- **Package:**

Example of Access Control

```java
Base.java
package p1;
public class Base {
    private int u;
    protected int v;
    public int w;
    private void g() {
        System.out.println(v);
    }
    protected void f() {
        System.out.println(u);
        g();
    }
}
```

```java
Test.java
package p2;
import p1.Base;
public class Test {
    public static void main(String[] args) {
        Base b = new Base();
        // Compilation error
        b.v = 2;
        // Compilation error
        b.g();
        // Compilation error
        b.f();
    }
}
```

```java
Derived.java
package p2;
import p1.Base;
public class Derived extends Base {
    void h() {
        f();
    }
    void k() {
        g();
    }
}
```

```java
Derived.java
package p2;
import p1.Base;
public class Derived extends Base {
    void h() {
        System.out.println(b.v);
    }
    void k() {
        System.out.println(b.w);
    }
    ...
}
```
Inheritance

```java
public class B extends A {
    ...
}
```

```java
public class A {
    ...
}
```

<table>
<thead>
<tr>
<th>Class Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Object</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Method equals()

- **==**: identical references to objects
  - String s = "Hello";
  - String t = new String("Hello");
  - if (s == t) System.out.println("the same");
  - // condition is false and print nothing!
- **equals**: equality of objects
  - Already implemented in class Object
  - Need to define for your class:
    - Point p1 = new Point(10, 10);
    - Point p2 = new Point(5, 5);
    - Rectangle rec1 = new Rectangle(p1, 5, 10);
    - Rectangle rec2 = new Rectangle(p2, 5, 10);
    - if (rec1 == rec2) System.out.println("the same");
    - // condition is false and print nothing!
    - if (rec1.equals(rec2)) System.out.println("the same");

```java
public class Rectangle {
    ...
    public boolean equals (Object obj) {
        Rectangle rec = (Rectangle) obj;
        return (w == rec.width()) && (h == rec.height());
    }
}
```

Override equals()

```java
public class Rectangle {
    ...
    public boolean equals (Object obj) {
        Rectangle rec = (Rectangle) obj;
        return (w == rec.width()) && (h == rec.height());
    }
}
```

Method toString()

- Rectangle rec = new Rectangle(p1, 5, 10);
- System.out.println (rec);
  - // prints something like Rectangle@736d6at

```java
public class Rectangle {
    ...
    public String toString() {
        return "width = " + w + ", height = " + h;
    }
}
```

Override toString()

```java
public class Rectangle {
    ...
    public String toString() {
        return "width = " + w + " , height = " + h;
    }
}
```

Output:

- Rectangle rec = new Rectangle(p1, 5, 10);
- System.out.println(rec);
  - Output: width = 5, height = 10
Inheritance

public class Circle
{
    private Point center;
    private double radius;

    public Circle( Point c, double r)
    { center = c;
        radius = r;
    }

    public double area()
    { return Math.PI * radius * radius; }
}

Inheritance (Cont’d)

public class Rectangle
{
    private Point center;
    private double width;
    private double height;

    public Rectangle( Point c, double w, double h)
    { center = c;
        width = w;
        height = h;
    }

    public double area()
    { return width * height; }
}

public abstract class Shape
{
    private Point center;

    public Shape( Point c)
    { center = c; }

    public abstract double area();
}

Inheritance (Cont’d)

public class Circle extends Shape
{
    public Circle( Point c, double r)
    {
        super(c);
        radius = r;
    }

    public double area()
    { return Math.PI * radius * radius; }
}

public class Rectangle extends Shape
{
    private double width;
    private double height;

    public Rectangle( Point c, double w, double h)
    { super(c);
        width = w;
        height = h;
    }

    public double area()
    { return width * height; }
}

Casting

• Upcasting:
  Assign a subclass reference to a superclass variable

• Downcasting:
  Assign Superclass Reference to subclass variable

Example of Casting

public class Queue
{ // constructors and instance variables omitted
    public void enqueue( int x) { … }
    public void dequeue() { … }
    public int front() { … }
    public boolean empty() { … }
}

Queue q1, q2;
Deque d;

d = new Deque();
q1 = d;
d.q1.deleteRear();
// Compilation error
d = q1;
// Compilation error
d = (Deque) q1;
d.deleteRear();
q2 = new Queue();
d = (Deque) q2;
// Runtime error
d.deleteRear();

Dynamic Binding

• Dynamic Binding
  – A mechanism by which, when the compiler can't determine which method implementation to use in advance, the runtime system (JVM) selects the appropriate method at runtime, based on the class of the object.

  – The process of binding a call to a particular method. This is performed dynamically at run-time.
Dynamic Binding Example

```java
class A {
    void p() { System.out.println("A.p"); }
    void q() { System.out.println("A.q"); }
    void f() { p(); q(); }
}
class B extends A {
    void p() { System.out.println("B.p"); }
    void q() { System.out.println("B.q"); super.q(); }
}
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.f();
        a = new B();
        a.f();
    }
}
```

Dynamic Binding Example (Cont’d)

```java
class A {
    public void q() {
        System.out.println("A.q");
    }
}
class B extends A {
    public void f() {
        System.out.println("B.f");
        super.q();
    }
}
class C extends B {
    public void f() {
        System.out.println("C.f");
        super.q();
    }
}
class D extends C {
    public void f() {
        System.out.println("D.f");
        super.q();
    }
}
```

Print:

```
A.p
A.q
B.p
B.q
A.q
```

Print:

```
B.f()
A.q()
```

Abstract Class

- Abstract classes
  - Cannot be instantiated
  - Incomplete: subclasses fill in "missing pieces"

- To make a class abstract
  - `public abstract class Shape {...}
  - Contain one or more abstract methods
    - No implementation
    - E.g., `public abstract void draw();`

- Subclasses:
  - fill in "missing pieces" (i.e., overriding the abstract methods)
    - E.g., `Circle`, `Triangle`, `Rectangle` extends `Shape`
  - Each must implement if `Base` is concrete

Example of Abstract Class

```java
public class Base {
    public String m1() {
        return "Base.m1";
    }
}
interface Interface1 { String m2(); }
interface Interface2 { String m3(); }
public class Derived extends Base implements Interface1, Interface2 {
    public String m2() {
        return "Derived.m2";
    }
    public String m3() {
        return "Derived.m3";
    }
}
```

Interface

- No method implementations

- Java doesn’t allow multiple inheritance:
  - E.g., `... C extends A, B ...

- Instead, use `Interface`
  - E.g., `... C implements I1, I2 ...

- One class may implement multiple interfaces
  - Must implement all functions in those interfaces if class is concrete

Interface Example

```java
public class Base {
    public String m1() {
        return "Base.m1";
    }
}
interface Interface1 { String m2(); }
interface Interface2 { String m3(); }
public class Derived extends Base implements Interface1, Interface2 {
    public String m2() {
        return "Derived.m2";
    }
    public String m3() {
        return "Derived.m3";
    }
}
```
Interface Example (Cont’d)

interface Interface1 { String m(String s); }

interface Interface2 { String m(int i); }

public class Derived implements Interface1, Interface2 {
    public String m(String s) {
        return "Derived.Interface1.m";
    }
    public String m(int i) {
        return "Derived.Interface2.m";
    }
}

public class Derived2 implements Interface1, Interface2 {
    public String m1(String s) {
        return "Derived2.m1";
    }
    public String m2(String s) {
        return "Derived2.m2";
    }
}

Derived d = new Derived();
Derived2 derived2 = new Derived2();
tmp = derived2.m1("Hello");
tmp = derived2.m2();

Derived derive2 = new Derived2();
tmp = derived2.m1("Hello");
tmp = derived2.m2();
String Processing

- Class java.lang.String
- Class java.lang.StringBuffer
- Class java.util.StringTokenizer

URL:
- http://java.sun.com/j2se/1.4.2/docs/api/

Important Methods in Class String

- Method length
  - Determine String length
  - Unlike array, Strings do not have length instance variable
- Method charAt
  - Get character at specific location in String
- Method getChars
  - Get entire set of characters in String
- Method startWith
  - Tests if this string starts with the specified prefix
- Method split
  - Splits this string around matches of the given regular expression

Important Methods in StringBuffer

- Class StringBuffer
  - When String object is created, its contents cannot change
  - Can store characters based on capacity
  - Capacity expands dynamically to handle additional characters
- Method charAt
  - Return StringBuffer character at specified index
- Method setCharAt
  - Set StringBuffer character at specified index
- Method getChars
  - Return character array from StringBuffer
- Method append
  - Allow data values to be added to StringBuffer
- Method reverse
  - Reverse StringBuffer contents
- Method deleteCharAt
  - Allow characters to be removed from StringBuffer

Important Methods in Class StringBuffer

- Method length
  - Return StringBuffer length
- Method capacity
  - Return StringBuffer capacity
- Method setLength
  - Increase or decrease StringBuffer length
- Method ensureCapacity
  - Set StringBuffer capacity
  - Guarantee that StringBuffer has minimum capacity
Class StringTokenizer

- Tokenizer
  - Partition String into individual substrings
  - Use delimiter
  - Java offers java.util.StringTokenizer

Important Methods in Class StringTokenizer

- Constructor StringTokenizer(String str)
  - Constructs a string tokenizer for the specified string
- Constructor StringTokenizer(String str, String delim)
  - Constructs a string tokenizer for the specified string
- Method hasMoreTokens()
  - Tests if there are more tokens available from this tokenizer's string
- Method nextToken()
  - Returns the next token from this string tokenizer