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

```
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

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();
      }
}
```

Derived.java

```
package p2;
import p1.Base;

public class Derived extends Base
{  void h ()
   {  f(); }
   void k ()
   {  g(); }
}
```

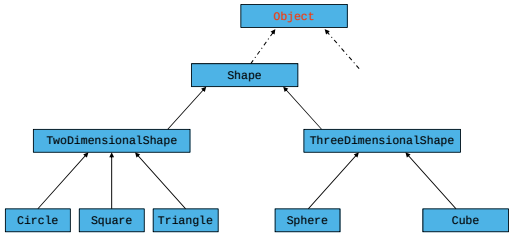
Test.java

```
package p2;
import p1.Base;

public class Test {
  public static void main(String[] args) {
     Base b = new Base();
     b.u = 1;
     // Compilation error
     b. v = 2;
     // Compilation error
     b.w = 3;
     // OK
     b.g();
     // Compilation error
     b.f();
     // Compilation error
     Derived d = new Derived();

     d.h();
     // OK
     d.k();
     // Compilation error
  }
}
```

Inheritance




```
public class B extends A
  {
…
}
```

```
public class A {
…
}
```


Class Object

```
class Object
{
   …
   public boolean equals ( Object obj ) { … }
   public String toString() { …}
   …
}
```


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!
if ( s.equals(t) ) System.out.println( "the same");
```

- 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 Rectnagle (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");
```


Overide equals ( )

```
public class Rectangle
{
   …
   public boolean equals ( Object obj )
   {
        Rectangle rec = (Rectangle) obj;
        return (w == rec.width()) && (h == rec.height());
   }
}
Rectangle rec1   =   new   Rectangle (p1, 5, 10);
Rectangle rec2   =   new   Rectangle (p2, 5, 10);
if  ( rec1.equals(rec2) ) System.out.println("the same!");
 // condition is true and print "the same"!
```


Method toString( )

```
Rectangle rec = new Rectangle (p1, 5, 10);
System.out.println( rec );

// prints something like Rectangle@73d6at
(class name +@+ hash code of object)
```


Overide toString( )

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

Rectangle rec = new Rectangle (p1, 5, 10);
System.out.println( rec );

Output:
width = 5, height = 10   (instead of Rectangle@73d6at)
```

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; }
}
```

```
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; }
}
```


Inheritance (Cont'd)

```
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() { … }
}
public class Deque extends Queue
{ …
  public void addFront( int x ) { … }
  public void deleteRear() { … }
}
```

```
Queue q1, q2;
Deque d;

d = new Deque();
q1 = 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

```
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(); }
}
```

Print:

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


Dynamic Binding Example (Cont'd)

```
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 q() {
    System.out.println("C.q()");
  }
}
public class D extends C {
  public static void main(String[] args){
    D d = new D();
    d.f();
  }
}
```

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 `draw` it is concrete

Example of Abstract Class

```
public class Base
{
    public String m1( )
    { return "Base.m1"; }
}

public abstract class Derived extends Base
{
    public abstract String m2( );
}

public class Derived2 extends Derived
{
    public String m2( )
    { return "Derived2.m2";}
}
```


Interface

- No method implementations
- Java doesn't allow multiple inheritance:
  - E.g., … C extends A, B …
- Instead, use Interface
  - E.g., … C implements I_1, I_2 …
- One class may implement multiple interfaces
  - Must implement all functions in those interfaces if class is concrete

Interface Example

```
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";
  }
```


Interface Example (Cont'd)

```
interface Interface1 { String m( String s ); }

interface Interface2 { String m( String s ); }

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


Interface Example (Cont'd)

```
interface Interface1 { String m( String s ); }
interface Interface2 { void m( String s ); }

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

  public void m(int i) {
    System.out.println( "Derived.Interface2.m");
  }
}

Derived d = new Derived();
d.m(10); // Compilation error
```


Interface (Java) vs. Multiple Inheritance (C++)

```
abstract class A {
  abstract public void f()
}
class B extends A {
  public void f(){
    System.out.println("B.f()");
  }
}
class C extends A {
  public void f() {
    System.out.println("C.f()");
  }
}
public class D extends B, C {
  public static void main(String[] args){
    D d = new D();
    d.f();
  }
}
```

In C++:

```
D* d = new D;

d->f();
// Compilation error

d->B::f();
// Legal
```


Interfaces vs. Abstract Classes

- A class may implement several interfaces
- An interface cannot provide any code at all
- Static final constants only
- A class may extend only one abstract class
- An abstract class can provide partial code
- Both instance and static constants are possible

Combination of Abstract Class and Interface

```
public class Base
{   public String m1( )
    { return "Base.m1"; }
}
interface Interface1
{   String m1( );
    String m2( );
}
public abstract class Derived extends Base implements
    Interface1
{   public String m2( )
    {     return "Derived.m2";    }
}
public class Derived2 extends Derived
{   public String m1( )
    {     return "Derived2.m1";   }
}
```

```
Derived2 derived2 = new
                    Derived2();
Base base = derived2;


tmp = derived2.m1( "Hello" );
  // tmp is "Derived2.m1"

tmp = base.m1( "Hello" );
  // tmp is "Derived2.m1"

tmp = derived2.m2();
  // tmp is "Derived.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
    - Like arrays, `String`s always "know" their size
    - Unlike array, `String`s 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 Class String

- Comparing `String` objects
  - Method `equals`
  - Method `equalsIgnoreCase`
  - Method `compareTo`
- Search for characters in `String`
  - Method `indexOf`
  - Method `lastIndexOf`
- Create `String`s from other `String`s
  - Method `substring`
- Concatenate two `String` objects Method `concat`
  - Method `concat`
- `String` provides `static` class methods
  - Method `valueOf`
    - Returns `String` representation of object, data, etc.

Class StringBuffer

- Class `StringBuffer`
  - When `String` object is created, its contents cannot change
  - Used for creating and manipulating dynamic string data
    - i.e., modifiable `String`s
  - Can store characters based on capacity
    - Capacity expands dynamically to handle additional characters
  - Uses operators + and += for `String` concatenation

Important Methods in StringBuffer

- 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 `insert`
  - Allow data-type values to be inserted into `StringBuffer`
- Methods `delete` and `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