# Principles of Computer Science I

**Prof. Nadeem Abdul Hamid**

**CSC 120A - Fall 2004**

**Lecture Unit 12**

**Packages, Scope, Access, and Copy Constructors**

## Packages

- Java allows grouping related classes into a package
- A Java package can be viewed as corresponding to a folder/directory on the hard drive
- Each package has a name
  - Each Java source code file that is included in that package will have this statement at the top of the file:

```
package someName;  // someName is the package's name
```

- Files in a package can be "bundled up" into a single archive file with extension .jar ("Java archive")
  - Allows for easy exchange/reuse of Java class files and applications
- Shapes example…



## Scope

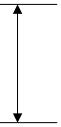
- Region of program code where it is legal to reference (use) an identifier
- Scoping rules are static: have to do with the program text, not run-time state
- Identifiers may be introduced as
  - the name of a variable in a variable declaration or parameter list
  - the name of a method in a method definition
  - This is called the *defining* or *naming occurrence* of the identifier
- Example

```
public class Counter {
  private int count;          // (1)
  ...
  public void increment() {
     count = count + 1;       // (2)
  }
}
```

## Scope of Local Variables

```
public class C {
    public void m(int x) {
        // can reference x here
        // but not y
        int y;
        // can reference both
        // x and y here
    }

    // can't reference x or y here
}
```





## Scope of Local Variables (cont)

- **Local variable definition can appear anywhere in method body**
  - **Scope extends to end of compound statement containing the definition**

```
public void m (int x) {
  if (x > 0) {
    int i = 1;
    ...
  }
}
```

  - **Method can contain definitions of several distinct variables with same name, but scopes must not overlap**



## Examples

```
public double m(int x) {
   double result;
   if (x > 1) {
       int sum = x;
       result = sum/2;
   } else {
       double sum = x;
       result = sum/2;
   }
   return result;
}
```

```
public double m(int x) {
   double result;
   double sum = x;
   if (x > 1) {
       int sum = x;
       result = sum/2;
   } else {
       result = sum/2;
   }
   return result;
}
```

## Shadowing

- A local variable may "hide" the definition of a field (instance variable) with the same name
- Use the this keyword, referring to an object itself, to reference shadowed object fields

```
public class Circle {
   private int radius;
   ...
   public void setRadius(int radius) {
       this.radius = radius;
   }
   ...
}
```



## Scope of Class Variables

- Class variable: a field defined with static in front
  - Cannot use one class variable to initialize another before first one is declared
  - Local variables can shadow definitions
- Not valid:

```
public static double twoPI = PI * 2;
public static double PI = 3.14;
```

- Valid:

```
public static double PI = 3.14;
public static double twoPI = PI * 2;
```



## Scope of Class Variables (cont).

- Rules of previous slide only apply to reference in other class variable initialization expressions

```
public class Circle {
    ...
    public static int circumferance(Circle c){
          return c.radius * twoPI;
    }
    ...
    public static double PI = 3.14;
    public static double twoPI = PI * 2;
}
```

## Accessibility Rules in Java

- Textbook calls this "external scope"
- We have labeled features of a class (methods, fields, named constants, *etc.*) as
  - public
  - private
  - protected
  - (none of the above)
- These are called "access modifiers"



## Accessibility Table

| External access | public | protected | (default) package | private |
|---|---|---|---|---|
| Same package | yes | yes | yes | no |
| Derived class in another package | yes | yes (inheritance only) | no | no |
| User code | yes | no | no | no |



## Circle

```
public class Circle {
   private int r;

   public Circle(int radius) { this.r = radius; }

   public boolean equals(Object obj) {
        if (obj instanceof Circle)
                return r == ((Circle)obj).r;
        else return false;
   }

   public int radius() { return r; }
}

...
Circle c1 = new Circle(10);
Circle c2 = new Circle(12);
if (c1.equals(c2)) ...
```

## ColoredCircle

```
// something's wrong…

public class ColoredCircle extends Circle {
   private Color c;

   public ColoredCircle(int radius, Color color) {
        super(radius);
        c = color;
   }

   public boolean equals(Object obj) {
        if (obj instanceof ColoredCircle)
            return r == ((Circle)obj).r     &&
                        c == ((ColoredCircle)obj).c;
        else return false;
   }

   public Color color() { return c; }
}
```



## Circle, revised

```
public class Circle {
   protected int r;     // now ColoredCircle can access this

   public Circle(int radius) { this.r = radius; }

   public boolean equals(Object obj) {
        if (obj instanceof Circle)
              return r == ((Circle)obj).r;
        else return false;
   }

   public int radius() { return r; }
}

...
Circle c1 = new Circle(10);
Circle c2 = new Circle(12);
if (c1.equals(c2)) ...
```



## Package-accessible Circle

```
public class Circle {
   int r;     // any class in same package can access this

   public Circle(int radius) { this.r = radius; }

   public boolean equals(Object obj) {
        if (obj instanceof Circle)
              return r == ((Circle)obj).r;
        else return false;
   }

   public int radius() { return r; }
}

...
Circle c1 = new Circle(10);
Circle c2 = new Circle(12);
if (c1.equals(c2)) ...
```

## Accessibility Summary

- Same rules for fields shown in these slides apply to methods and classes
  - Class accessibility overrides method/field accessibility
- To decide which access modifiers to use, ask
  - Do I want to access this field/method/constant from other classes in the same package, from derived classes, or from any arbitrary user code?



## Copying Objects

```
package shapes;

public class Circle extends Shape {

   public final static double PI = 3.14;

   double radius;

   public Circle(double radius) {
      //if (radius < 0) radius = 0;
      //this.radius = radius;
      setRadius(radius);
   }

   /* update method */
   public void setRadius(double radius) {
      if (radius < 0) radius = 0;
      this.radius = radius;
   }

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

```
Circle c1 = new Circle(4.5);
Circle c2 = c1;
System.out.println(c1.area());
System.out.println(c2.area());
c2.setRadius(1);
System.out.println(c1.area());
System.out.println(c2.area());
```

?



## Copy Constructor

```
public class Circle extends Shape {

   public final static double PI = 3.14;

   double radius;

   public Circle(double radius) {
      //if (radius < 0) radius = 0;
      //this.radius = radius;
      setRadius(radius);
   }

   // copy constructor
   public Circle(Circle other) {
      this.radius = other.radius;
   }

   ...
```

```
Circle c3 = new Circle(c1);
System.out.println(c1.area());
System.out.println(c3.area());
c3.setRadius(3);
System.out.println(c1.area());
System.out.println(c3.area());
```

## Shallow Copying

```
Drawing d1 = new Drawing(1);
Drawing d2 = new Drawing(d1);
d1.printArea();
d2.printArea();
d1.resize(5);
d1.printArea();
d2.printArea();
```

**Output:**

```
9.42
9.42
235.5
235.5
```

```
class Drawing {
   Circle[] circles;

   public Drawing(int radii) {
      circles = new Circle[3];
      for (int i=0; i<3; i++)
         circles[i] = new Circle(radii);
   }

   public Drawing(Drawing other) {
      this.circles = other.circles;
   }

   public void resize(int radii) {
      for (int i=0; i<3; i++)
         circles[i].setRadius(radii);
   }

   public void printArea() {
      double area = 0.0;
      for (int i=0; i<3; i++)
         area += circles[i].area();
      System.out.println(area);
   }
}
```



## Deep Copying

```
Drawing d1 = new Drawing(1);
Drawing d2 = new Drawing(d1);
d1.printArea();
d2.printArea();
d1.resize(5);
d1.printArea();
d2.printArea();
```

**Output:**

```
9.42
9.42
235.5
9.42
```

```
class Drawing {
   Circle[] circles;

   public Drawing(int radii) {
      circles = new Circle[3];
      for (int i=0; i<3; i++)
         circles[i] = new Circle(radii);
   }

   public Drawing(Drawing other) {
      this.circles = new Circle[3];
      for (int i=0; i<3; i++)
         this.circles[i] = new Circle(other.circles[i]);
   }

   public void resize(int radii) {
      for (int i=0; i<3; i++)
         circles[i].setRadius(radii);
   }

   public void printArea() {
      double area = 0.0;
      for (int i=0; i<3; i++)
         area += circles[i].area();
      System.out.println(area);
   }
}
```

## Copies

- Deep copy
  - Operation to create a copy of an object by traversing all the way through object references, down to primitive types
- Shallow copy
  - Operation to copy an object by simply duplicating references