Lecture Notes
Chapter #7
Objects & Classes

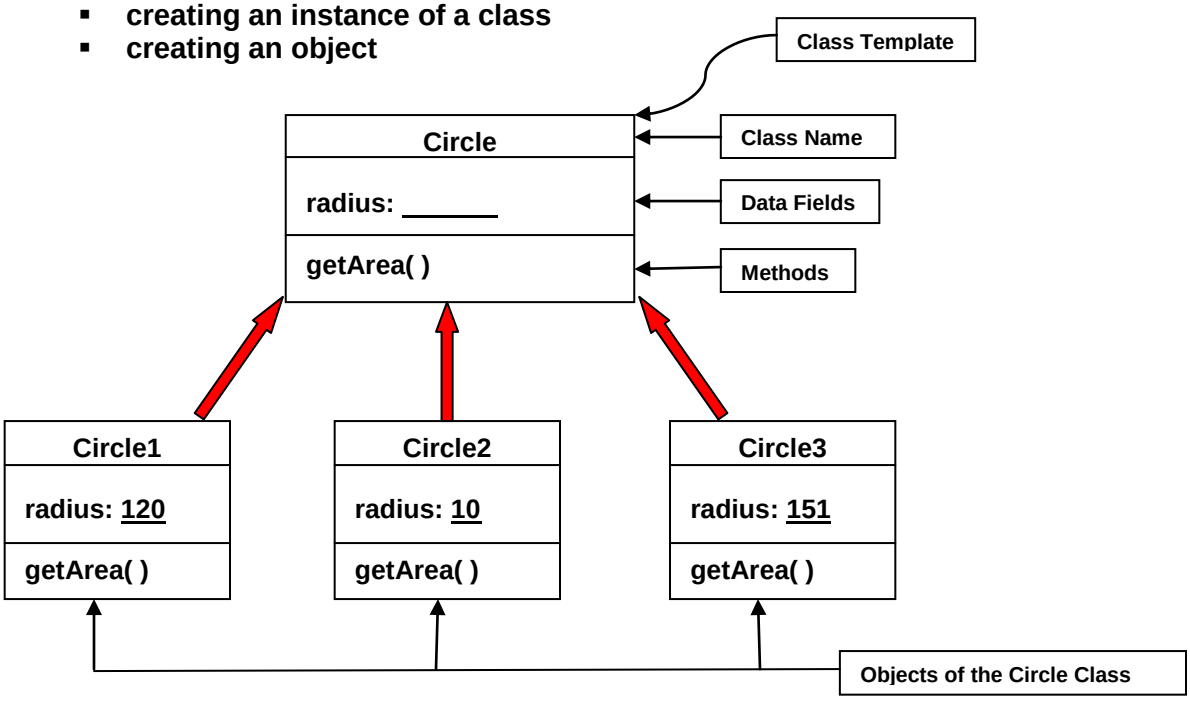
Classes & Objects are used when using and/or creating GUI Components

1. Defining Classes

- **Object**
  - represents a distinct entity in the real world
  - has a unique identity, state, & a set of distinct behaviors
    - the identity is established by the object's name
    - the state is represented by data fields with their current values, i.e., the object's properties
    - the set of behaviors are defined by a set of methods

- **Class**
  - template that defines a common set of objects
  - objects of the same type are defined by the class

- **Object (revisited)**
  - object is an instance of a class
  - instantiation
    - creating an instance of a class
    - creating an object

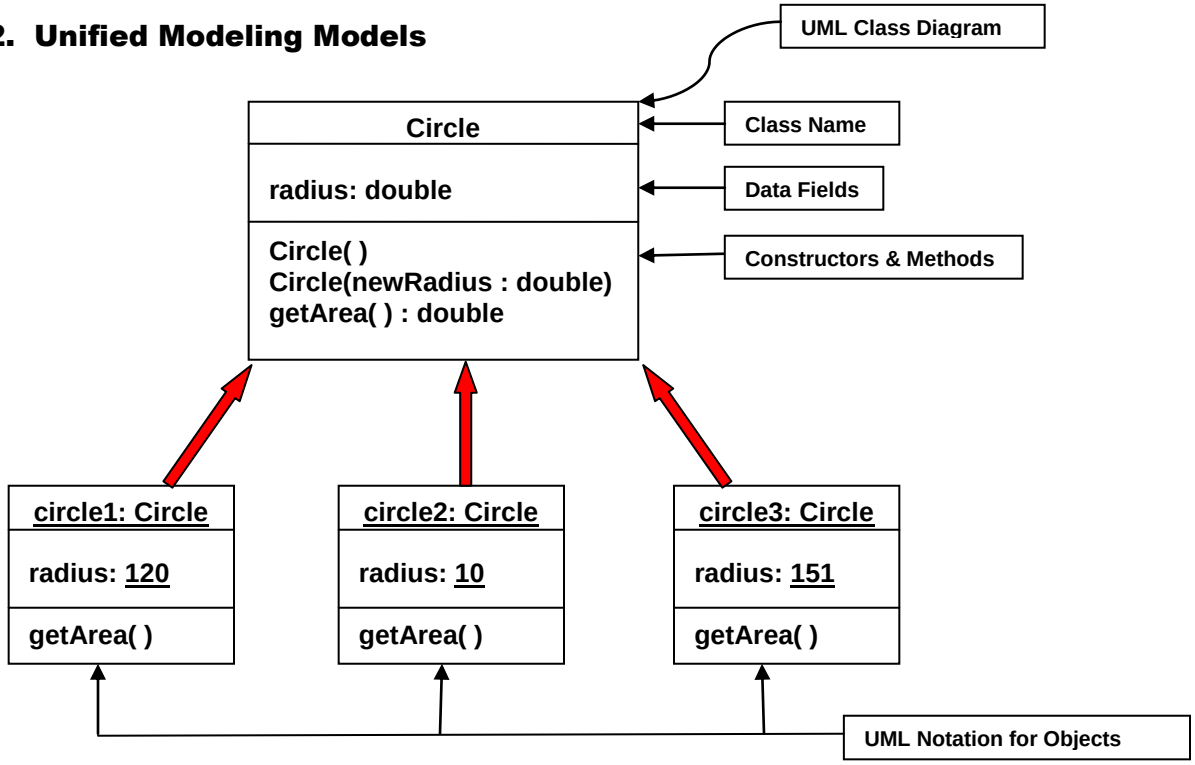


- **Java Class**
  - data fields are defined by variables
  - behaviors are defined by methods
  - constructors are special methods
    - invoked when an object is created
    - designed to perform initializing actions
    - has the same name as the class

```
class Circle
{
   double radius = 1.0;

   Circle( ){ }

   Circle( double newRadius))
   {
       Radius = newRadius;
   }

   double getArea( )
   {
      return radius * radius * Math.PI;
   }
}
```

Circle class does not have a main method
➔
- Circle is not executable
- Circle class is used as a definition to declare and create Circle type objects

2. Unified Modeling Models



Constructor denotation:	**ClassName(parameterName : parameterType)**
Method denotation:	**methodName(parameterName : parameterType) : returnType**

3. Constructors

- **used to construct objects, i.e.,**
  - **new ClassName(arguments)**

- **constructors may be overloaded**
  - **construct objects with different initial data values**
    - **new Circle(5);**
    - **new Circle(150);**

- **no-argument constructor no-arg Circle( )**
  - **used to create object without initializing data fields**

- **default constructor is invoked automatically if no constructors are declared in the class**

- **constructors have the same name as the class**

- **constructors do not have a return type, not even void**

- **constructors are invoked using the new operator when creating an object**

4. Reference Variables

- **used to hold objects**

- **class defines a reference variable for all objects of that type**
  - **Classname objectRefVar; Circle c1;**

- **creating an object using a reference variable**
  - **Circle myCircle = new Circle( );**

Arrays are treated as objects in java

- **Arrays are created using the new operator**
- **An array variable is a reference variable holding a reference to an array**

5. Accessing Data Fields & Methods in an Object

- object member access operator, i.e., dot operator
  - Circle c1 = new Circle(5); // creates object held by c1 reference variable
  - c1.radius; // accesses the radius in the object held by c1 reference variable
  - c1.getArea( ); // accesses the getArea( ) method in the object held by c1

- instance variables
  - variable whose value is dependent on a particular instance
- instance methods
  - method which can only be invoked on the particular instance
- anonymous objects
  - object created without a reference variable
  - held by an encompassing method, e.g.,

    System.out.println(new Circle(5).getArea( ));

6. Listing 7.1

```
public class TestCircle1
{
   public static void main(String [ ] args)
   {
      Circle myCircle = new Circle1(5.0);
      System.out.println("Radius: " + myCircle.radius + "\t Area: " + myCircle.getArea( ) );

      Circle yourCircle = new Circle1( );
      System.out.println("Radius: " + yourCircle.radius + "\t Area: " + yourCircle.getArea( ) );

      yourCircle.radius = 100;
      System.out.println("Radius: " + yourCircle.radius + "\t Area: " + yourCircle.getArea( ) );
   }
}
```

```
class Circle1
{
  double radius;

  Circle1( ) { radius = 1; }

  Circle1( double newRadius )
  {
     radius = newRadius;
  }

  getArea( )
  {
     return radius * radius * Math.PI;
  }

}
```

More than one class may be placed in a single file, but only one class may be a public class.

The public class must have the same name as the file name, i.e., the file name for the current public class must be TestCircle1.java

Listing 7.1 may also be constructed with each class stored in a separate file.

7. Listing 7.2

```
public class Circle1
{
   public static void main(String [ ] args)
   {
      Circle myCircle = new Circle1(5.0);
      System.out.println("Radius: " + myCircle.radius + "\t Area: " + myCircle.getArea( ) );

      Circle yourCircle = new Circle1( );
      System.out.println("Radius: " + yourCircle.radius + "\t Area: " + yourCircle.getArea( ) );

      yourCircle.radius = 100;
      System.out.println("Radius: " + yourCircle.radius + "\t Area: " + yourCircle.getArea( ) );
   }

      double radius;

      Circle1( ) { radius = 1; }

      Circle1( double newRadius )
      {
     radius = newRadius;
      }

      getArea( )
      {
     return radius * radius * Math.PI;
      }
}
```

The program in Listing 7.1 is rewritten in a single class in Listing 7.2

Any method stored in the Math class can be invoked without executing the Math class or creating an instance of the Math class. All the methods in the Math class are defined to be static methods, i.e., class methods, which can be invoked without executing or creating an instance of the Math class.

The methods in the Circle1 class are instance methods, not static methods, which can only be invoked by the objects which are derived from the Circle1 class. They must be invoked by using a reference to a particular object, e.g., c1.getArea( );

8. Reference Data Fields

```
class Student
{
   String name;
   int age;
   boolean isScienceMajor;
   char gender;
}
```

reference type data fields

if a data field of the reference type does not reference an object, the data field contains the null value which is a literal of the reference type, e.g., the String reference variable name does not contain a String object, hence it is assigned the null value.

data field default values

reference types	null
numeric type	zero (0)
boolean type	false
char type	'\u0000'

```
class Test
{
   public static void main(String [ ] args)
   {
     Student student = new Student( );
     System.out.println("Name: " + student.name );
     System.out.println("Age: " + student.age );
     System.out.println("Science Major: " + student.isSciencemajor( ) );
     System.out.println("Gender: " + student.gender );
   }
}
```

Valid code since all instance variables have default values

```
class Test
{
   public static void main(String [ ] args)
   {
    int x;
    String y;
     System.out.println("x: " + x );
     System.out.println("y: " + y );
   }
}
```

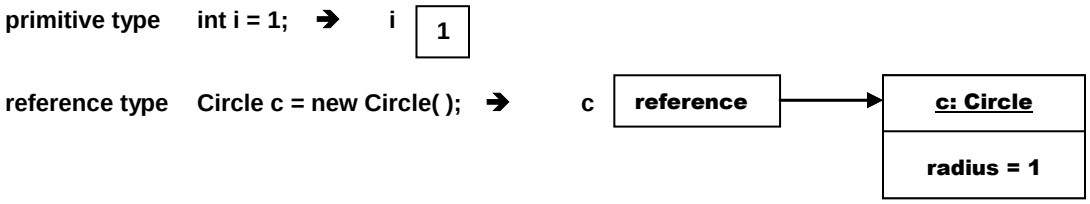
Compilation errors since local variables are not initialized

NullPointerException

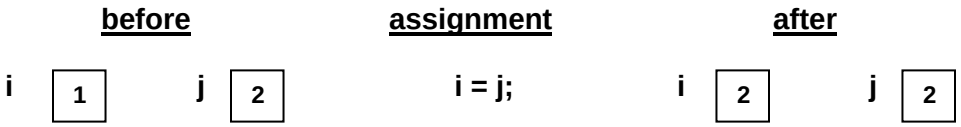
Occurs when a reference variable with a null value is invoked

9. Primitive Variable Types versus Reference Variable Types

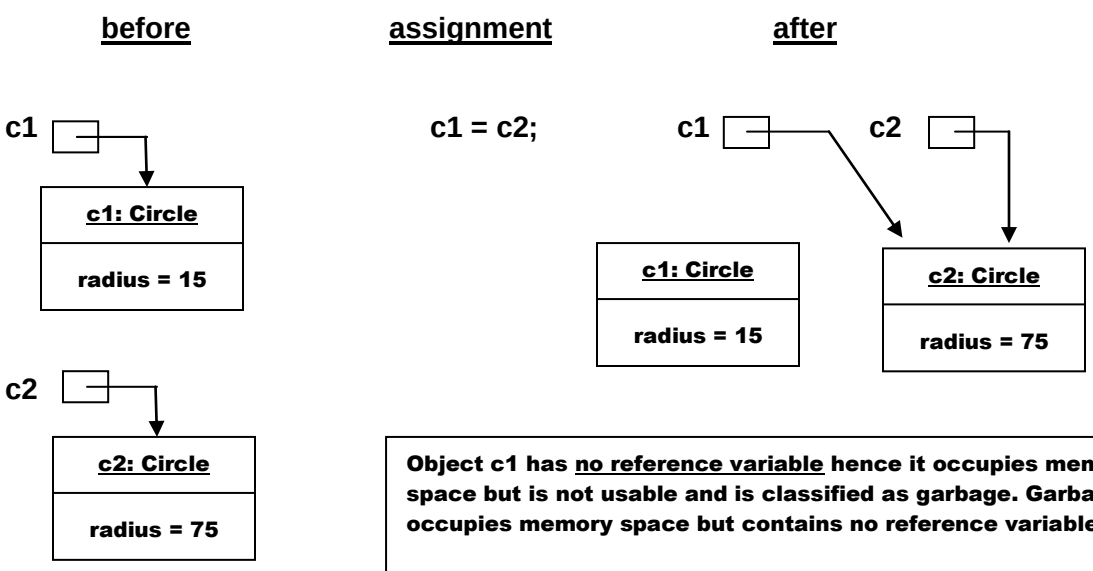
- every variable represents a memory location that holds a value of the declared type
  - a variable of the primitive type holds a value of the declared primitive type
  - a variable of a reference type holds a reference to the objects storage location



- primitive variable assignments



- reference variable assignments



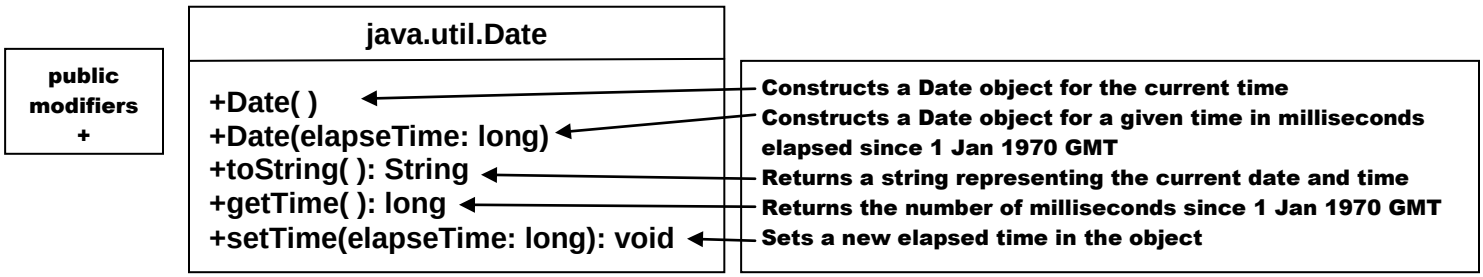
Object c1 has no reference variable hence it occupies memory space but is not usable and is classified as garbage. Garbage occupies memory space but contains no reference variables.

The Java runtime system detects garbage, deletes it, and automatically reclaims the unused space that it occupies; a process called garbage collection.

When an object is no longer required, explicitly assign the reference variable to the null value. If the object is no longer held by a reference variable, the JVM automatically collects the memory space, i.e., it deletes the object.

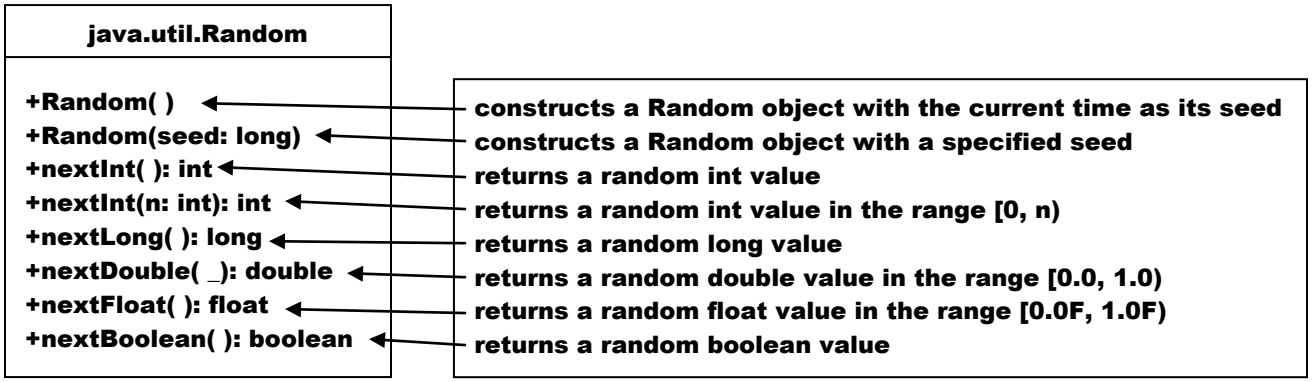
10. Java Library Classes

a. Date Class




```
java.util.Date date = new java.util.Date( );
System.out.println("Elapsed time: " + date.getTime( ) + " milliseconds");
System.out.println(date.toString( ));
```

b. Random Class



If two Random objects have the same seed, they will generate identical sequences of numbers. Generating the same sequence of random values is useful in software testing; testing the program using a fixed sequence of numbers provides a method of finding program errors.

c. GUI Components

- Java GUI Classes
  - JFrame creates frames
  - JButton creates buttons
  - JRadioButton creates radio buttons
  - JComboBox creates combo boxes
  - JList creates lists

- **Listing 7.3**

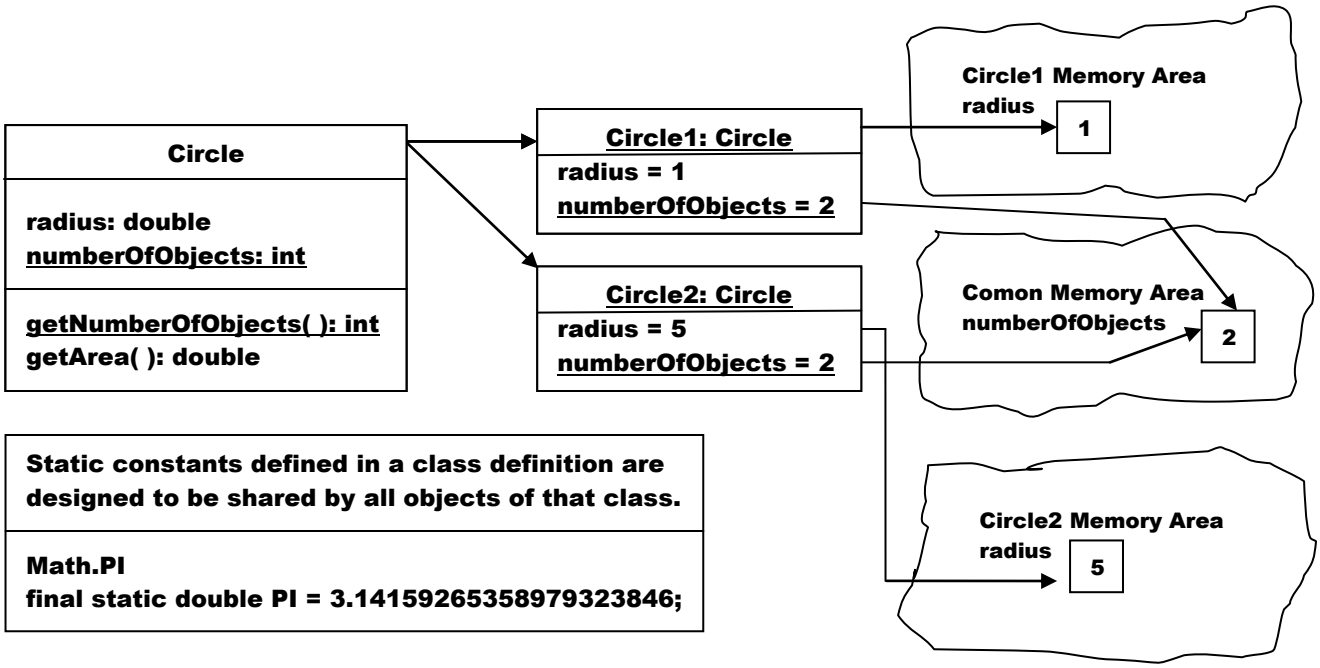
```
import javax.swing.JFrame
public class TestFrame
{
   public static void main(String [ ] args)
   {
         JFrame frame1 = new JFrame( );
         frame1.setTitle("Window 1");
         frame1.setSize(200, 150);
         frame1.setLocation(200, 100);
         frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame1.setVisible(true);

         JFrame frame2 = new JFrame( );
         Frame2.setTitle("Window 2");
         Frame2.setSize(200, 150);
         Frame2.setLocation(410, 100);
         Frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         Frame2.setVisible(true);
   }
}
```

- **Listing 7.4**

```
import javax.swing.*;
public class GUIComponents
{
   public static void main(String [ ] args)
   {
      JButton jbtOK = new JButton("OK");
      JLabel jlblName = new JLabel("Enter Name: ");
      JTextField jtfName = new JTextField("Type Name Here ");
      JCheckBox jchkBold = new JCheckBox("Bold");
      JRadioButton jrbRed = new JRadioButton("Red");
      JComboBox jcboColor = new JComboBox(new String [ ] {"Red", "Green", "Blue"});

     JPanel pane1 = new JPanel( );
     pane1.add(jbtOK);
     pane1.add(jlblName);
     pane1.add(jtfName);
     pane1.add(jchkBold);
     pane1.add(jbrRed);
     pane1.add(jcboColor);

     JFrame frame = new JFrame( );
     Frame.add(panel);
     frame.setTitle("Show GUI Components");
     frame.setSize(450, 100);
     frame.setLocation(200, 100);
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setVisible(true);
   }
}
```

11. Static Variables, Constants, & Methods

- instance variables are stored in a specific object, i.e., instance variables are not shared among objects of the same class
- static variables, i.e., class variables store values in a common memory location shared by all instances of the class, i.e., if the value is changed, all objects have access to the new value
- static variables can be accessed without creating an instance of the class



a. Listing 7.5

```
public class Circle2
{
  double radius;

  static int numberOfObjects = 0;

  Circle2( )
  {
    radius = 1.0;
    numberOfObjects++;
  }

  Circle2( double newRadius )
  {
    radius = newRadius;
    numberOfObjects++;
  }

  static int getNumberOfObjects( )
  {
    return numberOfObjects;
  }
}
```

e. Listing 7.6

```
public class TestCircle2
{
  public static void main(String [ ] args)
  {
    Circle2   c1 = new Circle2( );
    System.out.println("c1 radius: " + c1.radius +
                       "\t Number of Objects: " + c1.numberOfObjects);

    Circle2   c2 = new Circle2( 5 );

    c1.radius = 9;
    System.out.println("c1 radius: " + c1.radius +
                       "\t Number of Objects: " + c1.numberOfObjects);

    System.out.println("c2 radius: " + c2.radius +
                       "\t Number of Objects: " + c2.numberOfObjects);
  }
}
```

```
c1 radius: 1.0      Number of Objects: 1
c1 radius: 9.0      Number of Objects: 2
c2 radius: 5.0      Number of Objects: 2
```

Both **c1.numberOfObjects** and **c2.numberOfObjects** can be replaced by **Circle2. numberOfObjects**
also
Circle2. numberOfObjects can be replaced by **Circle2. getNumberOfObjects( )**

Use

a. **ClassName.methodName(arguments)** to invoke static methods
b. **Classname.staticConstant** to invoke static constants
c. **Classname.staticVariable** to invoke static variables

In JDK 1.5 use

import static java.lang.Math.*;

to import all static variables, static constants, and static methods from the java.lang.Math class

```
public class Foo
{
  int i = 5;
  static int k = 2;

  public static void main(String [ ] args)
  {
     int j = i;
     m( );
  }

  public void m1( )
  {
      i = i + k + m2( i, k );
  }

  public static int m2(int i, int j)
  {
     return (int)(Math.pow(i, j));
  }

}
```

Incorrect usage since the instance variable i and the instance method m1( ) cannot be used in a static method; a static method must be available for use independent from the instantiation of any particular object thus they may not rely upon instance variables or instance methods.

Correct usage since the instance variable i and the static variable j defined as part of the enclosing class can be used in an instance method.

Correct usage since the parameters i & j are different from the variables i & j defined above.

A variable or method that is dependent upon a particular instance of the class must be an instance variable or an instance method.

A variable or method that is not dependent upon a particular instance of the class should be a static variable or a static method.

radius and **getArea( )** are dependent upon the existence of a specific circle object and must, therefore be an instance variable and an instance method.

PI, random( ), pow( ), sin( ), cos( ), floor( ), ceil( ), etc. are not dependent on the existence of a particular object and are therefore static constants and static methods.

The method **factorial(int n)** is not dependent upon the existence of a specific object and should be declared to be static

12. Visibility Modifiers

- control access to classes, methods, and data fields

a. **public** makes classes, methods, and data fields accessible from any class

b. **private** makes methods, and data fields accessible only from within its own class

c. **package-access, package-private or default** access occurs if there is no public or private modifier; under this circumstance, the classes, methods, and data fields are accessible by any class in the same package

13. Packages

- are used to organize classes
- implementation of a package
  - the first non-comment and non-blank statement in program must be
    package packageName;
  - if package statement is not included, the class is placed in a default class which consists of just the single class
- good programming practice
  - place classes in explicit packages
  - Supplement III.F, "Packages"

```
package p1;

public class C1
{
  public int x;
  int y;
  private int z;

  public void m1( ) { }

  void m2( ) { }

  private m3( ) { }
}
```

```
public class C2
{
  void aMethod( )
  {
    C1 o = new C1( );
    // can access o.x
    // can access o.y
    // cannot access o.z

    // can invoke o.m1( )
    // can invoke o.m2( )
    // cannot invoke o.m3( )
```

```
package p2;

public class C3
{
  void aMethod( )
  {
    C1 o = new C1( );
    // can access o.x
    // cannot access o.y
    // cannot access o.z

    // can invoke o.m1( )
    // cannot invoke o.m2( )
    // cannot invoke o.m3( )
```

- Private restricts access to its defining class
- Default modifier restricts access to a package
- Public access enables unrestricted access

default, package-access, package-private class modifier limits visibility to package

```
package p1;

class C1
{
   public int x;
   int y;
   private int z;

  public void m1( ) { }

  void m2( ) { }

   private m3( ) { }
}
```

```
public class C2
{
   void aMethod( )
   {
     C1 o = new C1( );
     // can access o.x
     // can access o.y
     // cannot access o.z

     // can invoke o.m1( )
     // can invoke o.m2( )
     // cannot invoke o.m3( )
```

```
package p2;

public class C3
{
   void aMethod( )
   {
     // cannot access C1
     // can access C2
     // cannot access o.y
     // cannot access o.z

     // cannot invoke o.m1( )
     // cannot invoke o.m2( )
     // cannot invoke o.m3( )
```

- visibility modifiers specify how data fields and methods can be accessed from outside the class
- there are no restrictions on accessing data fields and methods from inside the class
- the private modifier applies only to members of a class
- the public modifier can apply to a class or members of a class
- public and private modifiers cannot be used on local variables

```
public class Foo
{                                    // OK
   private boolean x;
   public static void main(String[ ] args)
   {
     Foo foo = new Foo( );
     System.out.println(foo.x);
     System.out.println(foo.convert(x));
   }

   private int convert( )
   {
     return x ? 1 : -1;
   }
}
```

```
public class Test
{                                    // NOPE !
   public static void main(String[ ] args)
   {
     Foo foo = new Foo( );
     System.out.println(foo.x);
     System.out.println(foo.convert(x));
   }
}
```

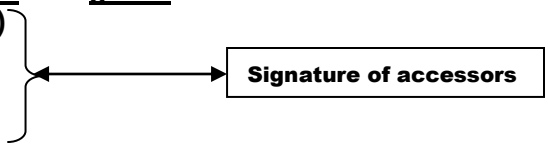
- class Test cannot access x nor convert( ) since they are declared private in a separate class
- class Foo can access both x & convert( ) since they are declared private in the same class

To prohibit a user from creating an instance of a class, include a no-arg constructor with a private modifier; the Math class includes the constructor <u>private Math( ) { };</u>

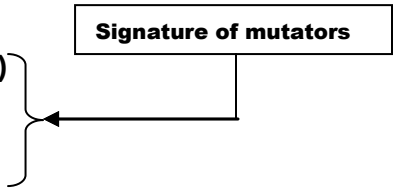
14. Data Field Encapsulation

- **prevent the direct modification of static (class) data fields, e.g.,**
  - **numberOfObjects should only be modified when a new object is created**
  - **numberOfObjects should be declared to be private in the class definition**
    - **to modify the value of the data field, provide a private setNumberOfObjects method in the class definition**
    - **to provide access to the current value of the numberOfObjects data item in the objects, provide a public getNumberOfObjects method in the class definition**

- **a get method is referred to as an accessor or a getter**
  - **public returnType getPropertyName( )**
    - **public int getNumberOfObjects( )**
  - **public boolean isPropertyName( )**
    - **public int isPrimeNumber( )**

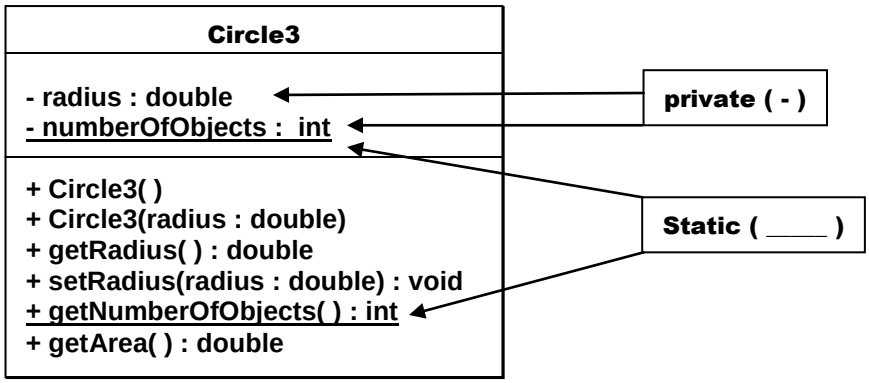


- **a set method is referred to as a mutator or a setter**
  - **private void setPropertyName(dataType propertyValue)**
    - **private void setNumberOfObjects(int n)**
  - **public void setPropertyName(dataType propertyValue)**
    - **public void setRadius(double r)**



a. Listing 7.7

```
public class Circle3
{
  private double radius;
  private static int numberOfObjects = 0;

  public Circle3
  {
    numberOfObjects++;
  }

  public Circle3(double newRadius)
  {
    radius = newRadius;
    numberOfObjects++;
  }

  public double getRadius( ){ return radius; }

  public void setRadius(double newRadius)
  {
    radius = (newRadius >= 0) ? newRadius : 0;
  }

  public static int getNumberOfObjects( ){ return numberOfObjects; }

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



b. Listing 7.8

```
public class TestCircle3
{
   public static void main(String [ ] args)
   {
      Circle3 myCircle = new Circle3(5.0);
      System.out.println("Radius: " + myCircle.getRadius( ) + "\tArea: " + myCircle.getArea( );
      myCircle3.setRadius(myCircle3.getRadius( ) * 1.1);
      System.out.println("Radius: " + myCircle.getRadius( ) + "\tArea: " + myCircle.getArea( );
   }
}
```

Compilation Protocol

When TestCircle3.java is submitted to compilation, Circle3.java will be automatically compiled before TestCircle3.java if it has been modified since the last compilation

15. Passing Objects to Methods

- **passing an object is accomplished by actually passing a reference to the object**

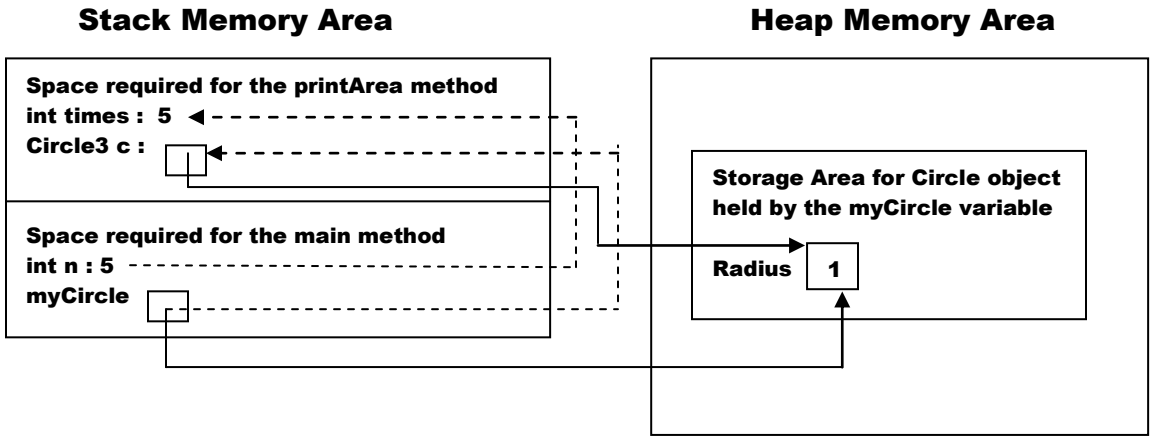
a. Passing an Object of Type Circle3

```
public class TestPassObject
{
public static void main(String [ ] args)
{
Circle3 myCircle = Circle3(5.0);
printCircle(myCircle);
}

public static void printCircle(Circle3 c)
{
System.out.println("Radius: " + myCircle.getRadius( ) + "\tArea: " + myCircle.getArea(
);
}
}
```

b. Listing 7.9

```
public class TestPassObject
{
  public static void main(String [ ] args)
  {
    Circle3 myCircle = Circle3(1);
    int n = 5;
    printAreas(myCircle, n);
    System.out.println("\nRadius: " + myCircle.getRadius( ));
    System.out.println("n: " + n);
  }

  public static void printAreas(Circle3 c, int times)
  {
    System.out.println("Radius \t\tArea");          // print report header
    while( times >= 1 )
    {
      System.out.println( c.getRadius( ) + "\t\t" + c.getArea( ));
      c.setRadius(c.getRadius( ) + 1);
      times--:
    }
  }
}
```

Creates report in Liang page 250



Pass by Reference ←→ Pass by Sharing
(semantics, connotation)

16. Array of Objects

Create an array of ten Circle objects

```
Circle [ ] circleArray = new Circle[10];
```

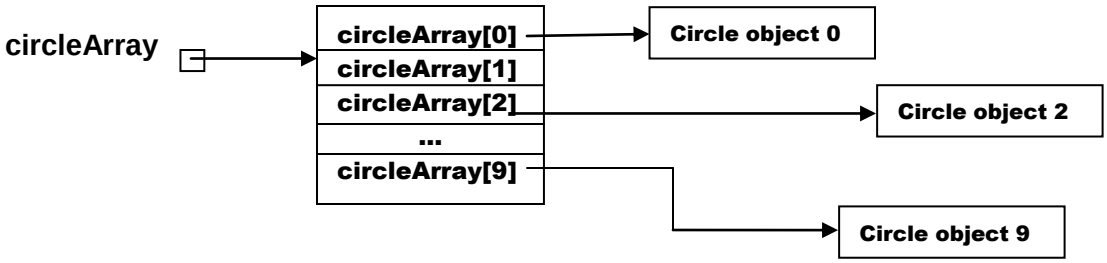
// each element in the array is a reference variable with a default value of null

Initialize the circleArray

```
for(int i = 0; i < circleArray.length; i++)
{
  circleArray[ i ] = new Circle( );
}
```

// each element in the array is a reference variable with a value of the new object

array of objects == array of reference variables



accessor

```
circleArray[1].getArea( );
```

➔

circleArray references the array
circleArray[1] references object 1
circleArray[1].getArea( ) access the getArea( ) method in object 1

a. **Listing 7.10**

Produces report in Liang page 252

```
public class TotalArea
{
  public static void main(String[ ] args)
  {
    Circle3[ ] circleArray;
    circleArray = createCircleArray( );
    printCircleArray(circleArray);
  }

  public static Circle3[ ] createCircleArray( )
  {
    Circle3[ ] circleArray = new Circle3[5];

    for(int i = 0; i < circleArray.length; i++)
    {
      circleArray[ i ] = new Circle3(Math.random( )* 100);
    }
  return circleArray;
  }

  public static void printCircleArray(Circle3[ ] circleArray)
  {
    System.out.println("Radius\t\t\t\t" + "Area");          // print report header

    for(int i = 0; i < circleArray.length; i++)
    {
      System.out.print(circleArray[ i ].getRadius( ) + "\t\t + circleArray[i].getArea( ) + '\n');
    }
    System.out.println("-----------------------------------------------------------------------");

    System.out.println("Total Area (all circles) \t" + sum(circleArray));
  }

  public static double sum(Circle3[ ] circleArray)
  {
    double sum = 0;

    for(int i = 0; i < circleArray.length; i++) { sum += circleArray[ i ].getArea( ); }

    return sum;
  }
}
```