APPinc.
Making Technology Reachable

Java Programming

RITA M. BARRIOS, Ph.D.

PRES. TRAINING & EDUCATION

Development strategy

Recommendations for managing complexity:

1. Write an English description of steps required (*pseudo-code*)
   - use pseudo-code to decide methods

2. Create a table of patterns of characters
   - use table to write loops in each method

```
#================#
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>............<>|
|<>............<>|
|  <>........<>  |
|    <>....<>    |
|      <><>      |
#================#
```

1. Pseudo-code

pseudo-code: An English description of an algorithm.

Example: Drawing a 12 wide by 7 tall box of stars

```
print 12 stars.
for (each of 5 lines) {
    print a star.
    print 10 spaces.
    print a star.
}
print 12 stars.
```

```
************
*          *
*          *
*          *
*          *
*          *
************
```

Pseudo-code algorithm

1. Line 1
   1 hash (#) sign , 16 equal (=) signs and 1 equal (#) sign

2. Top half
   - |
   - spaces (decreasing)
   - <>
   - dots (increasing)
   - <>
   - spaces (same as above)
   - |

3. Bottom half (top half upside-down)

4. Last Line
   1 hash (#) sign , 16 equal (=) signs and 1 equal (#) sign

```
#================#
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>............<>|
|<>............<>|
|  <>........<>  |
|    <>....<>    |
|      <><>      |
#================#
```

Methods from pseudocode

```
public class Mirror {
    public static void main(String[] args) {
        line();
        topHalf();
        bottomHalf();
        line();
    }

    public static void topHalf() {
        for (int line = 1; line <= 4; line++) {
            // contents of each line
        }
    }

    public static void bottomHalf() {
        for (int line = 1; line <= 4; line++) {
            // contents of each line
        }
    }

    public static void line() {
        // ...
    }
}
```

2. Tables

A table for the top half:

- Compute spaces and dots expressions from line number

line	spaces	line * -2 + 8	dots	4 * line - 4
1	6	6	0	0
2	4	4	4	4
3	2	2	8	8
4	0	0	12	12

```
#================#
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>............<>|
|<>............<>|
|  <>........<>  |
|    <>....<>    |
|      <><>      |
#================#
```

3. Writing the code

Useful questions about the top half:

- What methods can we use?
  - (think structure and redundancy)
- Number of (nested) loops per line?

```
#================#
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>............<>|
|<>............<>|
|  <>........<>  |
|    <>....<>    |
|      <><>      |
#================#
```

Partial solution – You finish the rest..

```java
public class Mirror {
      // Prints the expanding pattern of <> for the top half of the figure.
      public static void main(String [] args) {
            line();
            topHalf();
            bottomHalf();
            line();
      }

      // Prints the expanding pattern of <> for the top half of the figure.
      public static void topHalf() {
          for (int line = 1; line <= 4; line++) {
              System.out.print("|");

              for (int space = 1; space <= (line * -2 + 8); space++) {
                  System.out.print(" ");
              }

              System.out.print("<>");

              for (int dot = 1; dot <= (line * 4 - 4); dot++) {
                  System.out.print(".");
              }

              System.out.print("<>");

              for (int space = 1; space <= (line * -2 + 8); space++) {
                  System.out.print(" ");
              }

              System.out.println("|");
          }
      }

    public static void bottomHalf() {
        for (int line = 1; line <= 4; line++) {
            // contents of each line
        }
    }

    public static void line() {
        // ...
    }
}
```

```
#================#
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>............<>|
|<>............<>|
|  <>........<>  |
|    <>....<>    |
|      <><>      |
#================#
```

Stepwise Refinement

What exactly is *stepwise refinement*?
Answer: It is a process of programming, starting from an idea to finished, and refined, code

The main idea is to separate coding from problem-solving, and to proceed in small steps, even if the small steps seem like "a waste of time".

http://www.seas.gwu.edu/~drum/java/lectures/appendix/stepwise.html

Determine the Class of an Object

key.getClass().getName() method to find out the class

getClass() is defined in the Object class – can be called in ALL objects

- Returns a Class object that represents the object's class
- getName() method returns a string containing the name of the class

instanceof – Operator - comparison expression with a reference to an object on the left and a class name on the right

- Returns a boolean
  - True = object is an instance of the named class or subclass of the class
  - False = object is not an instance of

```
boolean check1 = “Texas” insanceof String; ← Returns true
Point pt = new Point(10, 10);
Boolean check2 = pt instanceof String;  ← Returns false
```

Chapter 5 – Classes and Methods

Java programs are made up a main class and any other classes needed to support the main()

We are going to **expand** our understanding of:

Classes and their definitions
- instance variables
- main() method
- Method creation
- **Method overloading**
- **Constructors**

Method Overloading

Method overloading in Java occurs when two or more methods shares same name and fulfill at least one of the following condition.

1) Have different number of arguments.
2) Have same number of arguments but their types are different.
3) Have both different numbers of arguments with a difference in their types.

How method overloading helps in Java

When it comes to, why to use method overloading and usage of method overloading. Two points comes out

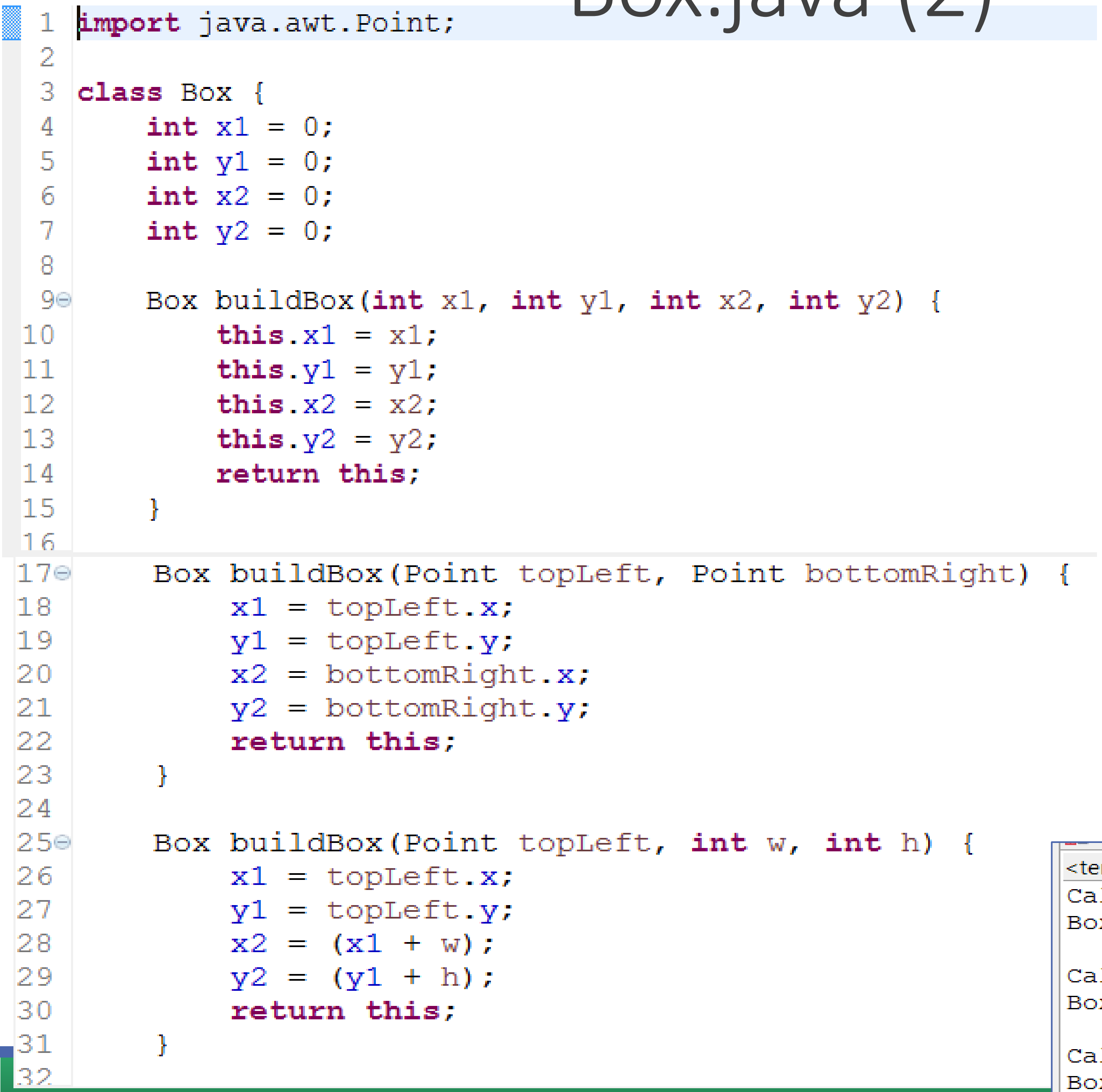
1) Helps in maintain consistency in method naming, doing same task with just a difference of input parameter.

2) Helps in reduce overhead of remember name of different functions for same task, one can pass possible different type of parameters or number of parameters to a single overloaded function to get the result.

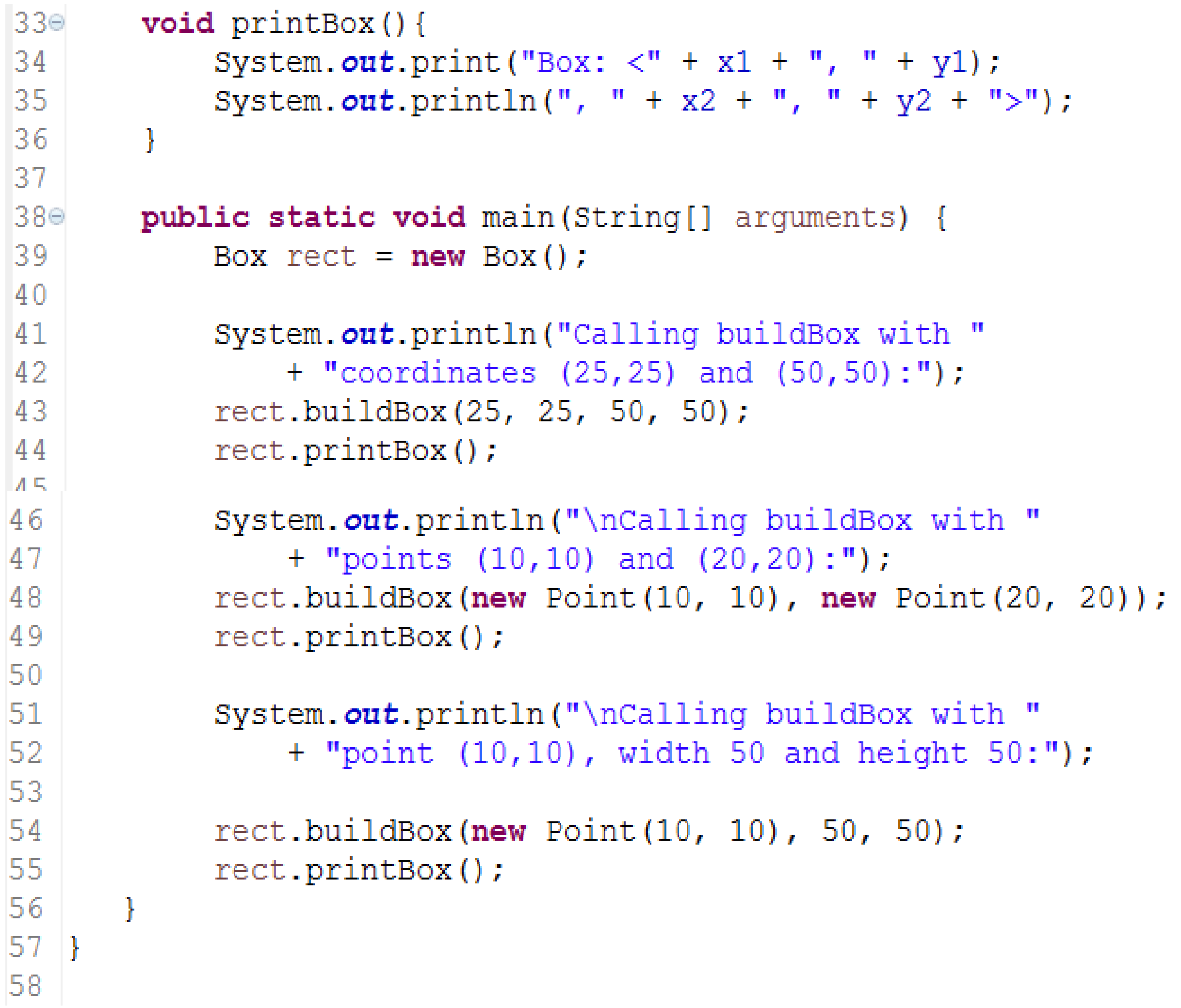
Exercise 1: Box.java – Page 106

Create an overloaded method beginning with a simple class called Box.java to define a rectangle with four instance variables to define the upper-left and lower-right bounds of the rectangle – (x1, y1) and (x2, y2)

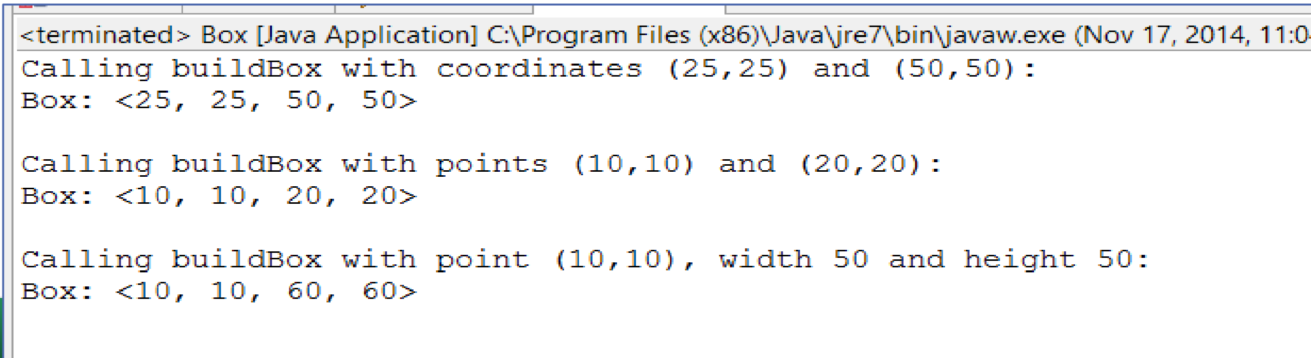
Box.java (2)



```java
import java.awt.Point;

class Box {
    int x1 = 0;
    int y1 = 0;
    int x2 = 0;
    int y2 = 0;

    Box buildBox(int x1, int y1, int x2, int y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
        return this;
    }

    Box buildBox(Point topLeft, Point bottomRight) {
        x1 = topLeft.x;
        y1 = topLeft.y;
        x2 = bottomRight.x;
        y2 = bottomRight.y;
        return this;
    }

    Box buildBox(Point topLeft, int w, int h) {
        x1 = topLeft.x;
        y1 = topLeft.y;
        x2 = (x1 + w);
        y2 = (y1 + h);
        return this;
    }

```



```java
    void printBox(){
        System.out.print("Box: <" + x1 + ", " + y1);
        System.out.println(", " + x2 + ", " + y2 + ">");
    }

    public static void main(String[] arguments) {
        Box rect = new Box();

        System.out.println("Calling buildBox with "
            + "coordinates (25,25) and (50,50):");
        rect.buildBox(25, 25, 50, 50);
        rect.printBox();

        System.out.println("\nCalling buildBox with "
            + "points (10,10) and (20,20):");
        rect.buildBox(new Point(10, 10), new Point(20, 20));
        rect.printBox();

        System.out.println("\nCalling buildBox with "
            + "point (10,10), width 50 and height 50:");

        rect.buildBox(new Point(10, 10), 50, 50);
        rect.printBox();
    }
}

```



```
<terminated> Box [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 17, 2014, 11:0
Calling buildBox with coordinates (25,25) and (50,50):
Box: <25, 25, 50, 50>

Calling buildBox with points (10,10) and (20,20):
Box: <10, 10, 20, 20>

Calling buildBox with point (10,10), width 50 and height 50:
Box: <10, 10, 60, 60>
```

Let’s Examine the Code

Lines 4 – 7: create a new instance of box, initialize instance variables

Lines 9 – 15 – buildBox() instance method sets the variables to their correct values (setter method)

- Since the arguments have the same name as the instance variables, we use keyword this inside of the method to refer to the method’s instance variables

Lines 17-23 & 25-31: Point object is used since x and y values are contained in the object. Overload buildBox() by changing its arguments to create alternate versions of buildBox() – line 1 imports the java.awt.Point class

Lines 33-36: printBox() to display the coordinates

Constructors

A constructor (although similar) is not a method

Creates an instance of a class (object)

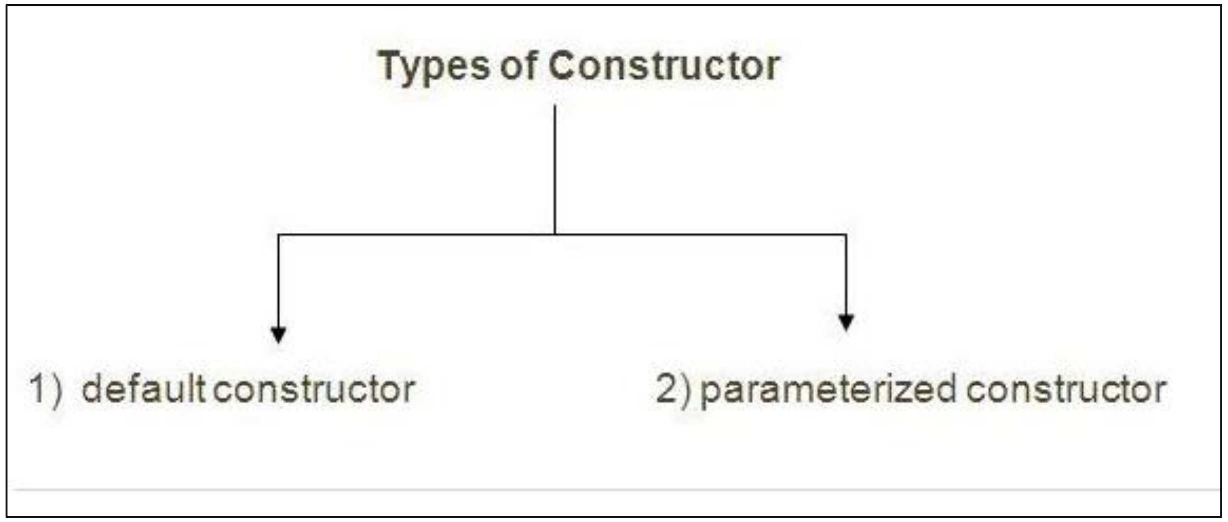
```
Platypus p1 = new Platypus();
```

Used to initialize objects – it constructs the values

- Provides data for the object

Two types

- Default – no parameters (arguments)
- Parameterized – has parameters (arguments)

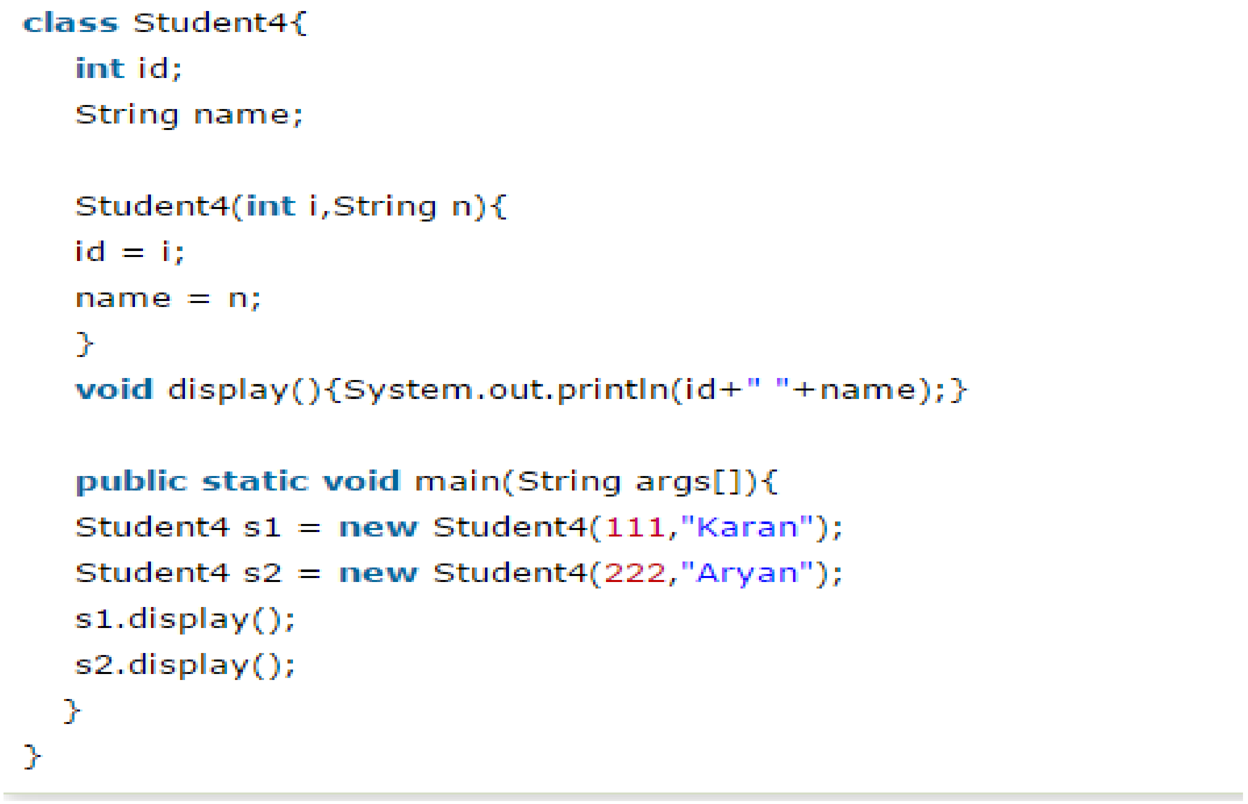


Constructors

DEFAULT

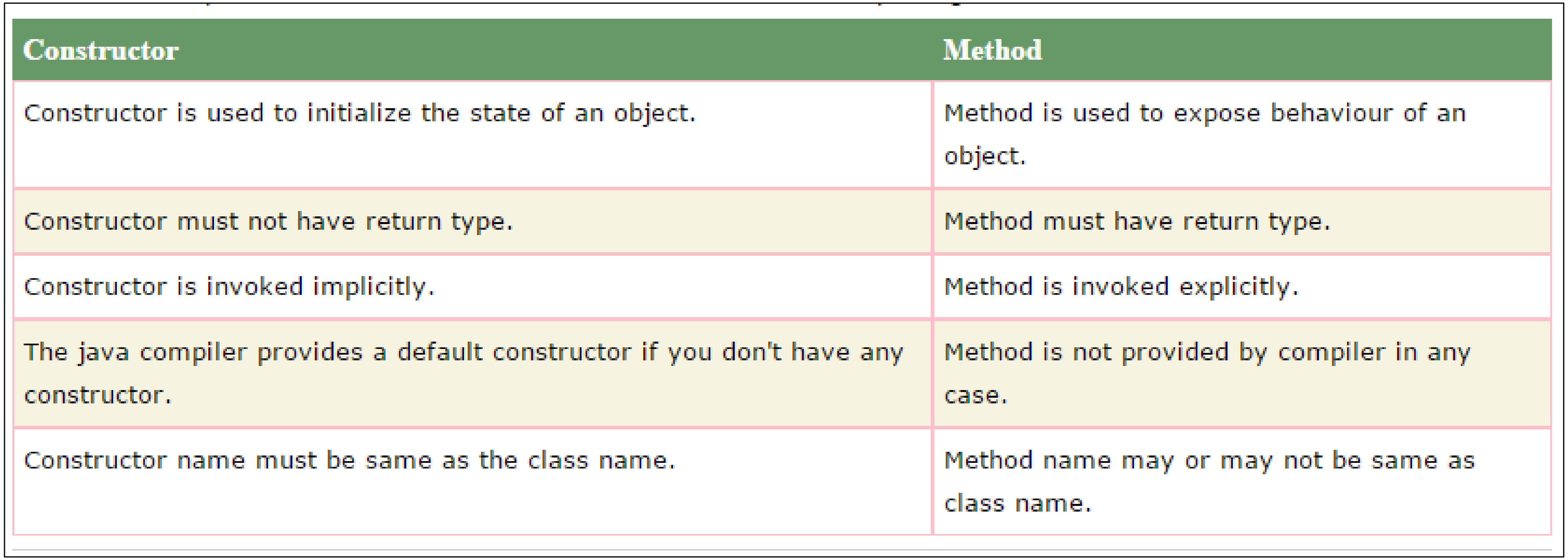
```java
class Student3{
int id;
String name;

void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
}
}
```

PARAMETERIZED



```java
class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

Difference Between Constructor and Method



Constructor	Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Chapter 6 - Java Packages, Interfaces and Advanced Class Development

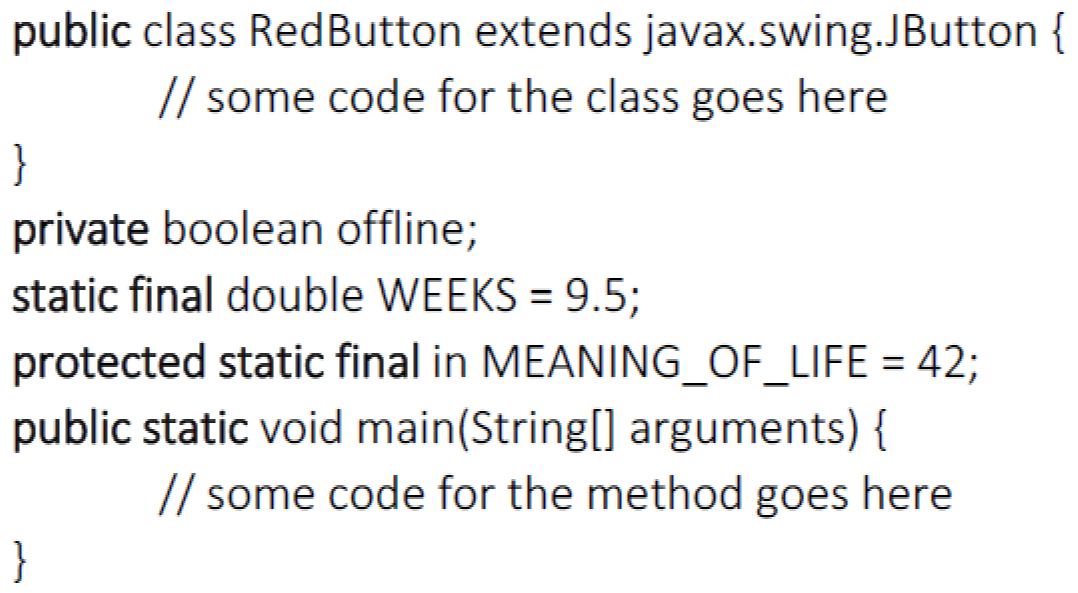
Modifiers – a set of keywords that are added to the class definition to change its meaning

- **protected, public, private** – used to control access to a class, method or variable
- **static** – used to crate class methods and variables – we've been using this one all along
- **final** – finalizes the implementation of classes, methods and variables
- **abstract** – creates abstract classes and methods
- **synchronized** and **volatile** – are modifiers for threads (more on this later)

Modifiers (2)

To use the modifier – include the keyword in the definition of the class

Modifiers are optional but can enhance/influence the functionality of the program



```
public class RedButton extends javax.swing.JButton {
            // some code for the class goes here
}
private boolean offline;
static final double WEEKS = 9.5;
protected static final in MEANING_OF_LIFE = 42;
public static void main(String[] arguments) {
            // some code for the method goes here
}
```

Access Control

Modifiers – public, private, protected and default

Control access to methods and variables

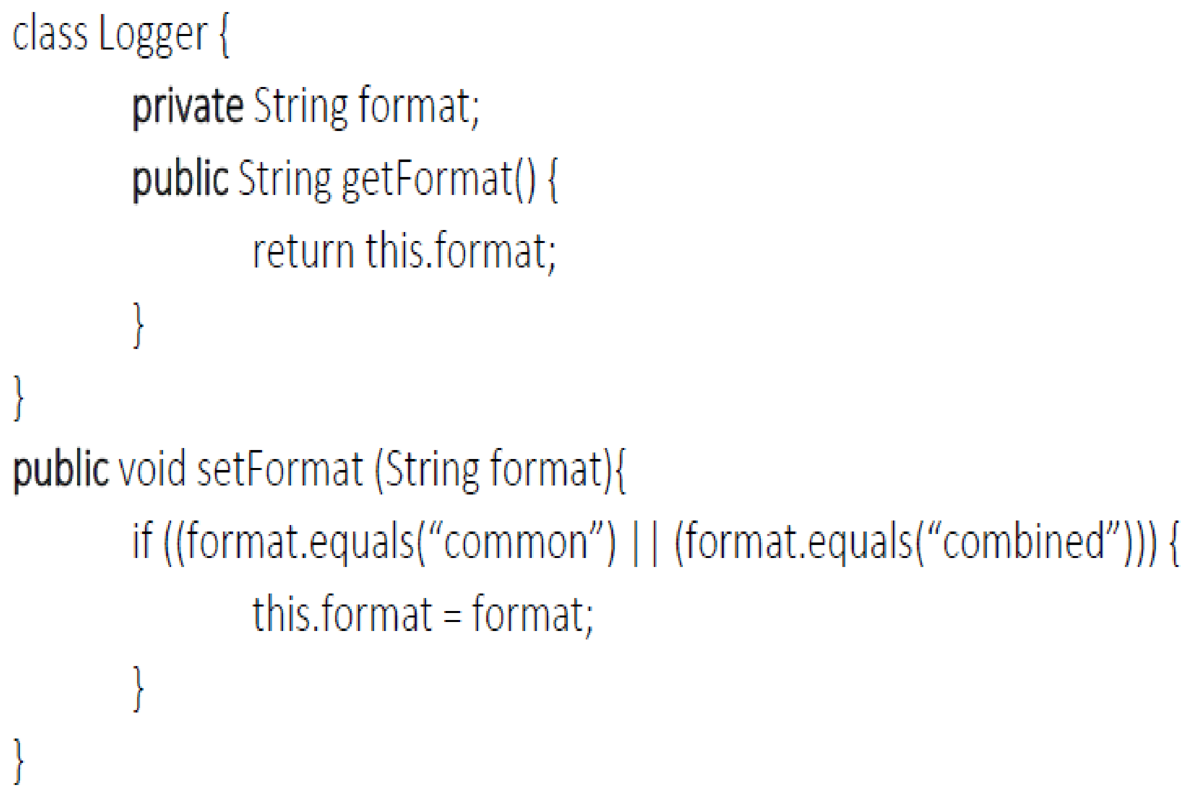
Determines which methods and variables are visible to other classes

Modifiers for AC

Default Access – no access control modifiers – available for read and change by any other class in the same package – not much access control in play

Private Access – completely hides a method or variable from outside to prevent usage

- Can be used only within its own class
- Not inherited by the subclass
- Variables of public within the class are available as defined
- In the example, the private is available thru getFormat and



```
class Logger {
        private String format;
        public String getFormat() {
                return this.format;
        }
}
public void setFormat (String format){
        if ((format.equals("common") | | (format.equals("combined"))) {
                this.format = format;
        }
}
```

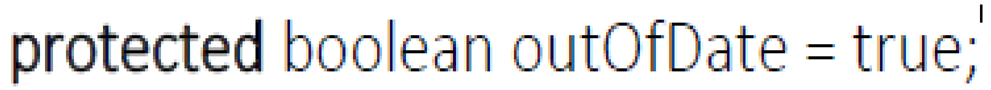
Modifiers for AC (2)

Public Access – make a method or variable completely available

- E.g. Color in java.awt have public variables for the color red or blue
- Ability to not limit common variables – there is no benefit in doing so
- Everything we’ve coded so far has been public
- Inherited by its subclasses

Protected Access – Limits use to only the subclasses of a class or other classes in the same package

- Don’t have to be in the same package
- Gives access to subclass but prevents an unrelated class from trying to use it



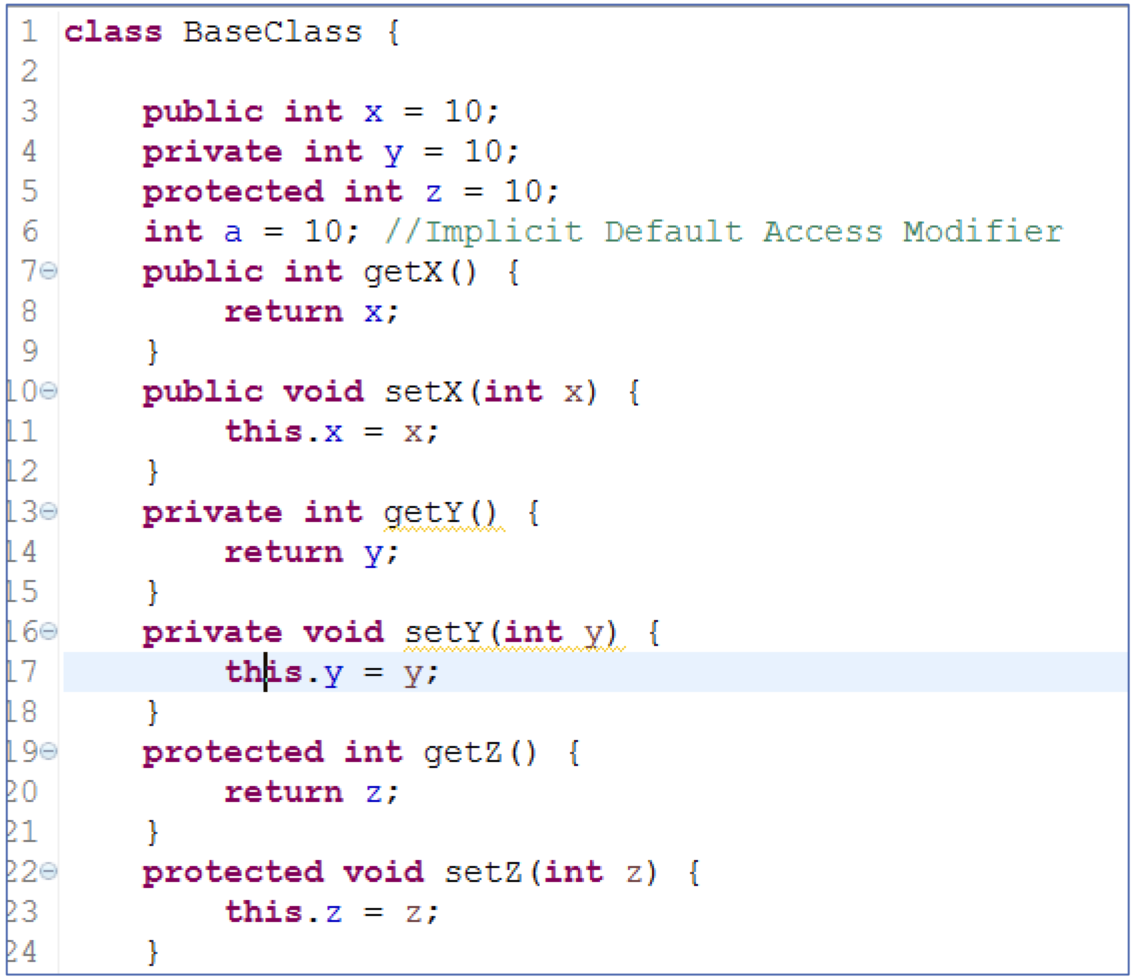
```
protected boolean outOfDate = true;
```

In this exercise we demonstrate the usage of an Abstract Class while exercising all modifiers

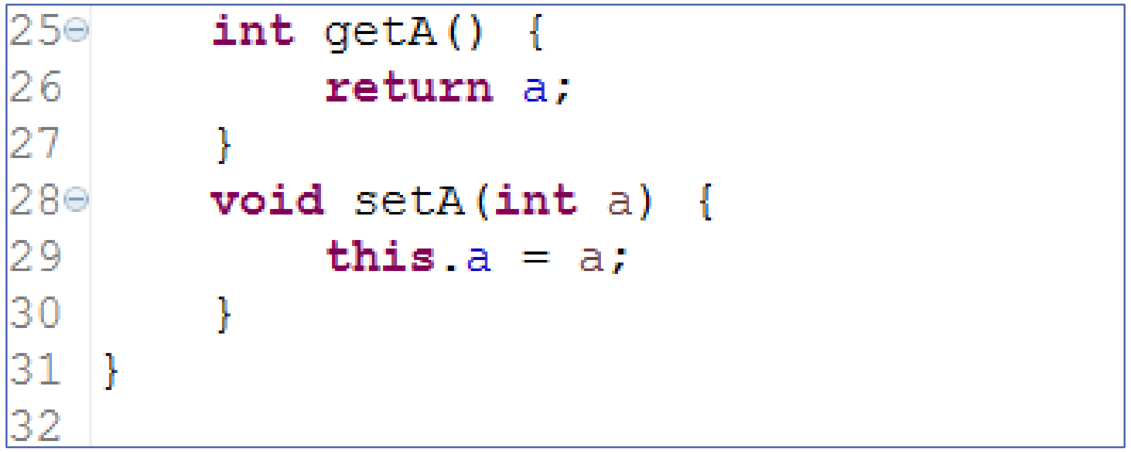
Exercise: Modifiers

Run as Java Application

BaseClass.java

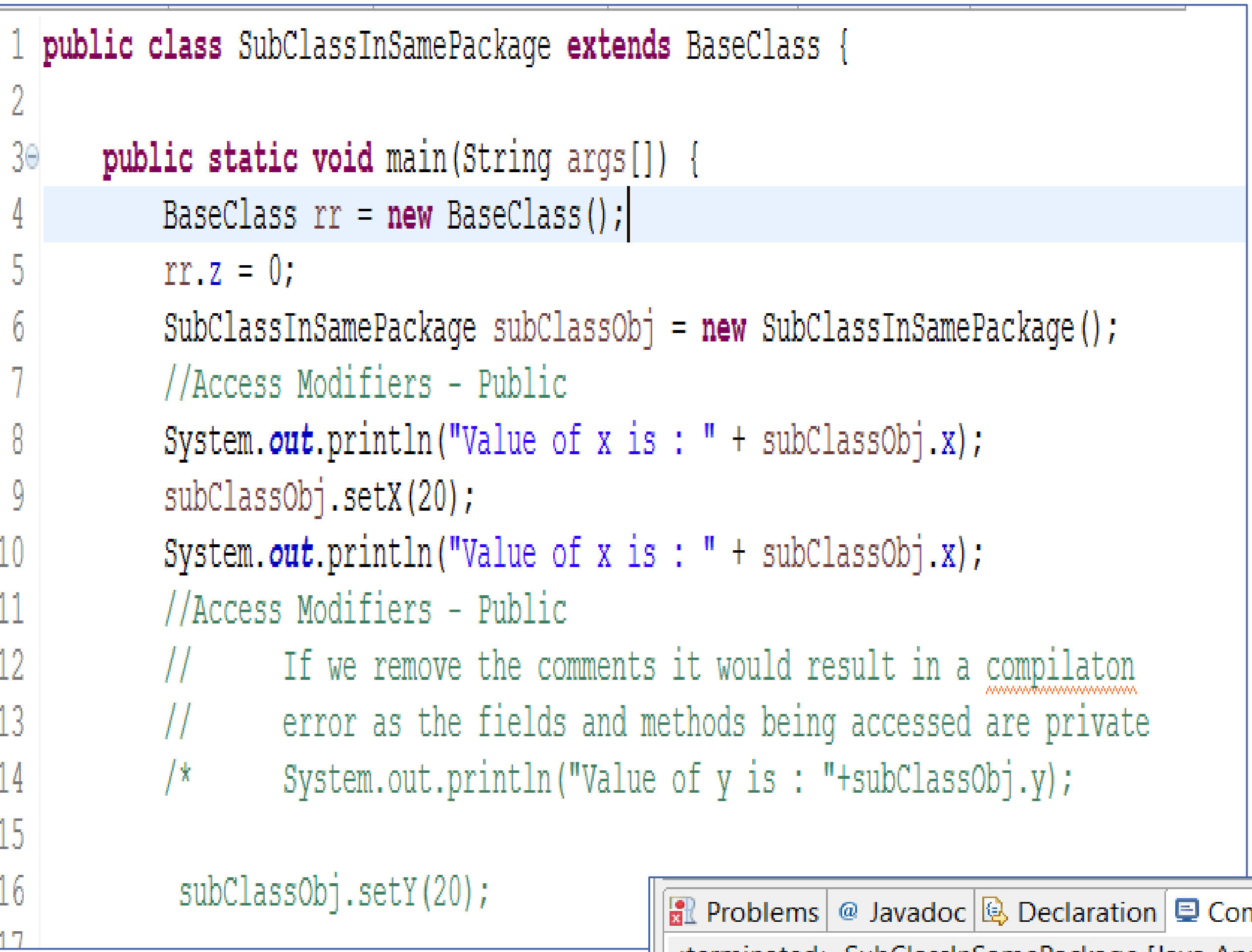


```
class BaseClass {

    public int x = 10;
    private int y = 10;
    protected int z = 10;
    int a = 10; //Implicit Default Access Modifier
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    private int getY() {
        return y;
    }
    private void setY(int y) {
        this.y = y;
    }
    protected int getZ() {
        return z;
    }
    protected void setZ(int z) {
        this.z = z;
    }
```

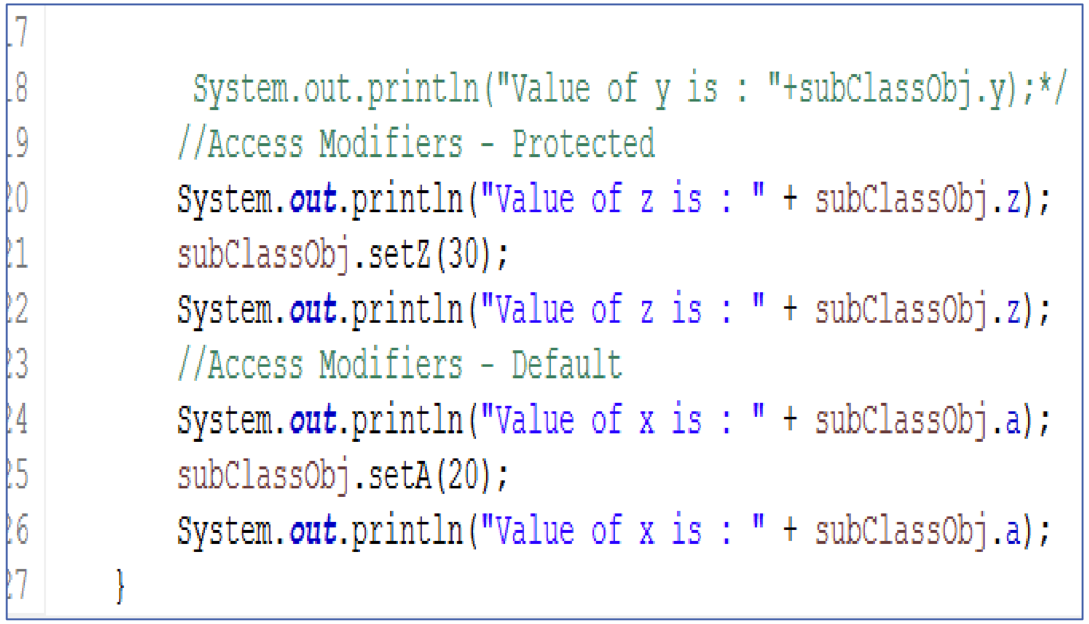


```
    int getA() {
        return a;
    }
    void setA(int a) {
        this.a = a;
    }
}

```

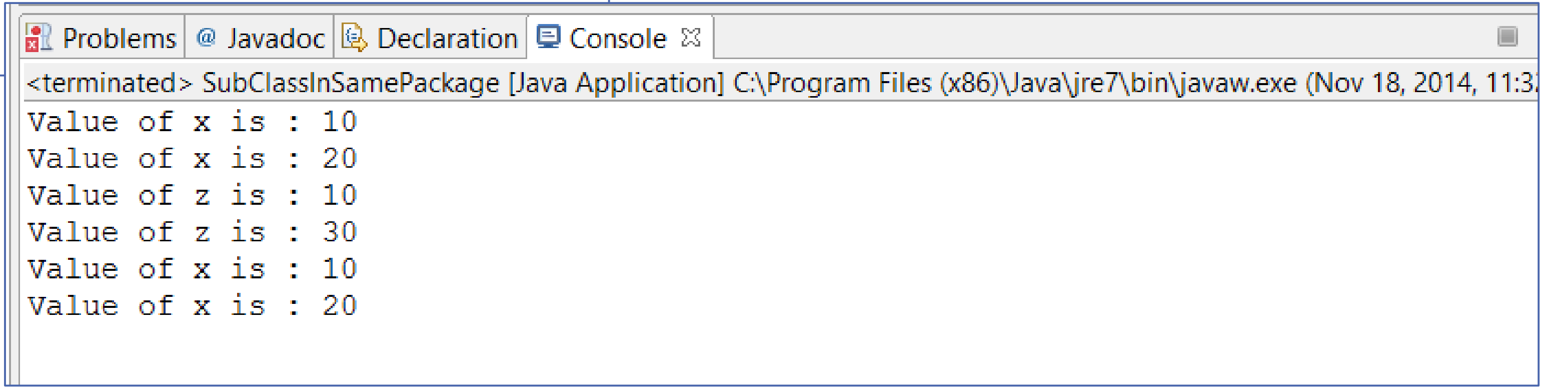
Exercise Modifiers – SubClassInSamePackage.java



```java
public class SubClassInSamePackage extends BaseClass {

    public static void main(String args[]) {
        BaseClass rr = new BaseClass();
        rr.z = 0;
        SubClassInSamePackage subClassObj = new SubClassInSamePackage();
        //Access Modifiers - Public
        System.out.println("Value of x is : " + subClassObj.x);
        subClassObj.setX(20);
        System.out.println("Value of x is : " + subClassObj.x);
        //Access Modifiers - Public
        //      If we remove the comments it would result in a compilaton
        //      error as the fields and methods being accessed are private
        /*      System.out.println("Value of y is : "+subClassObj.y);

         subClassObj.setY(20);
```



```java

         System.out.println("Value of y is : "+subClassObj.y);*/
        //Access Modifiers - Protected
        System.out.println("Value of z is : " + subClassObj.z);
        subClassObj.setZ(30);
        System.out.println("Value of z is : " + subClassObj.z);
        //Access Modifiers - Default
        System.out.println("Value of x is : " + subClassObj.a);
        subClassObj.setA(20);
        System.out.println("Value of x is : " + subClassObj.a);
    }
```

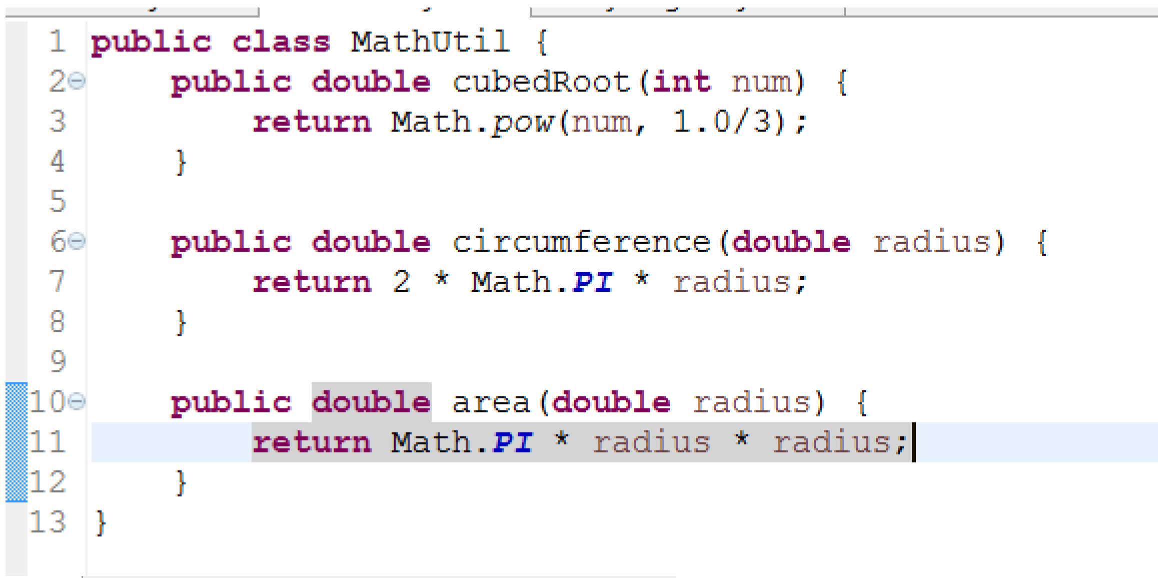


```
<terminated> SubClassInSamePackage [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 18, 2014, 11:3
Value of x is : 10
Value of x is : 20
Value of z is : 10
Value of z is : 30
Value of x is : 10
Value of x is : 20
```

Exercise: Public Modifier

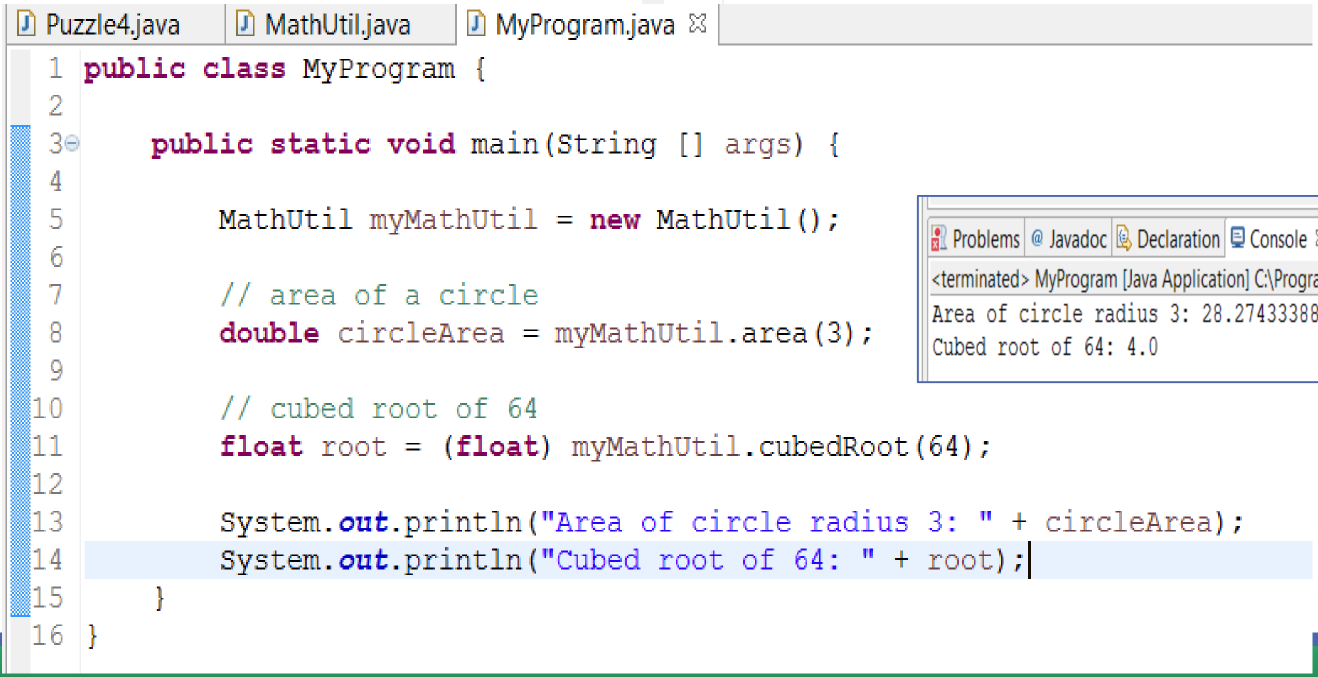
Create two classes:

MathUtil.java -- accept values from the calling program then compute the cubedRoot(), circumference() or the area().



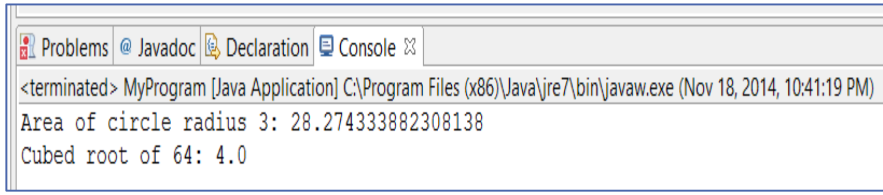
```java
public class MathUtil {
    public double cubedRoot(int num) {
        return Math.pow(num, 1.0/3);
    }

    public double circumference(double radius) {
        return 2 * Math.PI * radius;
    }

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

MyProgram.java – test the methods of the MathUtil class.



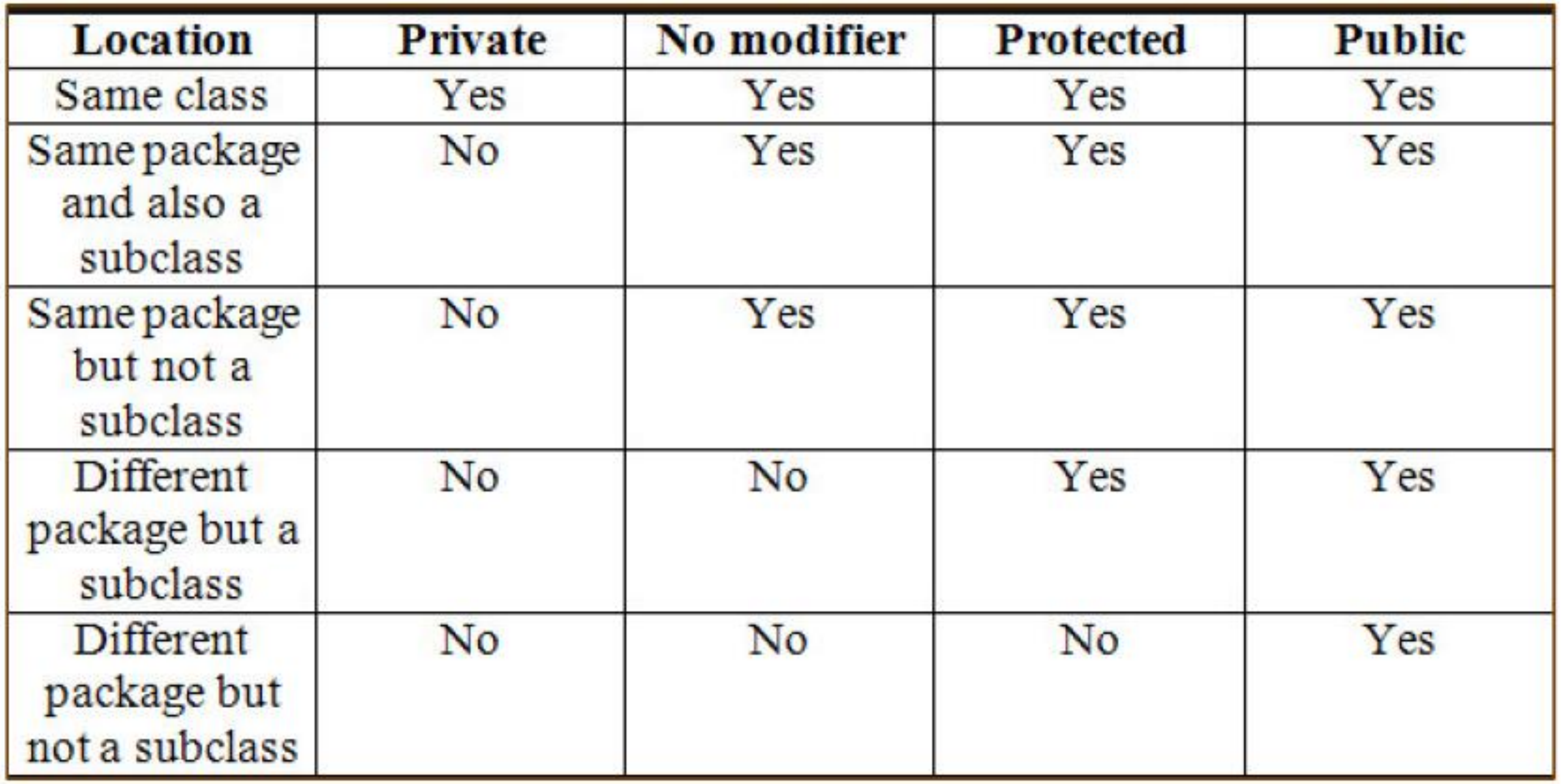
```java
public class MyProgram {

    public static void main(String [] args) {

        MathUtil myMathUtil = new MathUtil();

        // area of a circle
        double circleArea = myMathUtil.area(3);

        // cubed root of 64
        float root = (float) myMathUtil.cubedRoot(64);

        System.out.println("Area of circle radius 3: " + circleArea);
        System.out.println("Cubed root of 64: " + root);
    }
}
```

Run as a Java Application



```
<terminated> MyProgram [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 18, 2014, 10:41:19 PM)
Area of circle radius 3: 28.274333882308138
Cubed root of 64: 4.0
```

Comparing Access Control Methods



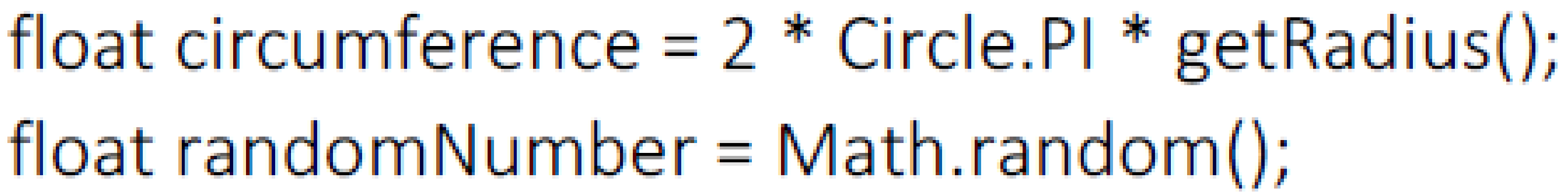
Location	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package and also a subclass	No	Yes	Yes	Yes
Same package but not a subclass	No	Yes	Yes	Yes
Different package but a subclass	No	No	Yes	Yes
Different package but not a subclass	No	No	No	Yes

Static Variables and Methods

Static – create **class methods** and variables

Class variables and methods are accessed using the class by dot notation -- Color.red; or Circle.PI;

Here are a few examples:



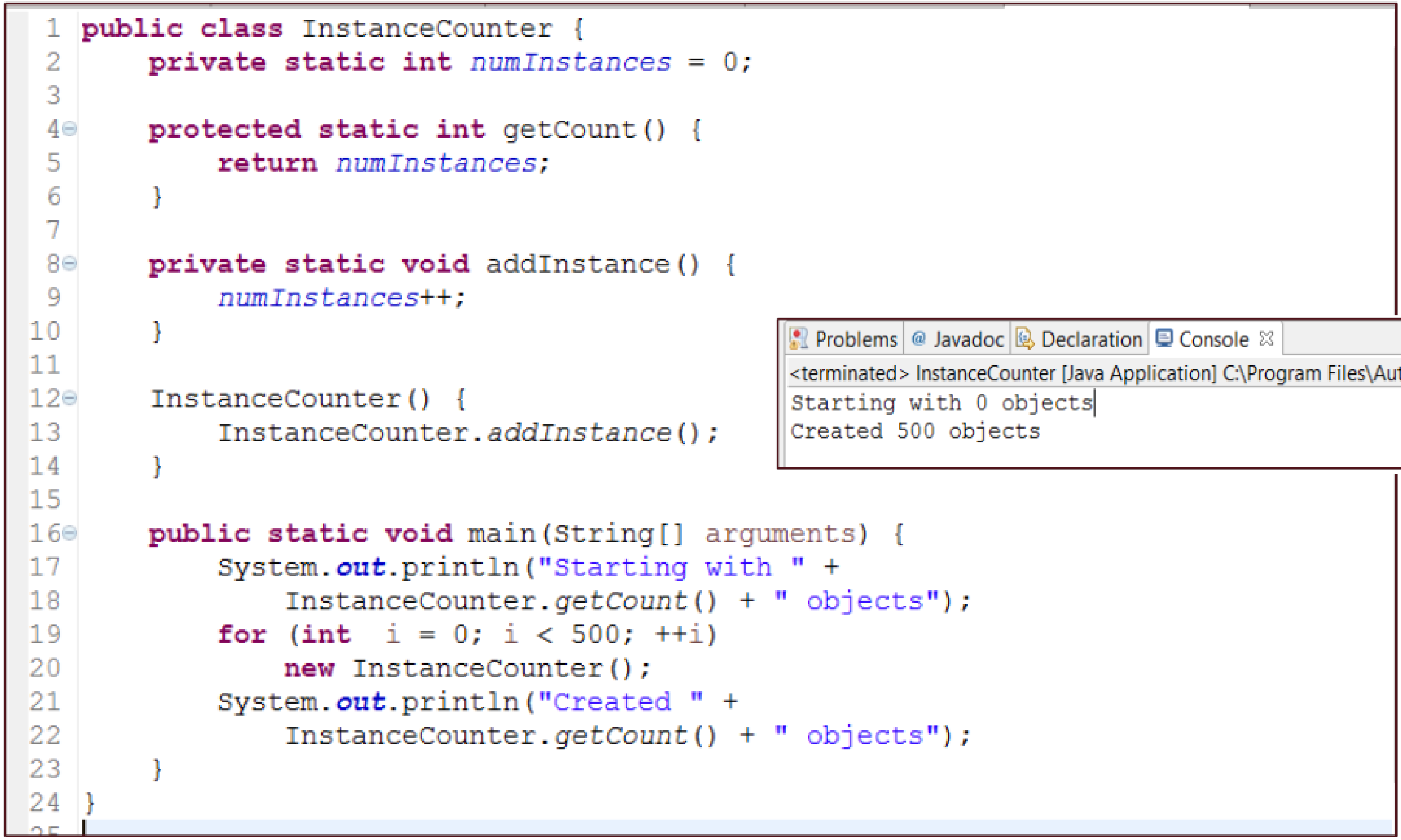
```
float circumference = 2 * Circle.PI * getRadius();
float randomNumber = Math.random();
```

Exercise 1: InstanceCounter.java – Pg 126

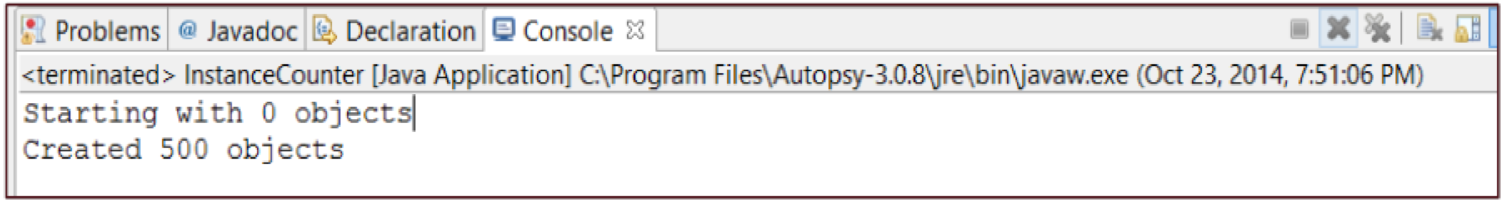
Create a class called InstanceCounter.java that uses class and instance variables to keep track of how many objects of that class that can be created

Run as a Java application

InstanceCounter.java



```java
public class InstanceCounter {
    private static int numInstances = 0;

    protected static int getCount() {
        return numInstances;
    }

    private static void addInstance() {
        numInstances++;
    }

    InstanceCounter() {
        InstanceCounter.addInstance();
    }

    public static void main(String[] arguments) {
        System.out.println("Starting with " +
            InstanceCounter.getCount() + " objects");
        for (int  i = 0; i < 500; ++i)
            new InstanceCounter();
        System.out.println("Created " +
            InstanceCounter.getCount() + " objects");
    }
}
```



```
<terminated> InstanceCounter [Java Application] C:\Program Files\Autopsy-3.0.8\jre\bin\javaw.exe (Oct 23, 2014, 7:51:06 PM)
Starting with 0 objects
Created 500 objects
```

Final Classes and Methods

final – used with classes, methods and variables to indicate they will never change

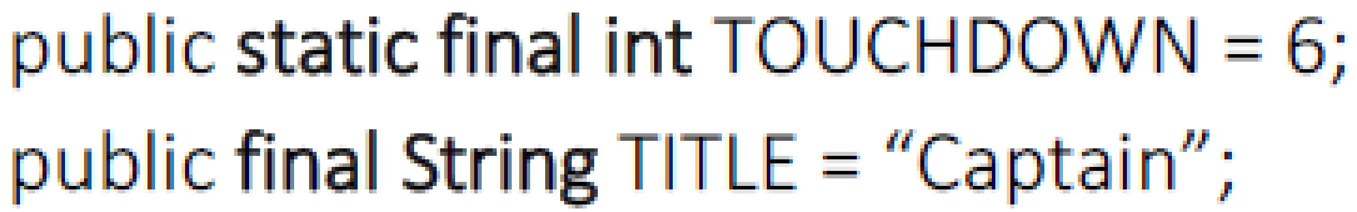
The meaning of final is driven by what is being created

- final class cannot be a subclass
- final method cannot be overridden by any subclass
- final variable cannot change its value

final Variables

Constants

Often used with static to make it a class variable (class variables are written in all caps)



```
public static final int TOUCHDOWN = 6;
public final String TITLE = “Captain”;
```

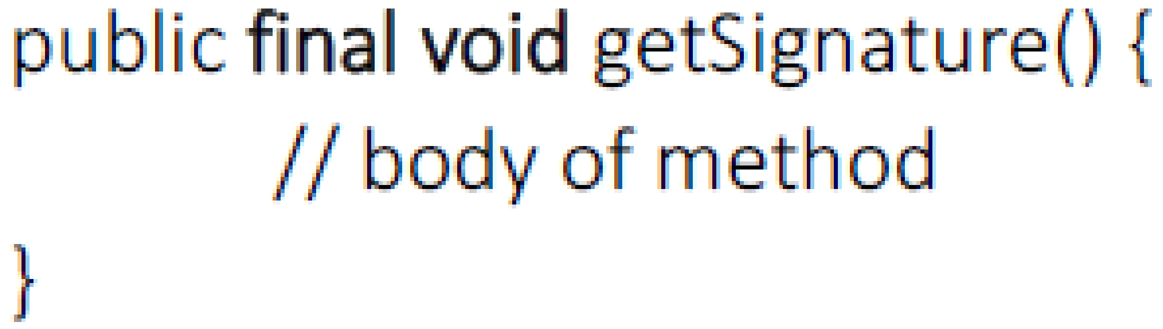
final Methods

Cannot override in a subclass

Declared final in the class declarations

Usually declared as final for performance reasons

Private methods are always final by default



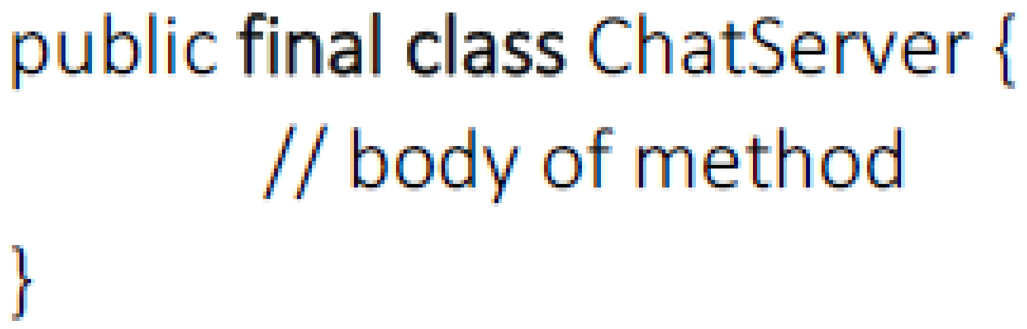
```
public final void getSignature() {
        // body of method
}
```

final Classes

Cannot be a subclassed by another class – if need a class that behaves like a String (for example) we must build a new class from the beginning

Improves performance

Methods in a final class are automatically final themselves



```
public final class ChatServer {
        // body of method
}
```

Abstract Classes and Methods

Abstract Class – A class that never needs to be instantiated directly

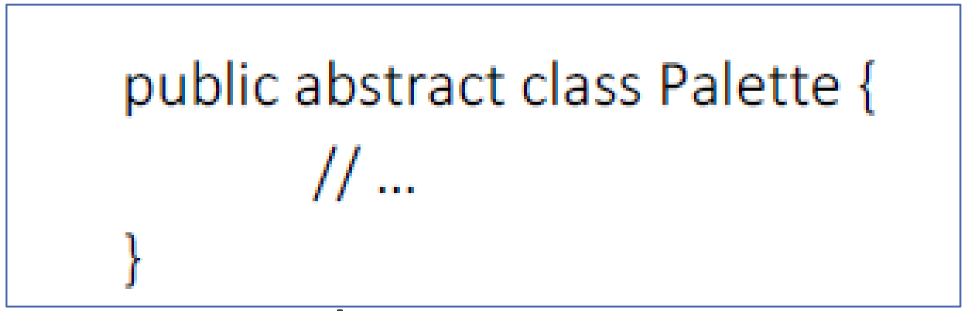
Holds common behavior and attributes that are shared by all subclasses

Uses the abstract modifier

Can contain anything a normal class can

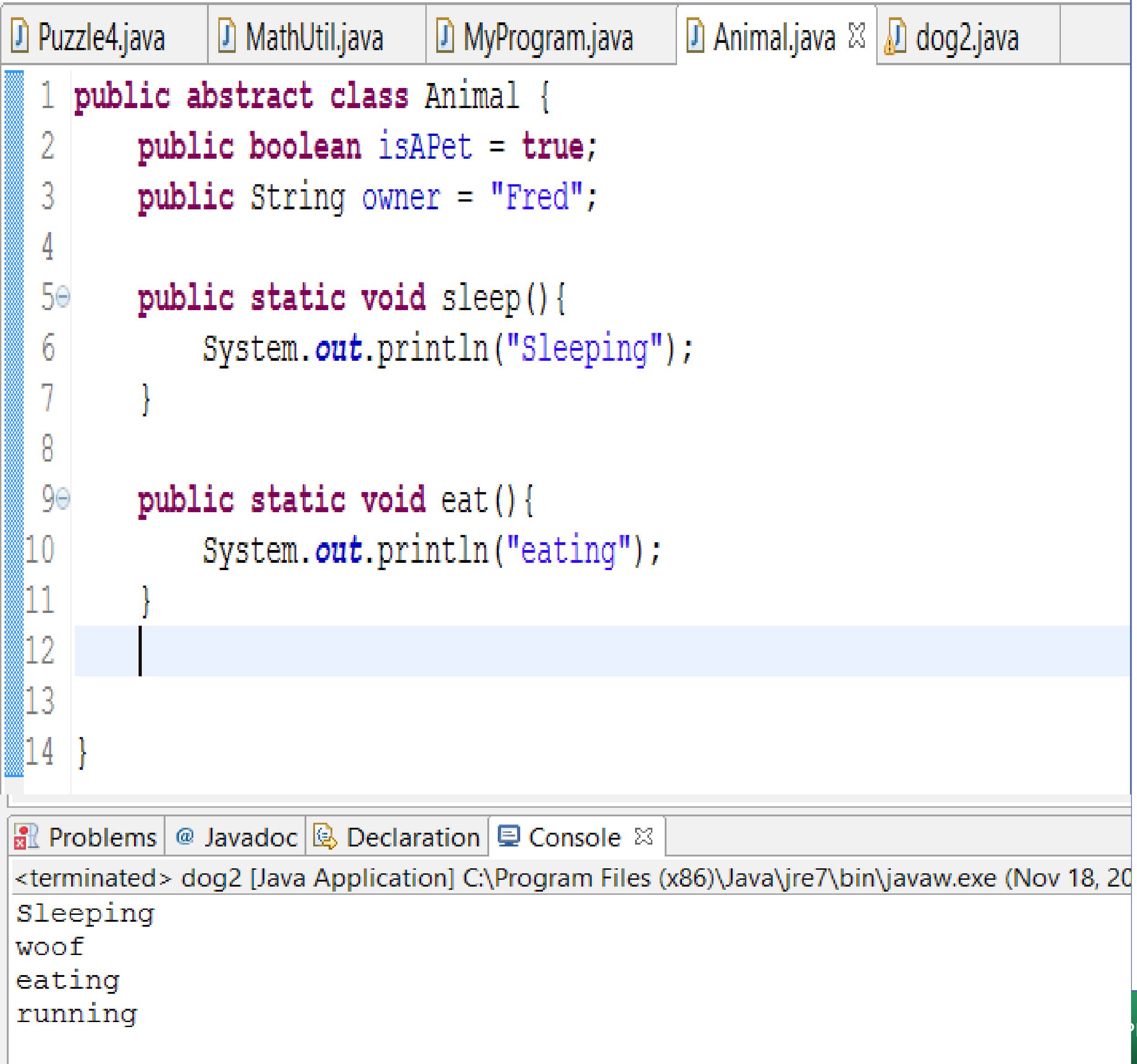
Can contain abstract methods (method signatures with no implementation)

Cannot declare an abstract method unless the class is too abstract



```
public abstract class Palette {
        // ...
}
```

Exercise: Abstract Class



Puzzle4.java | MathUtil.java | MyProgram.java | Animal.java | dog2.java

```java
public abstract class Animal {
    public boolean isAPet = true;
    public String owner = "Fred";

    public static void sleep(){
        System.out.println("Sleeping");
    }

    public static void eat(){
        System.out.println("eating");
    }


}
```

Problems | Javadoc | Declaration | Console

```
<terminated> dog2 [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 18, 20
Sleeping
woof
eating
running
```

```java
public class dog2 extends Animal {
private int numberOfLegs;
private boolean hasOwner;

public dog2() {
numberOfLegs = 4;
hasOwner = false;
}

private static void bark() {
System.out.println("woof");
}

public static void move() {
System.out.println("running");
}

public static void main(String [] args){
Animal.sleep();
bark();
Animal.eat();
move();
}
}
```

Import Statement

The **import statement** in **Java** allows to refer to classes which are declared in other **packages** to be accessed without referring to the full package name.

You **do not** need any **import statement** if you are willing to always refer to java.util.List by its full name, and so on for all other classes

Frequently Java Used Packages

java.lang
java.util
java.io
java.util.regex
java.text
java.sql
java.net
java.util.concurrent
java.util.concurrent.locks
java.util.concurrent.atomic
javax.servlet
javax.servlet.http
java.lang.reflect
java.nio
java.nio.channels
java.nio.charset
javax.swing
java.awt
java.awt.event
org.xml.sax
org.w3c.dom
javax.xml.parsers
javax.naming

Java.lang

java.lang

Packages

Previous lessons

- We’ve created classes in the same file as the main or in its own file
- We had to remember which file the class was in
- We had to have the same name as the class – the main class of the file
- Most of what we’ve done has been marked as public

Packages(2)

An organization of a group of classes

Related in purpose, scope or inheritance

Reduces problems with naming collisions

Enables the protection of variables, methods and classes on a larger scale

Packages are imported via the import statement or using the fully qualified name of a class

Packages (3)

Three techniques to accessing the classes of a package

- If the class is in the package java.lang (System or Date), we can use the class name to refer to that class. java.lang is automatically available in all programs
- If the class is in some other package, not java.lang, we refer to that class by its fully qualified name as in java.awt.Font
- If we use classes from a particular package frequently, we can import the individual classes or the entire package then we can refer to the needed class by its class name

Creating Packages

Must have one or more classes or interfaces (more on interfaces in a bit)

- Cannot be empty

Source files of classes or interfaces must reside in the same directory as the package

Name of the package must be unique

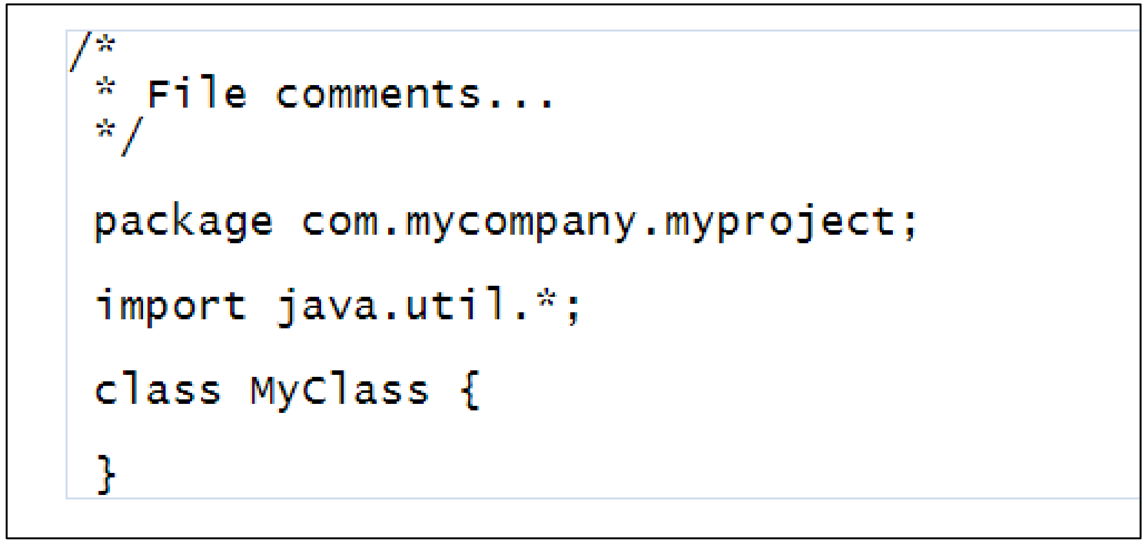
Declaring Package Members

Decide which classes and interface should belong in the package

Specify the package declaration in the source file of each class and interface that is part of the package

- Use the syntax: keyword package followed by the package name

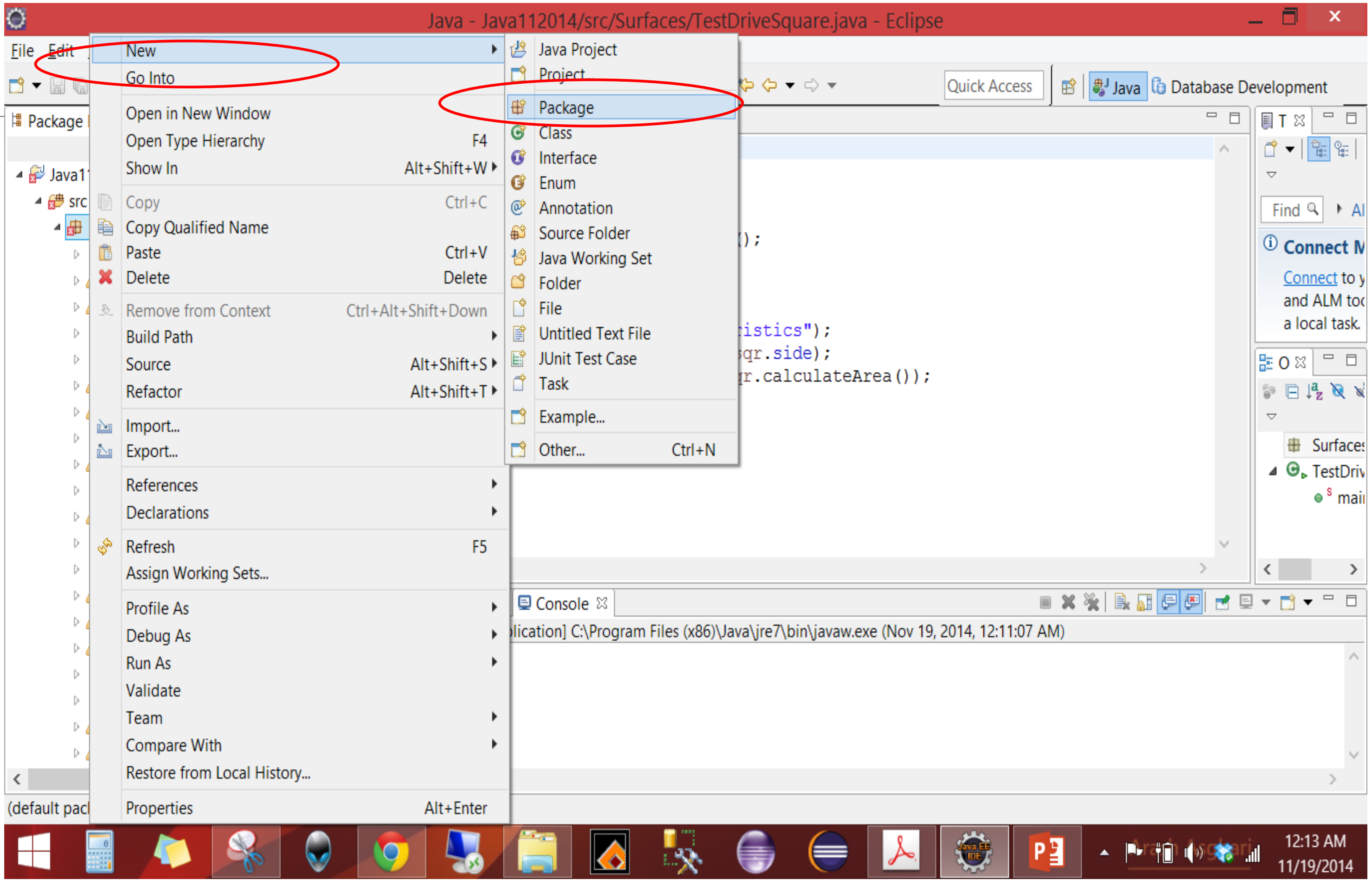
  package com.appinc-4me.myproject;



```
/*
 * File comments...
 */

package com.mycompany.myproject;

import java.util.*;

class MyClass {

}
```

Exercise: Create and Use a Custom Package

1. Create a Package
   - Right click on src
   - Select new
   - Select package

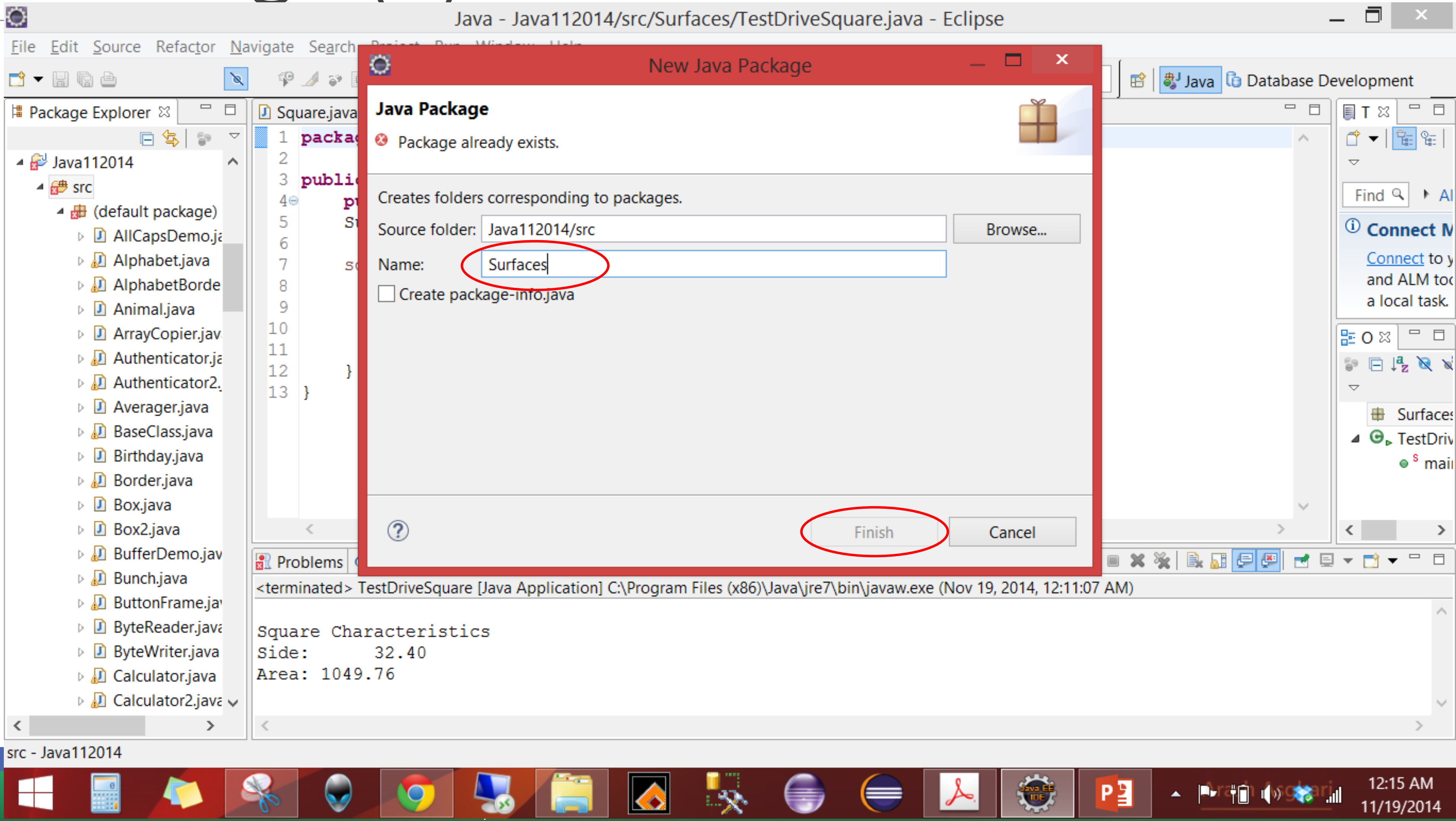


```
);
istics");
sqr.side);
r.calculateArea());
```

```
lication] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 19, 2014, 12:11:07 AM)
```

Create a Package (2)

2. Give it a name
   - Name the package Surfaces
   - Click Finish



Java - Java112014/src/Surfaces/TestDriveSquare.java - Eclipse

New Java Package

Java Package

Package already exists.

Creates folders corresponding to packages.

Source folder: Java112014/src

Name: Surfaces

Create package-info.java

```
<terminated> TestDriveSquare [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 19, 2014, 12:11:07 AM)

Square Characteristics
Side:       32.40
Area:  1049.76
```

Using Packages

Can be used by many programmers

There are three ways to use a package

- Inline declarations
- Importing only the package members
- Importing the entire package

Packages – Inline declarations

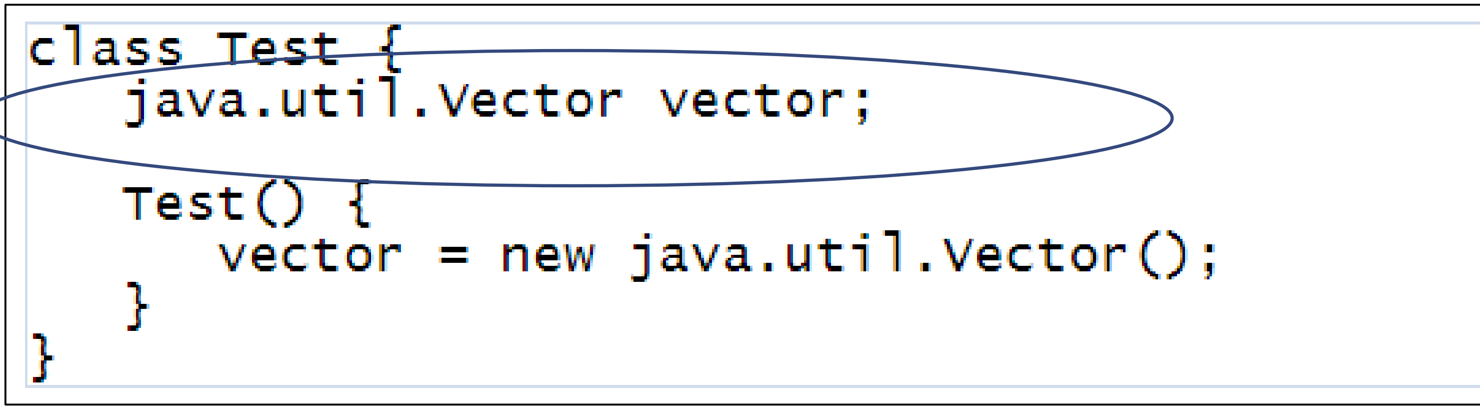
Simply declare the package members we wish to use with its fully qualified name

Using the Vector class of java.util

java.util.Vector vector;

Good when we only want to use a single member a few times

We initialize the object by calling its constructor using the fully qualified package name as shown:



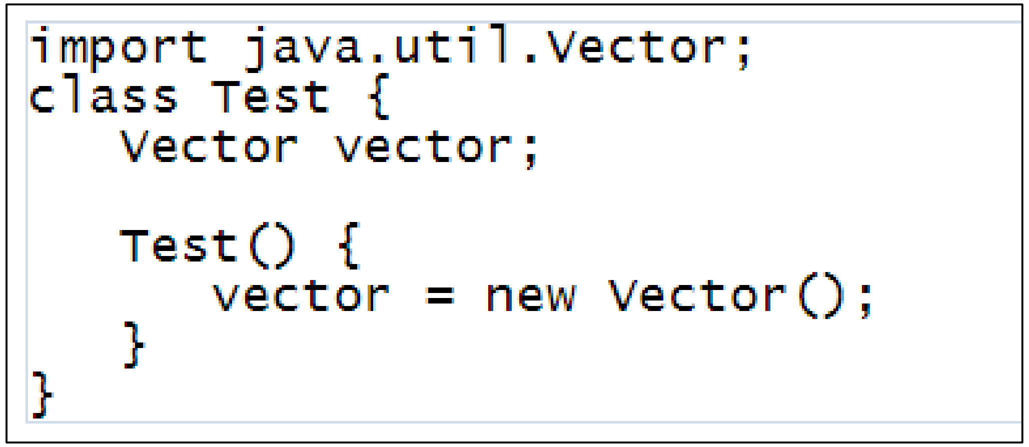
```
class Test {
   java.util.Vector vector;

   Test() {
      vector = new java.util.Vector();
   }
}
```

Packages – Importing a Single Package Member

Used when we need to use the package and its members many times during our code

Once declared, can use its members many times without the fully qualified name

Use the import statement followed by the fully qualified name of the member

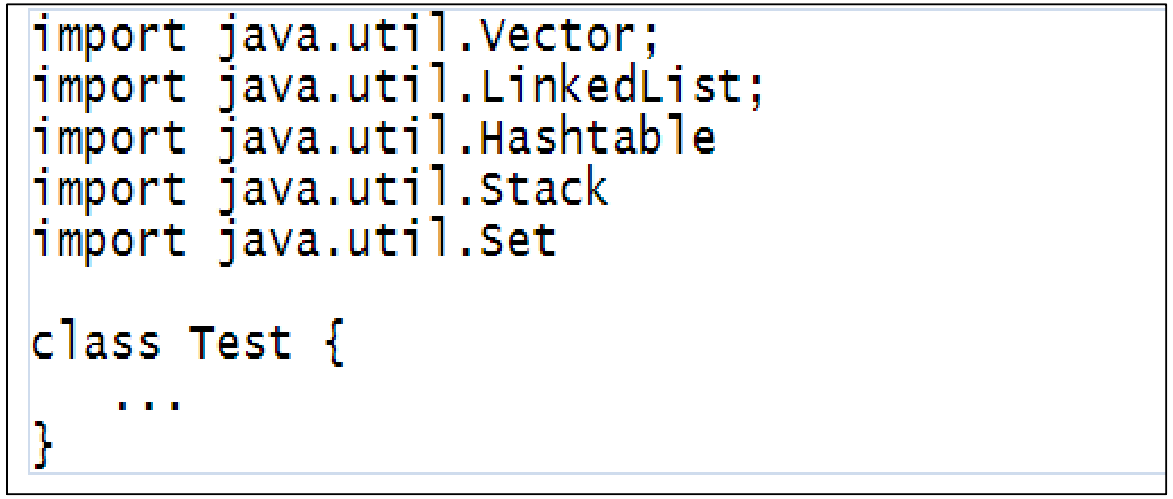


```
import java.util.Vector;
class Test {
   Vector vector;

   Test() {
      vector = new Vector();
   }
}
```

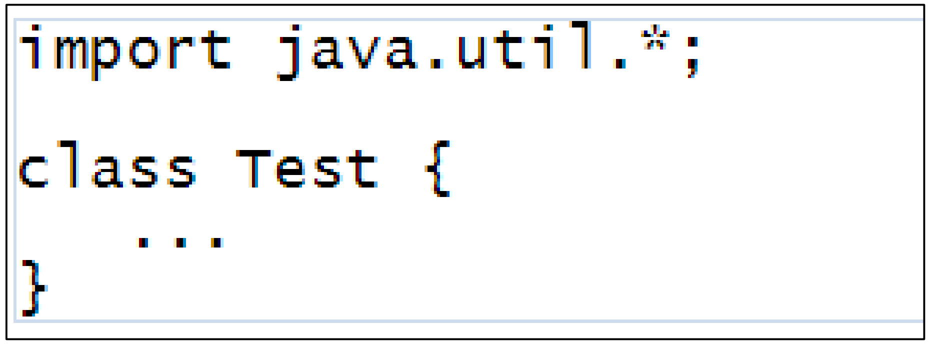
Packages – Importing the Entire Package

Used when we need a number of members from a package

Can replace the following with a single statement by using the Wild Card *



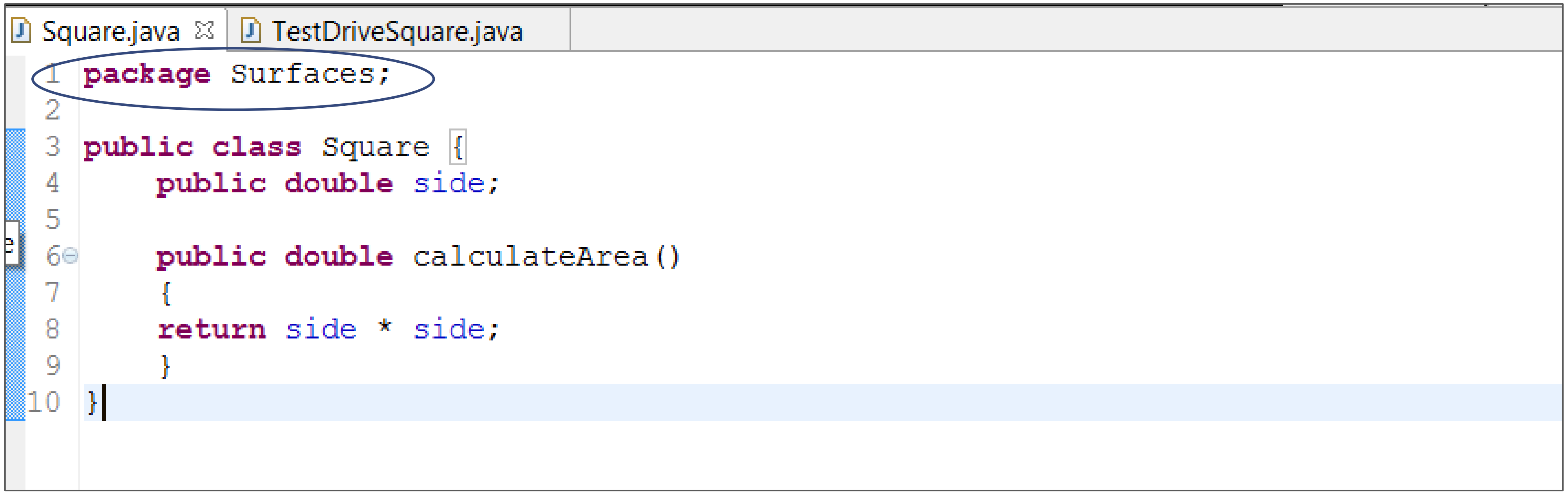
```
import java.util.Vector;
import java.util.LinkedList;
import java.util.Hashtable
import java.util.Stack
import java.util.Set

class Test {
   ...
}
```



```
import java.util.*;

class Test {
    ...
}
```

Create a class called Square.java in the Surfaces package

Under the package Surfaces, create the class called Square.java

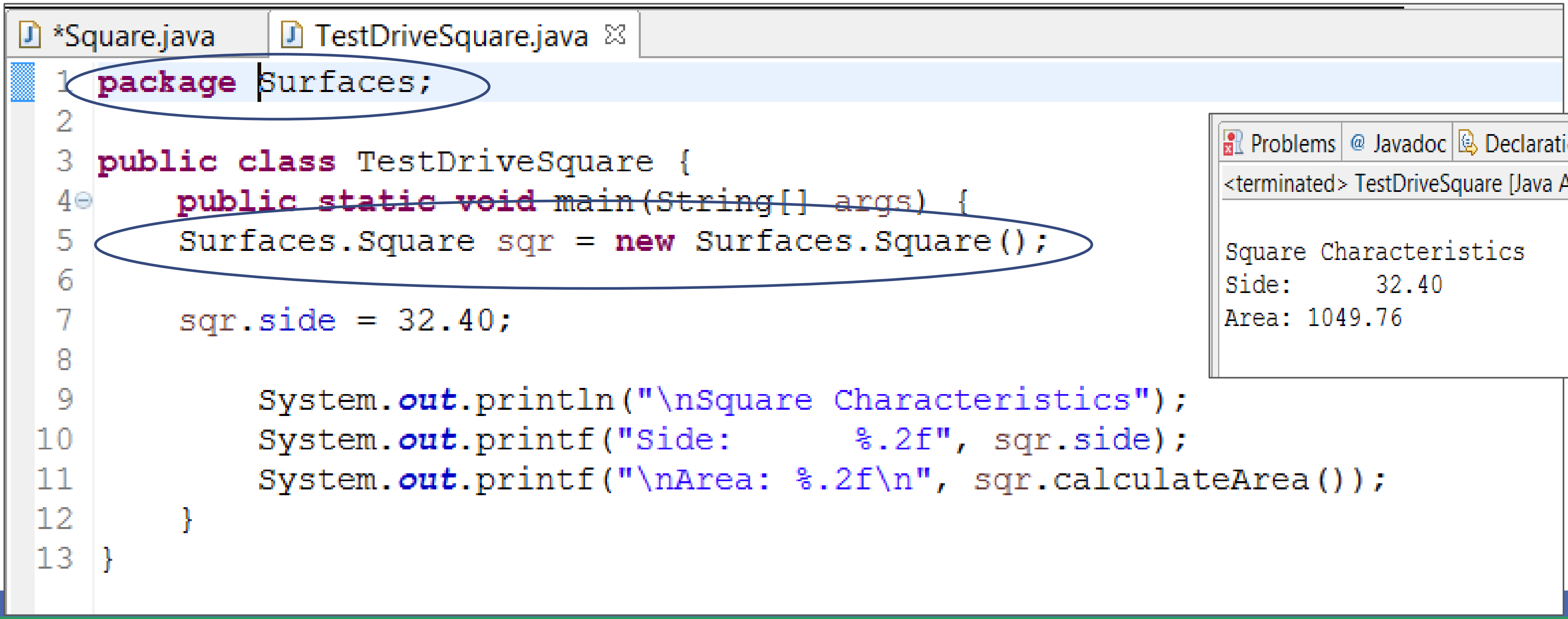


```java
package Surfaces;

public class Square {
    public double side;

    public double calculateArea()
    {
    return side * side;
    }
}
```

Use the Square.java in the TestDriveSquare.java class

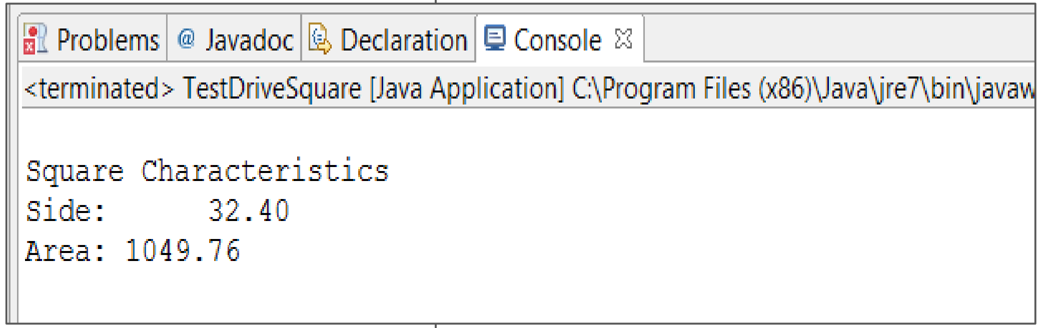
Create the tester class for Square.java under the package Surfaces

Run as a Java application



*Square.java | TestDriveSquare.java

```
package Surfaces;

public class TestDriveSquare {
    ~~public static void main(String[] args) {~~
    Surfaces.Square sqr = new Surfaces.Square();

    sqr.side = 32.40;

        System.out.println("\nSquare Characteristics");
        System.out.printf("Side:      %.2f", sqr.side);
        System.out.printf("\nArea: %.2f\n", sqr.calculateArea());
    }
}
```



```
<terminated> TestDriveSquare [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw

Square Characteristics
Side:      32.40
Area: 1049.76
```

Packages – Best Practices

Package by Feature

- Organized according to the feature set – place all items related to a single feature (and only that feature) into a single package
- Placing items that work together in the same package
- Names correspond to important, high-level aspects of the problem domain: com.app.doctor, com.app.patient, etc.
- Each package contains only the items related to that package feature: Doctor.java, DoctorAction.java

Package – Best Practices (2)

Highest level packages reflect various abstraction layers

com.app.action, com.app.model, etc.

Implementation over multiple directories

- Can be considered categories

Most use Package by Feature

- Higher modularity
- Easier code navigation
- Higher level of abstraction
- Separates both features and layers
- Minimizes scope
- Better growth style

Design Checklist

Favor placing API (Application Program Interfaces) and implementation into separate packages

- API in higher level packages
- Implementation into lower level packages

Break large APIs into several packages