Basics of the Object-Oriented Programming
Classes and Objects

Associate Professor Viorica Rozina Chifu
viorica.chifu@cs.utcluj.ro

Class in Java - Constructors

- Constructors are special methods in a class that are used to create new objects of that class.
- Constructors are invoked automatically when an object is created from the class using the "new" keyword.
- Constructor declarations have the same syntax as method declarations, but they have the same name as the class and do not have a return type.
- This distinguishes them from regular methods that may have different names and return types.

Class in Java - Constructors

- Example of constructor

```
public class Bicycle
{
    private int speed;
    public Bicycle(int startSpeed)
     {
      speed = startSpeed;
     }
    public void getSpeed(){
       return speed;
}
 … }
```

- How we create a **new** *Bicycle* object (i.e. *myBike*)?
  - Call the constructor by means of the **new** operator:

    Bicycle myBike = **new** *Bicycle*(30);
  - **new** *Bicycle*(30) creates space in memory for the object and initializes its fields

Class in Java - Constructors

- *Bicycle* class could have more than one constructor
  - A no-argument constructor
  - A constructor with arguments

```
public class Bicycle{
   int speed;
   public Bicycle(int startSpeed)
     { speed = startSpeed; }
   public Bicycle()
       { speed = 0; }
  …}
```

Class in Java - Constructors

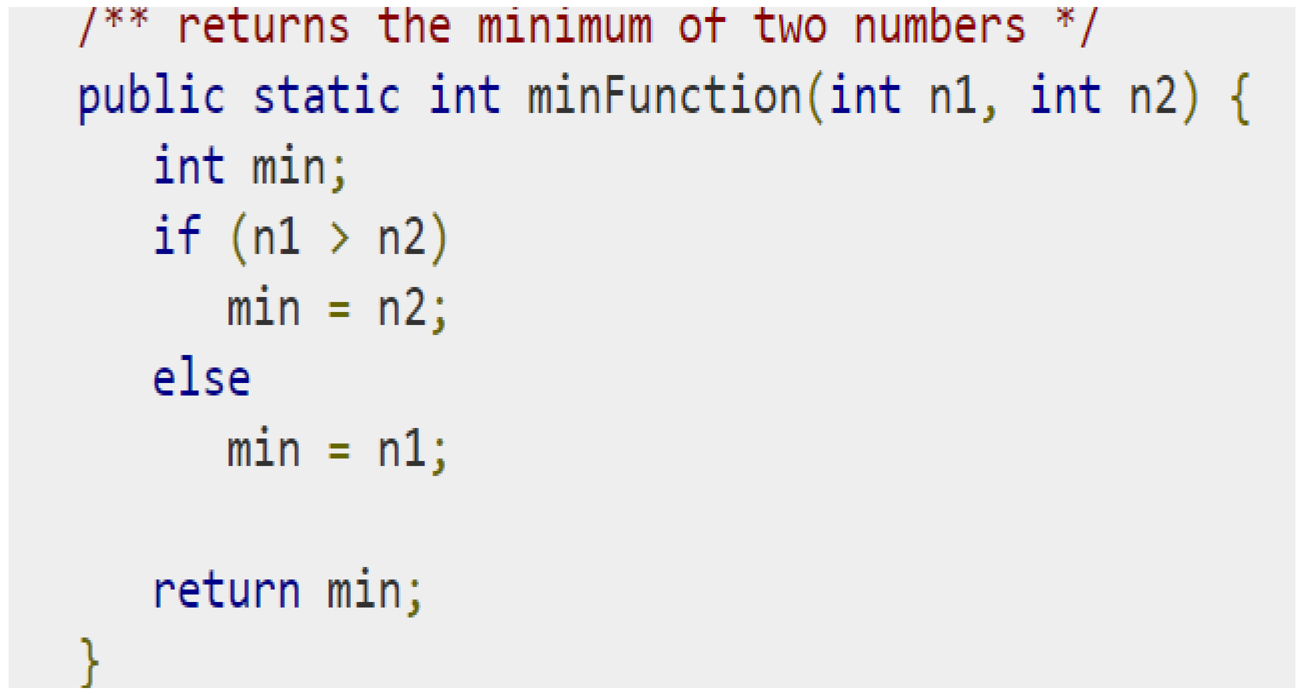
- Remarks:
  - Both constructors can be declared in *Bicycle* because they have different argument lists
  - Java differentiates constructors based on:
    - » The number of arguments in the list
    - » The types of arguments
  - Cannot be written two constructors that have the same number and type of arguments for the same class
    - » This causes a compile-time error

Class in Java - Constructors

- Remarks:
  - If there isn’t constructors defined in your class
    - The compiler automatically provides a no-argument (i.e., default constructor)
  - The default constructor will call the default constructor of the superclass
    - The compiler will throw an error, if the superclass doesn't have a default constructor
  - Access modifiers
    - Can be used in the constructor's declaration to control the access of other classes to the constructor

Class in Java - Passing Information to a Method/Constructor

- A method/constructor
  - Declares the number and the type of the arguments for that method/constructor



```
/** returns the minimum of two numbers */
public static int minFunction(int n1, int n2) {
   int min;
   if (n1 > n2)
      min = n2;
   else
      min = n1;

   return min;
}
```

- The method *minFunction* has two parameters: *n1, n2*
- Any data type can be used for a parameter of a method/constructor
  - Primitive data types
    - e.g., doubles, floats, or integers
  - Reference data types
    - e.g., objects or arrays

Class in Java - Passing Information to a Method/Constructor

- Example of a method that accepts an array as an argument

  public Polygon *polygonFrom*(Point[] *corners*)

  {//method body goes here}

  - We consider that this method is responsible for:
    - » Creating a new *Polygon* object
    - » Initializing the object from an array of *Point* objects

Class in Java - Passing Information to a Method/Constructor

- In Java we could declare methods with an arbitrary number of arguments by using *varargs* which is a feature introduced in Java 5.
  - *varargs* is used to pass an arbitrary number of values to a method
  - using *varargs* the method can be called with any number of parameters, including none
  - Varargs are represented by an ellipsis (i.e., "...") after the type of the parameter in the method signature.

Class in Java - Passing Information to a Method/Constructor

- Here's an example of a method that uses varargs to accept an arbitrary number of integers:

```
public static int sum(int... nums) {
    int sum = 0;
    for (int num : nums) {
        sum += num;
    }
    return sum;
}
```

  - In this example, the "sum" method takes an arbitrary number of integers as input using the "int... nums" parameter. This means that the method can be called with any number of integer parameters, including none.
  - For example, we can call the method like this:

```
int result1 = sum(1, 2, 3); // result1 = 6
int result2 = sum(4, 5, 6, 7, 8); // result2 = 30
int result3 = sum(); // result3 = 0
```

    » As shown in the example, the method can be called with any number of integer parameters, including none. The varargs parameter is treated as an array inside the method body, so we can loop through it and perform any desired operation.

Class in Java - Passing Information to a Method/Constructor

- Another example of a method that uses varargs to accept an arbitrary number of Objects:

```
void metoda(Object ... varargs)
  {
      for(int i=0; i<varargs.length; i++)
          System.out.println(varargs[i]);
   }
           ...
          metoda("Hello");
          metoda("Hello", "Java");
```

Class in Java - Passing Information to a Method/Constructor

- Parameters name of a method/constructor
  - In general, it is recommended to use unique parameter names that are descriptive and help to clearly communicate the purpose of the parameter within the method.
  - It is possible to use the same name for a parameter and one of the class's fields.
    - » In this case, the parameter "shadows" the field, meaning that the field is not accessible inside the method or constructor, unless it is accessed using the "this" keyword.
- Shadowing fields
  - Shadowing fields can make the code more difficult to read and can lead to errors if the programmer forgets that the parameter is shadowing the field.
  - It is generally recommended to use unique parameter names to avoid confusion and make the code easier to understand.

Class in Java - Passing Information to a Method/Constructor

- Example of shadowing fields

```
public class Person {
    private String name;
    private int age;
        public Person(String name, int age) {
        this.name = name;
       // "this.name" refers to the instance variable "name"
        this.age = age;
       // "this.age" refers to the instance variable "age"
    }
        public void printInfo(String name) {
        System.out.println("Name: " + name);
        // "name" refers to the parameter "name"
        System.out.println("Age: " + age);
        // "age" refers to the instance variable "age"
    }
}
```

- In this example, we have a class called "Person" with two instance variables, "name" and "age".
- The constructor takes in two arguments, "name" and "age", and sets the instance variables to the corresponding values.
- The class also has a method called "**printInfo**" that takes in a parameter called "name".
    - Inside the method, we print out the value of the parameter "name" and the value of the instance variable "age".
    - However, since the parameter "name" has the same name as the instance variable "name", it "shadows" the instance variable and makes it inaccessible inside the method.

Class in Java - Passing Information to a Method/Constructor

- Example of shadowing fields

```
public class Person {
    private String name;
    private int age;
        public Person(String name, int age) {
        this.name = name; // "this.name" refers to
the instance variable "name"
        this.age = age; // "this.age" refers to the
instance variable "age"
    }
        public void printInfo(String name) {
        System.out.println("Name: " + name); //
"name" refers to the parameter "name"
        System.out.println("Age: " + age); // "age"
refers to the instance variable "age"
    }
}
```

```
Person person = new Person("Alice", 25);
person.printInfo("Bob");

// Output:
// Name: Bob
// Age: 25
```

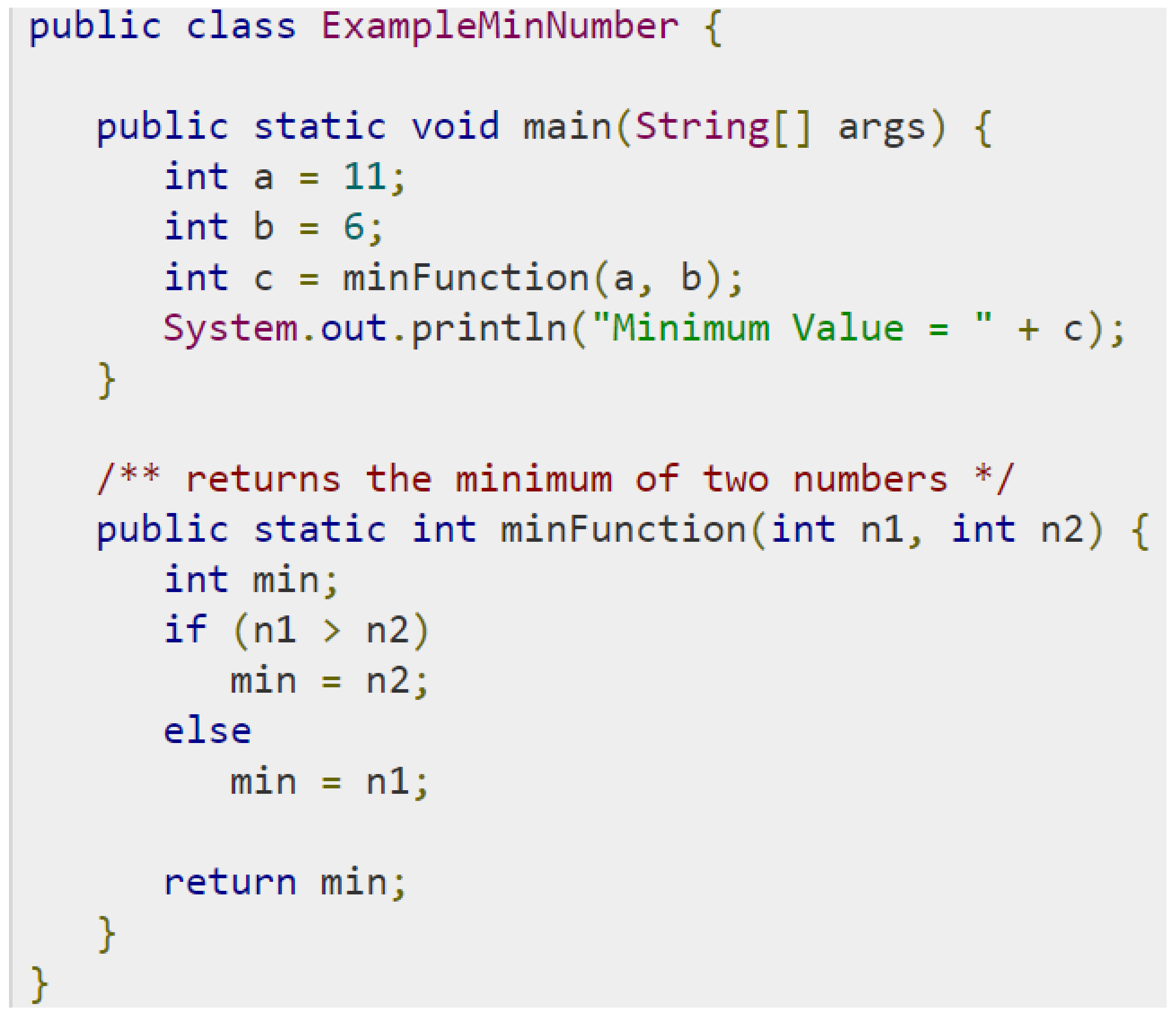
- In this output, we see that the value of the "name" parameter ("Bob") is printed instead of the value of the instance variable "name" ("Alice").
- This is because the parameter "name" shadows the instance variable "name" inside the "printInfo" method.

Class in Java - Passing Information to a Method/Constructor

- Method Calling
  - For using a method, it should be called
  - There are two ways in which a method is called depending
    - » if the method returns a value or
    - » if returning nothing (no return value)
- When a program invokes a method, the program control gets transferred to the called method
- The called method then returns control to the caller in two conditions:
  - When the return statement is executed
  - When the method code is complete executed

Class in Java - Passing Information to a Method/Constructor

- Consider the following example:

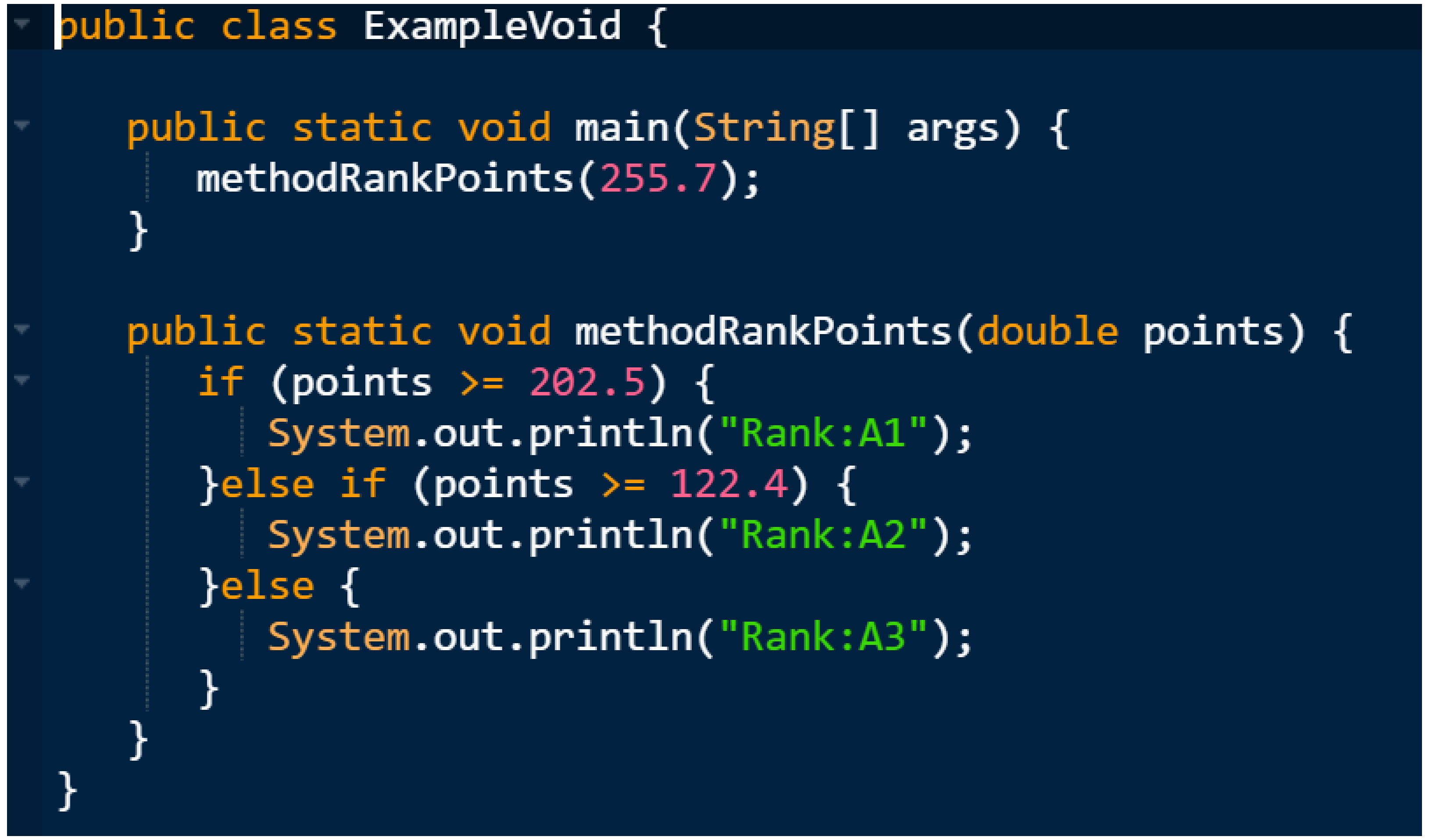


```
public class ExampleMinNumber {

   public static void main(String[] args) {
      int a = 11;
      int b = 6;
      int c = minFunction(a, b);
      System.out.println("Minimum Value = " + c);
   }

   /** returns the minimum of two numbers */
   public static int minFunction(int n1, int n2) {
      int min;
      if (n1 > n2)
         min = n2;
      else
         min = n1;

      return min;
   }
}
```

In this example the called method returns control to the caller when the return statement is executed.

Class in Java - Passing Information to a Method/Constructor

- The void keyword
  - Allows us to create methods which do not return a value
- Let's consider an example in which we have defined a void method *methodRankPoints*
  - We can call the void method as follow: *methodRankPoints(255.7)*



```java
public class ExampleVoid {

    public static void main(String[] args) {
        methodRankPoints(255.7);
    }

    public static void methodRankPoints(double points) {
        if (points >= 202.5) {
            System.out.println("Rank:A1");
        }else if (points >= 122.4) {
            System.out.println("Rank:A2");
        }else {
            System.out.println("Rank:A3");
        }
    }
}
```

In this example the called method returns control to the caller when the called method ends the execution of the code between the method braces

Class in Java - Passing Information to a Method/Constructor

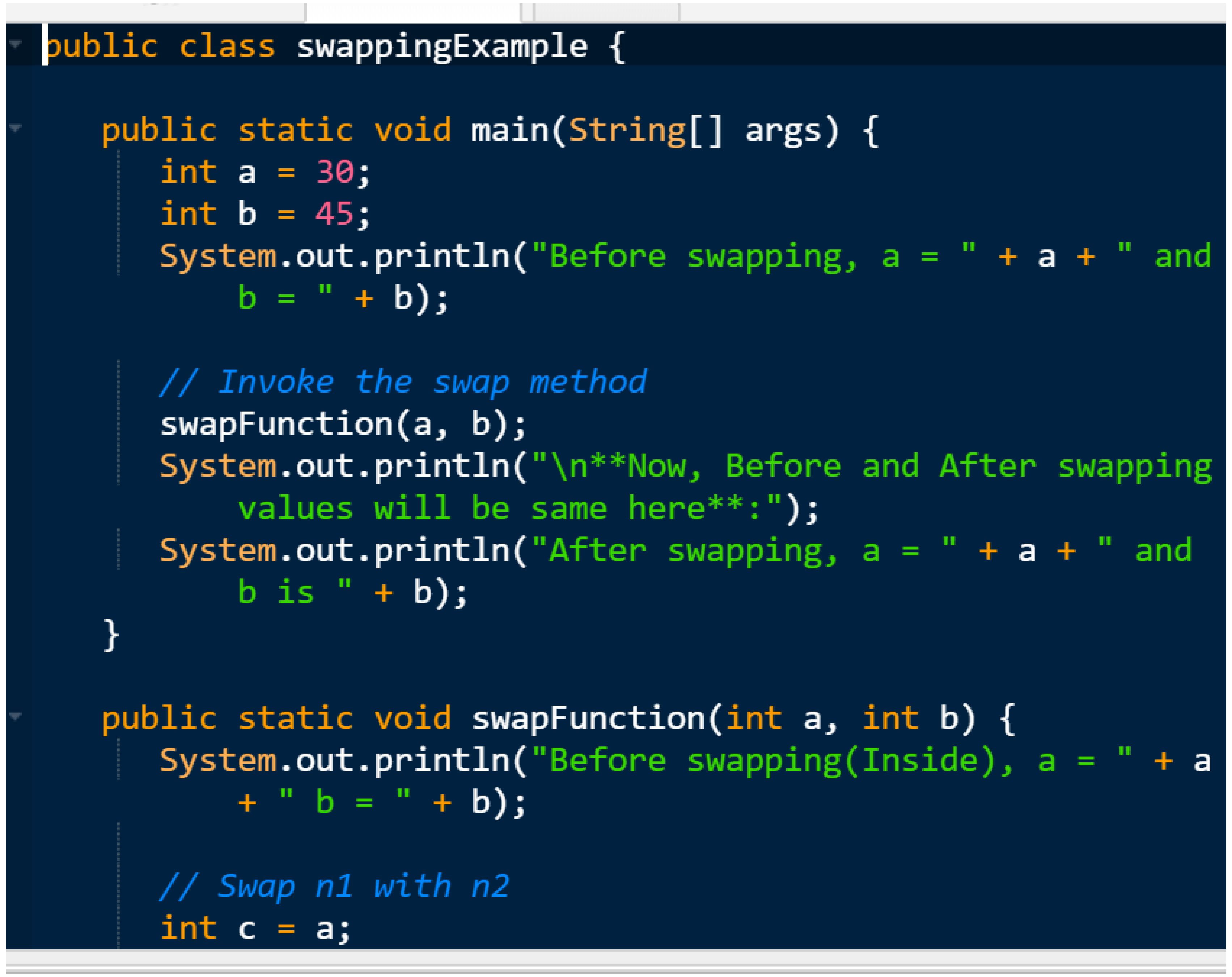
- When calling a method or constructor in a programming language, arguments are typically passed to the method or constructor.
- The arguments should be passed in the same order as the order of the parameters specified in the method or constructor.
  - » Arguments can be passed by value or by reference

Class in Java - Passing Information to a Method/Constructor

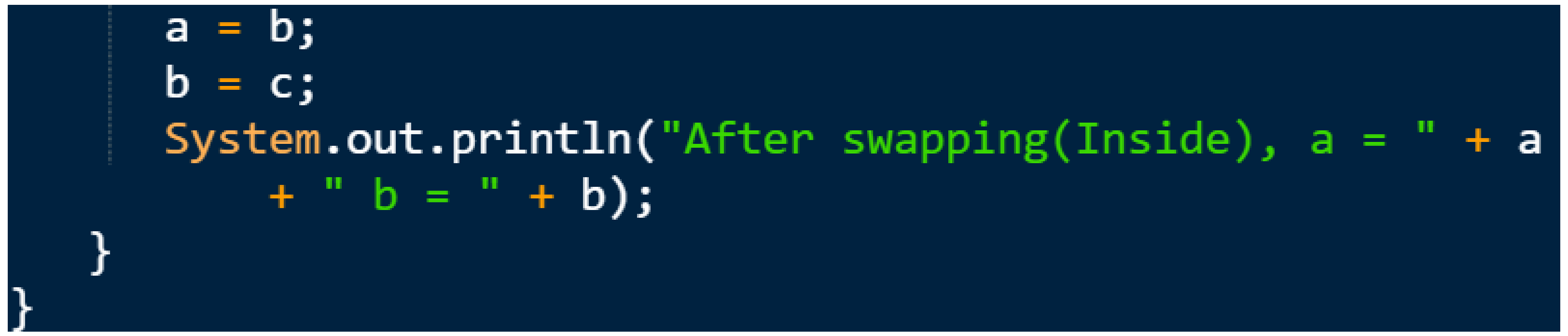
- Passing primitive data type arguments
  - In Java, when we pass a primitive data type such as int, double, boolean, etc., the argument is passed by value.
  - This means that the method receives a copy of the argument's value, and any changes made to the parameter inside the method do not affect the original value of the argument outside the method
    - Any changes to the values of the parameters exist only within the method
    - When the method returns, the parameters are gone and any changes to them are lost

Class in Java - Passing Information to a Method/Constructor

- Passing primitive data type arguments

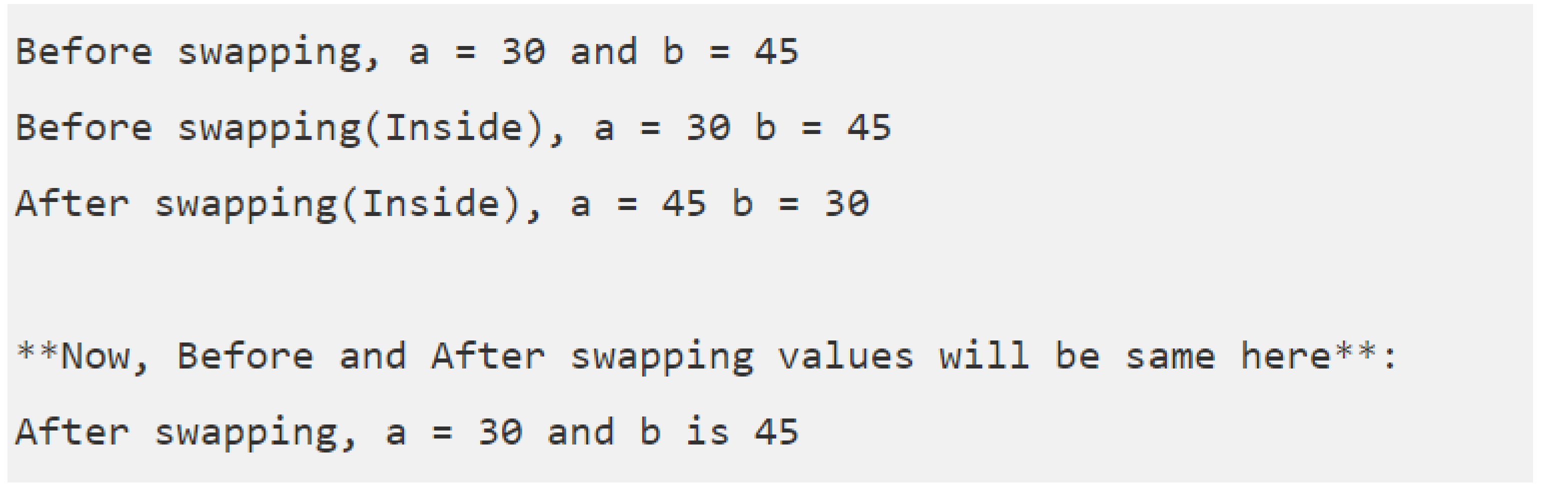


```
public class swappingExample {

    public static void main(String[] args) {
        int a = 30;
        int b = 45;
        System.out.println("Before swapping, a = " + a + " and
            b = " + b);

        // Invoke the swap method
        swapFunction(a, b);
        System.out.println("\n**Now, Before and After swapping
            values will be same here**:");
        System.out.println("After swapping, a = " + a + " and
            b is " + b);
    }

    public static void swapFunction(int a, int b) {
        System.out.println("Before swapping(Inside), a = " + a
            + " b = " + b);

        // Swap n1 with n2
        int c = a;
```



```
        a = b;
        b = c;
        System.out.println("After swapping(Inside), a = " + a
            + " b = " + b);
    }
}
```

This will produce the following result



```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

Class in Java - Passing Information to a Method/ Constructor

- Passing reference data type arguments
  - Reference data type parameters (objects) are also passed into methods by value
    - » This means that the method receives a copy of the reference to the object, not a copy of the object itself.
    - » Any changes made to the object inside the method affect the original object outside the method.
      - The values of the object's fields can be changed in the method, if they have the proper access level

Class in Java - Passing Information to a Method/ Constructor

- For example, we create a class **Number**

```
public class Number {
    private int value;
    public Number(int value) {
        this.value = value;
    }
    public int getValue() {
      return value;
    }
    public void setValue(int value) {
      this.value = value;
    }
    public String toString() {
       return String.valueOf(this.getValue());
    }
    public static void swapNumber (Number a, Number b ) {
     int c = a.getValue();
     a.setValue(b.getValue());
     b.setValue(c);    }
```

```
public static void main(String args[])
    {
    Number m = new Number(3);
    Number n = new Number(8);
    System.out.println("Before swap");
    System.out.println("m = " + m.toString());
    System.out.println("n = " + n.toString());
    swapNumber(m, n);
    System.out.println("After swap");
    System.out.println("m = " + m.toString());
    System.out.println("n = " + n.toString());
    }
}
```

Output will be:

```
Before swap
m = 3
n = 8
After swap
m = 8
n = 3
```

Objects

- A Java program creates many objects
- The objects interact by invoking methods
- Through object interactions, a program can carry out various tasks such as:
  - Implementing a GUI
  - Running an animation
  - Sending and receiving information over a network
- Once an object has completed the work for which it was created, its resources are recycled for use by other objects

Objects

```
public class CreateObject{
  public static void main(String[] args) {
      //Declare and create  the objects
     Point origin = new Point(23, 94);
     Rectangle rect = new Rectangle(origin, 100, 200);
     Rectangle rectT = new Rectangle(50, 100);
    System.out.println("Width of rect: " + rect.width);
    System.out.println("Height of rect: " +rect.height);
    System.out.println("Area of rect: " + rect.getArea());
     //set rectTwo's position
     rectT.origin = origin;
      //display rectTwo's position
     System.out.println("X Position of rectT: " +  rectT.origin.x);
     System.out.println("Y Position of rectT: " + rectT.origin.y); }
```

Life Cycle of an Object

- The class *CreateObject* creates three objects:
  - One *Point* object
  - Two *Rectangle* objects
- Each of these statements has three parts:
  - Declaration:
    - » Associate a variable name with an object type
  - Instantiation:
    - » The **new** keyword is a Java operator that creates the object
  - Initialization:
    - » The **new** operator is followed by a call to a constructor, which initializes the new object

Life Cycle of an Object - Declaring a variable to refer to an object

- Example of declaring a variable

  type *name*;

  - The *name* will be used to refer to data whose type is **type**
  - For a primitive variable, the declaration also reserves the proper amount of memory for the variable
- Example of declaring a primitive variable in Java

```
int age; // declaring an integer variable named "age"
double height = 1.75; // declaring and initializing a double variable named "height"
boolean isStudent = true; // declaring and initializing a boolean variable named "isStudent"
String name = "John"; // declaring and initializing a String variable named "name"
```

  - In this example, we are declaring four variables of different data types (int, double, boolean, and String).
  - The first variable, age, is declared without an initial value, which means it has a default value of 0.
  - The second variable, height, is declared and initialized with the value 1.75.
  - The third variable, isStudent, is declared and initialized with the value true.
  - The fourth variable, name, is declared and initialized with the value "John".
  - It's important to note that in Java, variables must be declared before they can be used. This means that you cannot use a variable before it has been declared.

Life Cycle of an Object - Declaring a variable to refer to an object

- Example of declaring a variable referring an object:

Point *originOne*;

  - Declaring a reference variable does not create an object
  - Before to used the *originOne* in the code an object must be assigned to it;
    - Otherwise, a compiler error is returned

Life Cycle of an Object - Instantiating an object

- Is realized with the **new** operator
  - **new** operator instantiates a class by allocating memory for a new object and returning a reference to that memory

Life Cycle of an Object - Initializing an object

- Implies invoking the class constructor which returns a reference to the object it created
  - The reference is assigned to a variable of the appropriate type:

Point *originOne* = **new Point**(23, 94);

- Note: The reference returned by the **new** operator does not have to be assigned to a variable. It can also be used directly in an expression

int *height* = **new** *Rectangle*().*height*;

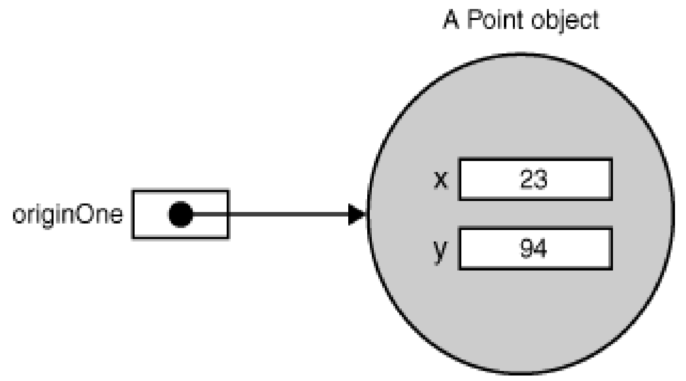
Life Cycle of an Object - Initializing an object

```
public class Point {
   public int x = 0;
   public int y = 0;
   //constructor
   public Point(int a, int b)
        { x = a; y = b; }
 }
```

– The class contains a constructor with two arguments (**int** *a*, **int** *b*)

Point *originOne* = **new** Point(23, 94);

– The result of executing this statement can be illustrated in the following figure:



Life Cycle of an Object - Example

```
public class Rectangle {
   public int width = 0;
   public int height = 0;
   public Point origin;
   public Rectangle()
     {origin = new Point(0, 0);  }
   public Rectangle(Point p)
    {  origin = p;  }
   public Rectangle(int w, int h)
     {  origin = new Point(0, 0);
        width = w;
        height = h;  }
   public Rectangle(Point p, int w, int h)
      { origin = p;
        width = w;
        height = h; }
```

```
// a method for moving the rectangle
public void move(int x, int y)
        {   origin.x = x;
            origin.y = y;
         }
// a method for computing the area of the rectangle
public int getArea()
 {   return width * height;  }
```

Life Cycle of an Object –Example (cont.)

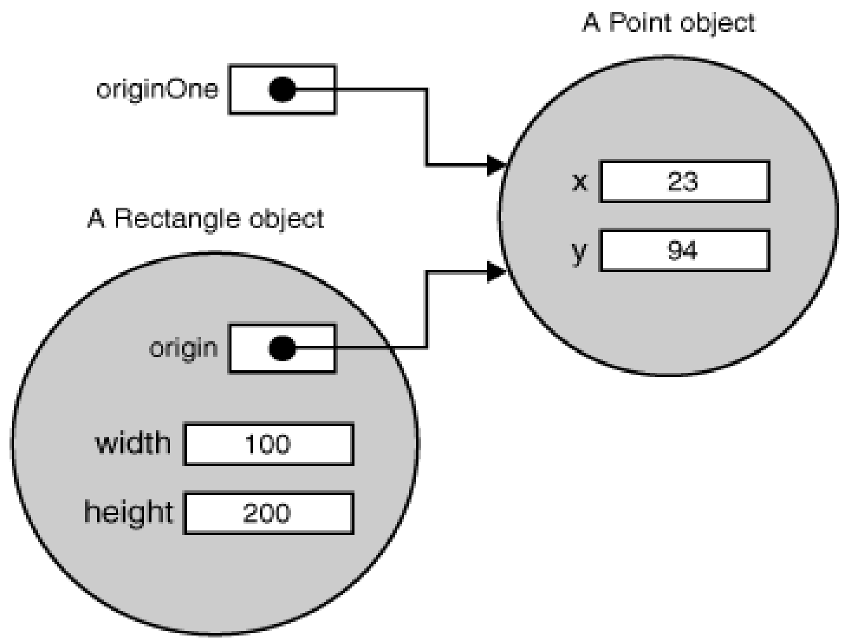
- Class *Rectangle*
  - Each constructor provides initial values for the rectangle's fields, using both primitive and reference types
- Java compiler differentiates constructors based on the number and the type of the arguments

Point *originOne* = **new** Point(23, 94);

Point *origin* = **new** Point(23, 94);

Rectangle *rect* = **new** Rectangle(*origin*, 100, 200);

- There are two references to the same **Point** object
  - An object can have multiple references to it



This Keyword

- **this**
  - Is a reference to the current object within an instance method/constructor
- Using **this** with a field
  - **this** keyword is used with a field when the field is shadowed by a method or constructor parameter

This keyword

```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int x, int y) {
            this.x = x;
            this.y = y;
  }
}
```

– Each argument to the constructor shadows one of the object's fields

– To refer to the **Point** field *x*, the constructor must use **this**.*x*

This keyword

- **this keyword** can be used within a constructor to call another constructor in the same class**.**
  - This is called an explicit constructor invocation or constructor chaining.

```
public class Rectangle {
  private int x, y;
  private int width, height;
  public Rectangle()
     {  this(0, 0, 0, 0);}
 public Rectangle(int width, int height)
     {  this(0, 0, width, height);   }
 public Rectangle(int x, int y, int width, int height)
     {
       this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
       }
... }
```

This keyword

- A constructor
  - Initializes some/all of the class's member variables
- The compiler identifies which constructor to call based on:
  - The number and the type of arguments
- The invocation of another constructor in a constructor must be the first line

Packages

- Package
  - Allows a developer to group classes and interfaces together
  - Classes and interfaces within a package are related because they typically have a common purpose or perform a specific set of tasks.
    - » For example, a package might contain classes that handle database operations, or interfaces that define a set of related behaviors.
- Classes/interfaces, part of the Java platform, are members of various packages:
  - Fundamental classes are in **java.lang** package
  - Classes for reading and writing are in **java.io** package

Packages

- Benefits of using packages
  - Classes/interfaces can be easily retrieval
    - A specific class can be retrieval based on the functionality it provides
      - It will naturally find it in the package which provide that functionality
  - Names of classes/interfaces of a specific programmer won't conflict with those of other programmers
    - Example
      - A programmer implement a **String** class and place it in a package
      - This class will be distinguishable from the **java.lang.String** class that comes with the Java API
  - Using packages makes access control to source code much easier

Packages

- Example of using packages
  - Consider
    - A group of classes that represent graphic objects (e.g., circles, lines, points)
    - An interface, *Drag* that classes implement if they can be dragged with the mouse

Packages

//in the Draggable.java file

public interface *Drag* { . . . }

//in the Graphic.java file

public abstract class *Graphic* { . . . }

//in the Circle.java file

public class *Circle* **extends** *Graphic* **implements** *Drag* { . . . }

//in the Rectangle.java file

public class *Rectangle* **extends** *Graphic* **implements** *Drag* { . . . }

//in the Point.java file

public class *Point* **extends** *Graphic* **implements** *Drag* { . . . }

//in the Line.java file

public class *Line* **extends** *Graphic* **implements** *Drag* { . . . }

Packages

- The classes and the interface from the previous example can be grouped in a package because:
  - The classes are related
  - All these classes provide graphics-related functions
  - The names of the classes won't conflict with the classes' names in other packages because the package creates a new namespace
  - The classes within the package
    - » Have unrestricted access to one another
    - » Have restrict access for classes outside the package

Packages

- Steps in creating a package
  - Choose the name for the package
  - Put the **package** statement with that name at be the first line in the source file
    - There can be only one package statement in each source file
    - The package statement it applies to all classes/interfaces in the source file
      - A source file can contain multiple classes declarations
        - Only one of the classes can be public, and it must have the same name as the source file
    - The non-public classes within the source file can not be accessible from outside of the package (i.e., package private classes)

Packages

- Example of creating a package

```
//in the Drag.java file
package graphics;
public interface Drag {…}

//in the Graphic.java file
package graphics;
public abstract class Graphic {...}

//in the Circle.java file
package graphics;
public class Circle extends Graphic implements Drag {...}

//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic implements Drag
{...}

//in the Point.java file
package graphics;
public class Point extends Graphic implements Drag{…}

//in the Line.java file
package graphics;
public class Line extends Graphic implements Drag {...}
```

Packages

- If it is not using a package statement, then the classes are put in a default package
- Default package
  - Is used only for:
    - Small or temporary applications or
    - When you are just beginning the development process
- Otherwise, classes and interfaces belong in named packages

Packages

- Naming a package
  - Considering the previous example:
    - It defines a *Rectangle* class but there is already a *Rectangle* class in the **java.awt** package
    - The compiler allows both classes to have the same name if they are in different packages
    - The fully qualified name of each *Rectangle* class includes the package name
      - e.g. *graphics.Rectangle*
      - e.g. *java.awt.Rectangle*
- Naming convention
  - Companies use their reversed Internet domain name to begin their package names
    - e.g., *com.example.orion* for a package named *orion* created by a programmer at *example.com*
  - Packages in the Java language begin with *java*. or *javax.*

Packages

- Using package members
  - Classes and interfaces that are contained in package are *package members*
  - To use a public class/interface from outside its package, you must either:
    - » Refer to the class/interface by its fully qualified name
    - » Import the package class/interface
    - » Import the entire package that class/interface
- A package class/interface's simple name
  - Can be used if the code you are writing is in the same package as that class/interface or
  - If that class/interface has been imported

Packages

- The class/interface's fully qualified name must be used if:
  - The class/interface is used in a different package and the package of that class/interface has not been imported
- Example
  - The fully qualified name can be used to create an instance of *graphics.Rectangle*

    graphics.Rectangle *myRect* = **new** g**raphics.Rectangle**();

  - The fully qualified name for the *Rectangle* class declared in the *graphics* package is *graphics.Rectangle*
    - » This approach is ok in the case of infrequent use of the class

Packages

- To import a specific class/interface into the current file the following statement is used:

import graphics.Rectangle;

  - In this case, the *Rectangle* class can be accesses by its simple name:

Rectangle *myRectangle* = **new Rectangle**();

- Importing an entire package
  - Is recommended when many members (classes of interfaces) from a package are used
  - Is made by using the import statement with the asterisk (*) character

import graphics.*;

Packages

- Importing an entire package:
  - Asterisk in the **import** statement
    - » Is used to specify all the classes/ interfaces within a package are imported

```
import graphics.*;
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

    - » Cannot be used to match a subset of the classes in a package
      - The following example doesn't match all the classes in the *graphics* package that begin with A.

```
import graphics.A*;  //does not work
```

Packages

- Packages appear to be hierarchical, but they are not:
  - Java API includes:
    - » **java.awt** package
    - » **java.awt.color** package
    - » **java.awt.font** package
    - » Many others that begin with **java.awt**
- For example, **java.awt.color** and **java.awt.font** packages are not included in the **java.awt**

Packages

- **java.awt** is used for a number of related packages to make the relationship evident, but not to show inclusion
- **java.awt.***
  - Imports all classes from the **java.awt** package
  - Doesn’t import from **java.awt.font** or **java.awt.color**
- If it is necessary to use classes from two packages (e.g., java.awt.color and java.awt), both packages must be imported:

```
import java.awt.*;
import java.awt.color.*;
```

Packages

- Name collisions
  - If a class in one package has the same name with a class from another package and both packages are imported, you must refer to each class by its qualified name

- Example of name collisions
  - Consider the **graphics** package that defines a class **Rectangle**
  - **java.awt** package also contains a **Rectangle** class
  - If both **graphics** and **java.awt** are imported, the following statement is ambiguous:

    Rectangle *rect*;

  - In this case the member's fully qualified name must be used to indicate exactly which **Rectangle** class you want

    graphics.Rectangle rect;

Packages

- Static import statement
  - Is used when you need frequent access to:
    - Static final fields (constants)
    - Static methods from one or two classes
  - Gives a way to access the constants and static methods so that you do not need to prefix the name of their class

Packages

- Example of static import statement
  - Consider **java.lang.Math** class which:
    - » Defines the **PI** constant
    - » Many static methods, including methods for calculating *cosines*, *maxima*, *minima*, etc.

public static final double PI 3.141592653589793

public static double cos(double *a*)

  - To use these methods/constants from another class
    - » You prefix the class name, as follows:

double *r* = **Math.cos**(**Math.PI** * *theta*);

    - » You can use the static import statement to import the static members of the class

import static java.lang.Math.*; …..

double *r* = **cos**(*PI* * *theta*);

Managing source and class files

- Many implementations of the Java platform rely on:
  - Hierarchical file systems to manage source and class files
- The strategy is:
  - Put the source code for a class/interface in a text file whose name is the name of the class and whose extension is **.java**

    //in the Rectangle.java file

    package *graphics*;

    public class *Rectangle* { . . . }

  - Then, put the source file in a directory whose name reflects the name of the package to which the class belongs:

    .....\graphics\*Rectangle*.java

Managing source and class files

- The qualified name of the package member (e.g., class/ interfaces) and the path name to the file containing the class/ interface are parallel

class *name*	*graphics.Rectangle*
pathname to file	*graphics\Rectangle.java*

- Suppose there is a company with the Internet domain name example.com
  - All of its package names will have **com.example** as a prefix, and each part of the package name corresponds to a subdirectory
  - For instance, if the company has a **com.example.graphics** package that includes a source file named **Rectangle**.java, the file would be located in a sequence of subdirectories as follows:

```
....com\example\graphics\Rectangle.java
```

Managing source and class files

- For every class that is defined in a program, the compiler generates a distinct output file.
- These files are named after their corresponding class, with the extension .class.
- As an example, let's take the following source code from Rectangle.java file:

```
//in the Rectangle.java file
package com.example.graphics;
public class Rectangle { . . . } class Helper{ . . . }
```

- After compilation, the resulting output files will be located in the following directory structure, relative to the parent directory of the output files:

```
com/example/graphics/Rectangle.class
com/example/graphics Helper.class
```

- In other words, the compiled output for the Rectangle class will be saved in the file named Rectangle.class, while the Helper class will have its own output file named Helper.class.
- Both of these files will be located in the com/example/graphics/ directory relative to the parent directory of the output files.