

18. Objects and Classes (Part 2)

Java

Fall 2009

Instructor: Dr. Masoud Yaghini

Outline

- Static Variables, Constants, and Methods
- Visibility Modifiers
- Data Field Encapsulation
- Immutable Objects and Classes
- Passing Objects to Methods
- The Scope of Variables
- Array of Objects
- References

Static Variables, Constants, and Methods



Instance Variables, and Methods

- **Instance (non-static) variables** belong to a specific instance.
- **Instance methods** are invoked by an instance of the class.

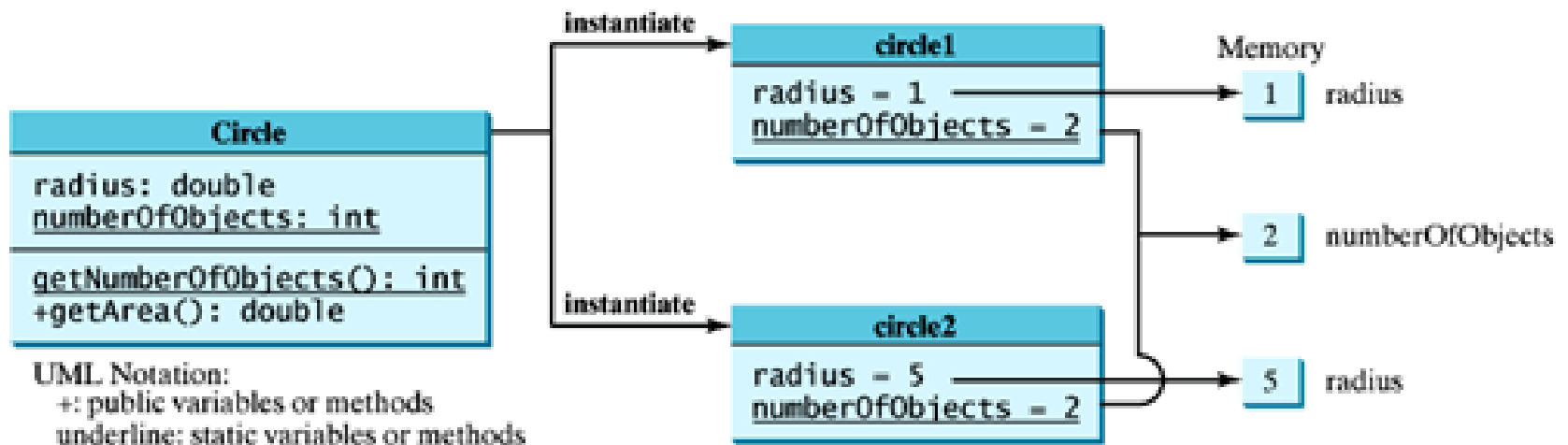
Static Variables, Constants, and Methods

- **Static variables (Class variable)** are shared by all the instances of the class.
- **Static methods** are not tied to a specific object. Static methods can be called without creating an instance of the class.
- **Static constants** are final variables shared by all the instances of the class.
- To declare static variables, constants, and methods, use the **static** modifier.

Objects and Classes: Part 2

Static Variables, Constants, and Methods

- Let us modify the `Circle` class by adding a static variable `numberOfObjects` to count the number of circle objects created and the static method `getNumberOfObjects`



Static Variables, Constants, and Methods

- Constants in a class are shared by all objects of the class.
- Thus, constants should be declared `final static`.
- For example, the constant `PI` in the `Math` class is defined as:

```
final static double PI = 3.14159265358979323846;
```

TestCircle2.java

- Example:
 - Circle2.java
 - TestCircle2.java
- The output of the program:
Before creating c2
c1 is : radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1's radius to 9
c1 is : radius (9.0) and number of Circle objects (2)
c2 is : radius (5.0) and number of Circle objects (2)

TestCircle2.java

- You can replace `Circle2.numberOfObjects` by
 - `c1.numberOfObjects` and
 - `c2.numberOfObjects`.
- You can also replace `Circle2.getNumberOfObjects` by
 - `Circle2.getNumberOfObjects()`.

Static Variables, Constants, and Methods

- To improve readability use
`ClassName.methodName(arguments)`
 - to invoke a static method and
`ClassName.staticVariable`
 - to use a static variable
- Because the user can easily recognize the static method and data in the class.

Import static variables and methods

- You can import static variables and methods from a class.
- The imported data and methods can be referenced or called without specifying a class.
- For example, you can use `PI` (instead of `Math.PI`), and `random()` (instead of `Math.random()`),
- if you have the following import statement in the class:

```
import static java.lang.Math.*;
```

Static Variables, Constants, and Methods

- **Instance methods** can use both:
 - Static variables and methods, and
 - Instance variables and methods
- **Static methods** can use only:
 - Static variables and methods
- Because static variables and methods belong to the class as a whole and not to particular objects.
- What is wrong?
 - [Test1.java](#)

Static Variables, Constants, and Methods

- How do you decide whether a variable or method should be an instance one or a static one?
 - A variable or method that is dependent on a specific instance of the class should be an instance variable or method, otherwise it should be a static variable or method.
- None of the methods in the `Math` class is dependent on a specific instance. Therefore, these methods are static methods.
- The main method is static, and can be invoked directly from a class.



Visibility Modifiers



Visibility Modifiers

- Java provides several modifiers that control access to data fields, methods, and classes.
 - **public**: The class, variable, or method is visible to any class in any package (in a NetBeans project)
 - **By default** (no modifier), the class, variable, or method can be accessed by any class in the same package. This is known as **package-private** or **package-access**.
 - **private**: The data or methods can be accessed only by the own class.

Visibility Modifiers

```
package p1;
```

```
public class C1 {  
    public int x;  
    int y;  
    private int z;  
  
    public void m1() {  
    }  
    void m2() {  
    }  
    private void 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();  
    }  
}
```

- The **private modifier** restricts access to within a class
- The **default modifier** restricts access to within a package
- The **public modifier** enables unrestricted access

Visibility Modifiers

- If a class is not declared public, it can only be accessed within the same package

```
package p1;
```

```
class C1 {  
    ...  
}
```

```
public class C2 {  
    can access C1  
}
```

```
package p2;
```

```
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

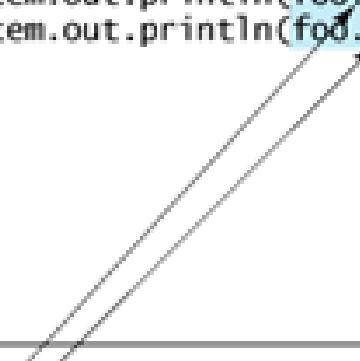
Visibility Modifiers

- An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

```
public class Foo {  
    private boolean x;  
  
    public static void main(String[] args) {  
        Foo foo = new Foo();  
        System.out.println(foo.x);  
        System.out.println(foo.convert());  
    }  
  
    private int convert() {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is OK because object foo is used inside the Foo class

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



(b) This is wrong because x and convert are private in Foo.

Note

- Visibility modifiers are used for the members of the class, **not local variables** inside the methods.
- Using a visibility modifier on local variables would cause a compilation error.

Objects and Classes: Part 2

Note

- In most cases, the constructor should be public.
- However, if you want to prohibit the user from creating an instance of a class, you can use a `private` constructor.
- For example, there is no reason to create an instance from the `Math` class because all of the data fields and methods are static.
 - One solution is to define a dummy private constructor in the class.
 - The `Math` class cannot be instantiated because it has a private constructor, as follows:

```
private Math()  
{  
}
```



Data Field Encapsulation



Data Field Encapsulation

- Why Data Fields Should Be private?
- **To protect data.**
 - For example, `numberOfObjects` is to count the number of objects created, but it may be set to an arbitrary value (e.g., `Circle2.numberOfObjects = 10`).
- **To make class easy to maintain.**
 - Suppose you want to modify the `Circle2` class to ensure that the `radius` is non-negative after other programs have already used the class.
 - You have to change not only the `Circle2` class, but also the programs that use the `Circle2` class.
 - Such programs are often referred to as **clients**.

Data Field Encapsulation

- **Data field encapsulation**
 - To prevent direct modifications of properties, you should declare the field private, using the private modifier.
 - This is known as *data field encapsulation*.
- To make a private data field accessible, provide a **get** method to return the value of the data field.
- To enable a private data field to be updated, provide a **set** method to set a new value.

Data Field Encapsulation

- A `get` method is referred to as a **getter** (or **accessor**), and a `set` method is referred to as a **setter** (or **mutator**).
- `get` method has the following signature:
`public returnType getPropertyname()`
- `set` method has the following signature:
`public void setPropertyName(dataType propertyValue)`

Data Field Encapsulation

- The class diagram to create a new circle class with a private data field **radius** and its associated **accessor** and **mutator** methods.

The - sign indicates private modifier →

```
-radius: double  
-numberOfObjects: int  
  
+Circle()  
+Circle(radius: double)  
+getRadius(): double  
+setRadius(radius: double): void  
+getNumberOfObject(): int  
+getArea(): double
```

The radius of this circle (default: 1.0).
The number of circle objects created.

Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

TestCircle3.java: Demonstrate private modifier

- Example:
 - Circle3.java
 - TestCircle3.java

- The output:

The area of the circle of radius 5.0 is 78.53981633974483

The area of the circle of radius 5.5 is 95.03317777109125

- Note:
 - When you compile `TestCircle3.java`, the Java compiler automatically compiles `Circle3.java` if it has not been compiled since the last change.

Immutable Objects and Classes



Immutable Objects and Classes

- If the contents of an object cannot be changed once the object is created, the object is called an **immutable object** and its class is called an **immutable class**.
- If you delete the `set` method in the `Circle3` class in the preceding example, the class would be immutable because `radius` is private and cannot be changed without a `set` method.

Immutable Objects and Classes

- A class with all private data fields and no mutators is not necessarily immutable.
- An example:
 - [Student.java](#)
 - [BirthDate.java](#)
 - [TestStudent.java](#)

What Class is Immutable?

- For a class to be immutable:
 - it must mark all data fields private and
 - provide no mutator methods and
 - no accessor methods that would return a reference to a mutable data field object.

Passing Objects to Methods



Passing Objects to Methods

- Like passing an array, passing an object is actually passing the reference of the object.
- Java uses exactly one mode of passing arguments: **pass-by-value**.
 - Passing by value for primitive type value (the value is passed to the parameter)
 - Passing by value for reference type value (the value is the reference to the object)

TestPassObject.java

- Example:
 - [TestPassObject.java](#)
- The output:

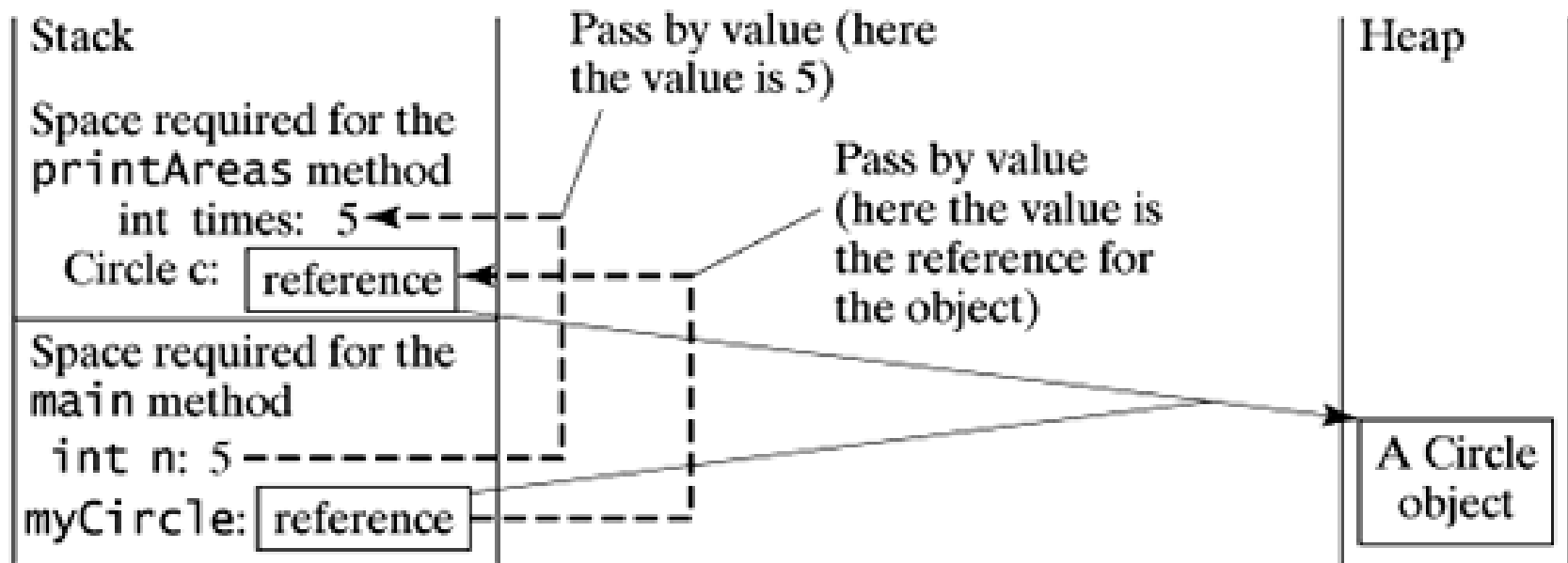
Radius	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	28.274333882308138
4.0	50.26548245743669
5.0	78.53981633974483

Radius is 6.0

n is 5

Passing Objects to Methods

- The figure shows the call stack for executing the methods in the program. Note that the objects are stored in a heap.





The Scope of Variables



The Scope of Variables

- In Methods chapter, discussed local variables and their scope rules.
- Local variables are declared and used inside a method locally.
- This section discusses the scope rules of all the variables in the context of a class.

The Scope of Variables

- **Local variables:**

- A variable defined inside a method is referred to as a local variable.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
- A local variable must be initialized explicitly before it can be used.

The Scope of Variables

- **Instance and static variables:**
 - Instance and static variables in a class are referred to as the **class's variables** or **data fields**.
 - The scope of a class's variables is the entire class, regardless of where the variables are declared.
 - A class's variables and methods can be declared in any order in the class
- You can declare a class's variable only once, but you can declare the same variable name in a method many times in different non-nesting blocks.

The Scope of Variables

- Example:

```
public class Circle {  
    public double find getArea() {  
        return radius * radius * Math.PI;  
    }  
  
    private double radius = 1;  
}
```

(a) variable `radius` and method `getArea()` can be declared in any order

```
public class Foo {  
    private int i;  
    private int j = i + 1;  
}
```

(b) `i` has to be declared before `j` because `j`'s initial value is dependent on `i`.

The Scope of Variables

- If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is hidden.
- An example:
 - [TestScopeOfVariables.java](#)

The Scope of Variables

- As demonstrated in the example, it is easy to make mistakes.
- To avoid confusion, do not declare the same variable name twice in a class, except for method parameters.



Array of Objects



Array of Objects

- Before arrays of primitive type elements were created. You can also create arrays of objects.
- The following statement declares and creates an array of ten `Circle` objects:

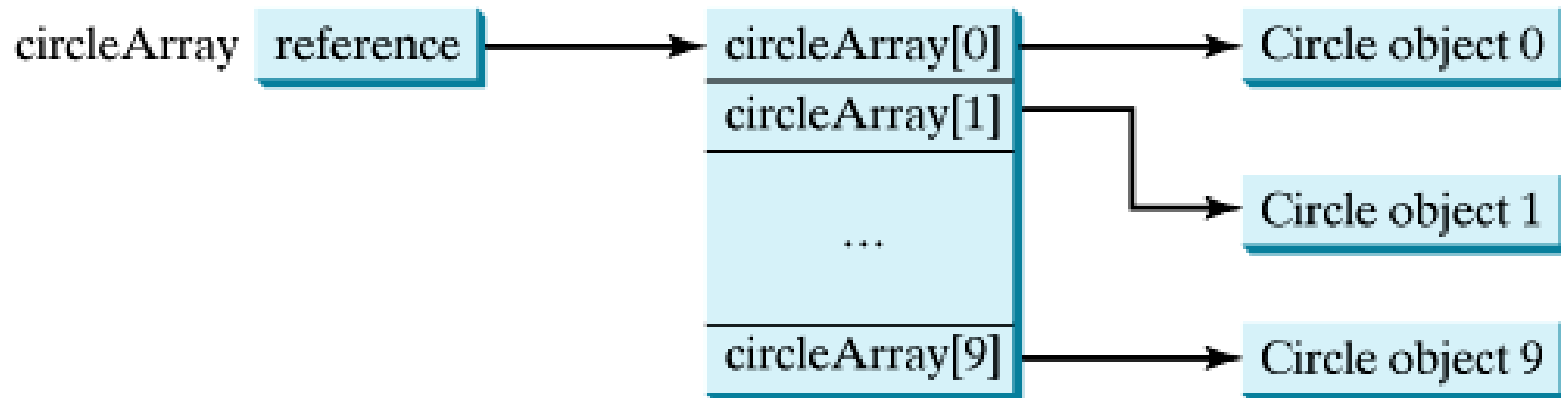
```
Circle[] circleArray = new Circle[10];
```

- To initialize the `circleArray`, you can use a for loop like this one:

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

Array of Objects

- An array of objects is actually an array of reference variables.
- So invoking `circleArray[1].getArea()` involves two levels of referencing



TotalArea.java

- **TotalArea** program summarizes the areas of an array of circles.
- The program creates **circleArray**, an array composed of ten **Circle3** objects
- It then initializes circle radii with random values, and displays the total area of the circles in the array.
- Programs:
 - [Circle3.java](#)
 - [TotalArea.java](#)

TotalArea.java

- The output:

Radius	Area
58.068804279569896	10593.406541297387
36.33710413653297	4148.112246400217
85.02001103760188	22708.695490093254
99.67002343283416	31208.938214899797
68.99814612628313	14956.318906523336
66.51192311899847	13897.890417494793
79.79530733791314	20003.43485868224
11.2738794456952	399.29755019510003
43.04292750675902	5820.408629351761
43.85596734227498	6042.369260300506

The total areas of circles is 129778.8721152384



References



References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition, Pearson Education, 2007. (Chapter 7)
- S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, M. Hoeber, **The Java Tutorial: A Short Course on the Basics**, 4th Edition, Prentice Hall, 2006. (Chapter 4)



The End