Object-Oriented Programming

Creating Java Classes (Cont.)

United International College

Review

- Class
- Object
- Constructor
- Instance Variables
- Methods
- **`this`**

Outline

- **Overloading**
- **Java access modifiers**
- **Keyword: `static`**
- **Packages and `import`**
- **Java API**

Java Method Overloading

- Overloaded methods:
  - appear in the same class;
  - have the **same name**; but
  - have different **parameter lists**; and
  - can have different **return types**.
  - **Method overloading** is also called “ad-hoc polymorphism”.
- Why overload?

Example

```
public class Test {
        public char max(char a, char b) {
                return a > b ? a : b;
        }
        public int max(int a, int b) {
                return a > b ? a : b;
        }
        public double max(double a, double b) {
                return a > b ? a : b;
        }
        public static void main(String[] args) {
                Test t = new Test();
                System.out.println("char: " + t.max('A', 'B'));
                System.out.println("integer: " + t.max(1, 2));
                System.out.println("double: " + t.max(1.0, 2.0));
        }
}
```

Java Constructor Overloading

Add multiple constructors to the class **Person**

```
public class Person {
      private int id;
      private int age;
      public Person() { id = 0; age = 20;}
      public Person(int i) { id = 0; age = i;}
      public Person(int n, int i) { id = n; age = i;}
      public int getId() { return id; }
      public void setId(int id) { this.id = id; }
      public int getAge() { return age; }
      public void setAge(int age) { this.age = age; }
}
```

Java Access Modifiers

- Methods and fields are regulated using access modifiers: **`private`**, **`protected`**, default (no modifier), **`public`**

Modifier	same class	same package	subclass	Other place
`private`	Yes	No	No	No
no modifier	Yes	Yes	No	No
`protected`	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes

- Only **`public`** and no modifier can be used in the **`class`** declaration
  - A public class can be accessed from any place.
  - A no modifier class can only be accessed from within the same package.

Example

```
class AccessTest { // no modifier
        private int i = 1;
        int j = 2; // no modifier
        protected int k = 3;
        public int l = 4;

        private int getI() {
                return i;
        }
        public void m() {
                i = 5;
        }
}
public class Test {
        public static void main(String[] args) {
                AccessTest at = new AccessTest();
                // System.out.println(at.i); // i is private
                at.m(); // m is public
                // System.out.println(at.getI()); // getI() is private
                System.out.println(at.j);
                System.out.println(at.k);
                System.out.println(at.l);
        }
}
```

`public` and `private` Modifiers

- The modifier `public` means that there are no restrictions on where an instance variable or method can be used.
- The modifier `private` means that an instance variable or method cannot be accessed by name outside of the class.
- It is considered good programming practice to make **all instance variables** `private`
- Most methods are `public`
- Usually, methods are `private` only if used as helping methods for other methods in the class
- Use these modifiers to **encapsulate** data (hide from user).

Information Hiding

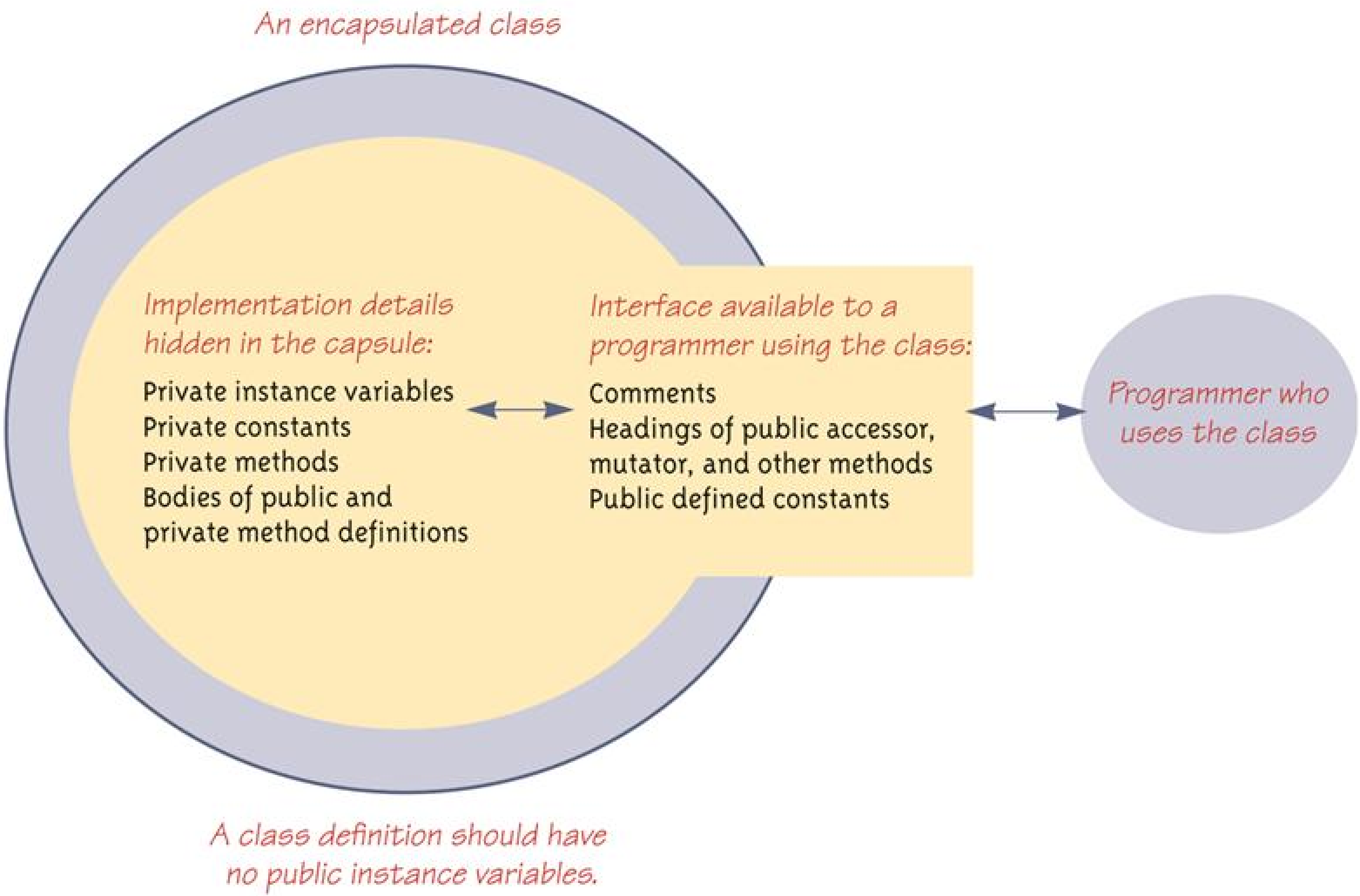
- **Information hiding** is the practice of separating how to use a class from the details of its implementation.
- **Abstraction** is another term used to express the concept of hiding details in order to avoid information overload.

Encapsulation

- **Encapsulation** means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details.
- Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface.
- In Java, hiding details is done by marking them **`private`**

Encapsulation

Encapsulation



An encapsulated class

Implementation details hidden in the capsule:

Private instance variables
Private constants
Private methods
Bodies of public and private method definitions

Interface available to a programmer using the class:

Comments
Headings of public accessor, mutator, and other methods
Public defined constants

Programmer who uses the class

A class definition should have no public instance variables.

`protected` Methods and Variables

- `protected` instance variables or methods can be accessed by:
  - the code of the class itself;
  - subclasses;
  - other code in the same package.
- Weak protection compared to `private`
  - It allows direct access to any programmer who defines a suitable subclass.

Package Methods and Variables

- No-modifier means ***package access***
  - Package access is also known as ***default*** or ***friendly access***
- Instance variables or methods having package access can be accessed by name from code inside the same package.

Accessor and Mutator Methods

- **Accessor (getter)** methods allow the programmer to obtain the value of an object's instance variables.
  - The data can be accessed but not changed.
  - The name of an accessor method typically starts with the word `get`
- **Mutator (setter)** methods allow the programmer to change the value of an object's instance variables *in a controlled manner.*
  - Incoming data is typically tested and/or filtered.
  - The name of a mutator method typically starts with the word `set`

Person Example

- Accessor (getter) method:

```
public String getName() {
    return name;
}
```

- Mutator (setter) method:

```
public void setName(String name) {
    this.name = name;
}
```

- *If you don't want other programs to access your private data, simply do NOT provide these methods.*

Static Methods

- A **static method** is one that can be used without a calling object, it is independent of objects of that class.
- A static method still belongs to a class, and its definition is given inside the class definition.
- A static method can NOT access instance variables (why not?)

Example

```
public class Test {
        // main is static so no Test object is necessary.
        public static void main(String[] args) {
                double i = 3.24;
                double j = 56.2;
                // abs, max, and sqrt are static methods of the
                // Math class so no Math object is necessary.
                System.out.println(Math.abs(i));
                System.out.println(Math.max(i, j));
                System.out.println(Math.sqrt(j));
        }
}
```

Static Methods

- When a static method is defined, the keyword `static` is placed in the method header:

```
public static returnedType myStaticMethod(parameters)
{ . . . }
```

- Static methods are invoked using the class name instead of using the name of an object:

```
returnedValue = MyClass.myStaticMethod(arguments);
```

Static Variables

- A **static variable** is a variable that belongs to the class as a whole, and not just to one object.
- There is only one copy of a static variable per class, unlike instance variables where each object has its own copy.
- All objects of the class can read and change a static variable.
- Although a static method cannot access an instance variable, a static method can access a static variable.
- A static variable is declared like an instance variable, with the addition of the modifier `static`:

```
private static int myStaticVariable;
```

Static Variable

- Accessing static variables
  - Use object reference: `object.staticVariable`
  - Use: `ClassName.staticVariable`

```
Example:
public class Circle {
    static double pi = 3.1415;
}
```

Example

```
public class Cat {
        private static int sId = 1;
        private String name;
        private int id;
        // Each cat automatically gets a new ID number.
        public Cat(String name) {
                this.name = name;
                id = sId++;
        }
        public void info() {
                System.out.println("My name is " + name + " No. " + id);
        }
        public static void main(String arg[]) {
                Cat bob = new Cat("bob");
                bob.info();
                Cat alice = new Cat("alice");
                alice.info();
        }
}
```

Can Static method call nonstatic variable/methods?

- A static method cannot refer to an instance variable of an object of the class, and it cannot invoke a nonstatic method of an object of the class.
- A static method cannot refer to `this`, since it is not part of an object, so it cannot use any instance variable.

Packages

- A **package** is a group of classes that have been placed in a directory or folder.
- A package can be used in any program that includes an **`import`** **statement** that names the package.
- The **`import`** statement must be located at the **beginning** of the program file: only blank lines, comments, and package statements may precede it.
- The program can be in a different directory from the package.

Example: java.util.Scanner

```
import java.util.Scanner; // Scanner class, java.util
package

public class Test {
    public static void main(String[] args) {
        System.out.println("Please enter your answer: ");
        Scanner keyboard = new Scanner(System.in);
        String s = keyboard.next();
        System.out.println("Your answer is: " + s);
    }
}
```

Packages

Example: to import all classes from the package java.util:

```
import java.util.*; // all classes in java.util package
                    // (including Scanner)

public class Test {
   public static void main(String[] args) {
       ...
   }
}
```

Creating Packages

- To make a package, group all the classes together into a single directory (folder), and add the following package statement to the top of each class file:

```
package package_name;
```

- Note: `package` goes in front of `import`:

```
package cst.uic;
import java.util.*;
```

Creating Packages

- **Example**: Suppose we have four classes: `Circle`, `Rectangle`, `Point`, and `Line` that we want to put in a package called “`graphics`”.
- We would add a line of the form

```
package graphics;
```

at the start of each java file.

Creating Packages

```
// In the Circle.java file:
package graphics;
public class Circle { . . . }

// In the Rectangle.java file:
package graphics;
public class Rectangle { . . . }

// In the Point.java file:
package graphics;
public class Point { . . . }

// In the Line.java file:
package graphics;
public class Line { . . . }
```

Naming Packages

- Package names are written in **all lowercase** to avoid conflict with the names of classes.
- Companies use their Internet domain name to begin their package names:
  - E.g.: `package com.oracle.orion;` for a package named `orion` created by a programmer at `oracle.com`.
- Packages in the Java language itself begin with `java` or `javax`.

Java files and packages

- The source code of a public class must be in a `.java` file with the same name:

```
// Must be in the Rectangle.java file:
package graphics;
public class Rectangle { . . . }
```

- Put the source file in a directory whose name is the name of the **`package`** to which the class belongs:

```
NameOfProject\graphics\Rectangle.java
```

How to Use Packages

- Declare the fully-qualified class name:

```
graphics.Rectangle rec = new graphics.Rectangle();
```

- Or use an `import` keyword:

```
import graphics.Rectangle;
Rectangle rec = new Rectangle();
```

Introduction to Java SDK Packages

- `java.lang` - Provides classes that are fundamental to the design of the Java programming language. E.g.: `String`, `Integer`, `Math`, `System` and `Thread`.
- `java.awt` - Contains all of the classes for creating user interfaces and for painting graphics and images.
- `javax.swing` - Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms.
- `java.net` - Provides the classes for implementing networking applications.
- `java.io` - Provides for system input and output through data streams, serialization and the file system
- `java.util` - Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes .

Example

- Study `TestCircle.java`

Summary

- Java overload method
- Modifiers
- Static
- Packages
- Java API