CMPE 101
Object Oriented Programming

İstanbul
Bilgi University

Dr. Emel Küpçü

Department of Computer Engineering

İstanbul Bilgi University

Week-5: Static Variables, Static Methods, Method Overloading

6.3 Static Variable and Static Fields

- Recall that each object of a class maintains its *own* copy of every instance variable of the class.
- **There are variables for which each object of a class does *not* need its own separate copy**.
  - Such variables are declared **static** and are also known as class variables.
- When multiple objects of a class are created, they all share the same copy of the class's static variables, rather than each object having its own separate copy.
- Together a class's **`static` variables** and **instance variables** are known as its fields.

`static` Methods

- Sometimes a method performs a task that does not depend on an object.
  - Applies to the class in which it's declared as a whole
  - Known as a **`static` method** or a **class method**
- It's common for classes to contain convenient `static` methods to perform common tasks.
- To **declare** a method as `static`, place the keyword `static` before the return type in the method's declaration.
  - ```
    static void methodName(ParameterList) {
       // Method body
    }
    ```
  - ```
    static ReturnType methodName(ParameterList) {
       // Method body
       return value; // (if applicable)
    }
    ```
- **Calling** a `static` method
  - *ClassName*`.`*methodName*`(`*arguments*`)`

Static Variable

- A **static variable** belongs to the class rather than to any specific instance. This means that **all instances (objects) of the class share the same static variable.**

```
class Counter {
    static int count = 0;

    Counter() {
        count++;
    }

    static void displayCount() {
        System.out.println("Count: " + count); // Accessing static variable
    }
}

public class Main_Counter {
    public static void main(String[] args) {
        Counter obj1 = new Counter();
        Counter obj2 = new Counter();
        obj1.displayCount();
    }
}
```

Static variable

Constructor

Static method

Output:
Count: 2

Static Method

- A static method is a method that belongs to the class, not an instance of the class.
  - It can be called without creating an object of the class.
  - Static methods can only directly access static variables and other static methods.
- Example:

```
class Calculator {
    static int add(int a, int b) {
        return a + b;
    }
}

public class Main_Calculator {
    public static void main(String[] args) {
        int result = Calculator.add(5, 3);

        System.out.println("Sum: " + result);
    }
}
```

Static Method

Calling Static Method

Output:
Sum: 8

Notes on Declaring and Using Methods (Cont.)

- Non-`static` methods are typically called instance methods.
- A `static` method can call other `static` methods of the same class directly and can manipulate `static` variables in the same class directly.
  - To access the class's instance variables and instance methods, a `static` method must use a reference to an object of the class.

Example- static Variable and Static Method

```
public class StaticExample {

    static int staticVariable = 10; // Static variable
    int instanceVariable = 20; // Instance variable

    // Static method
    static void staticMethod() {
        System.out.println("Static method called.");
        System.out.println("Static variable: " + staticVariable);

        // Cannot directly access instanceVariable or instanceMethod()
        // System.out.println(instanceVariable); // ✗ Compilation error
        // instanceMethod(); // ✗ Compilation error
    }

    // Instance method
    void instanceMethod() {
        System.out.println("Instance method called.");
        System.out.println("Instance variable: " + instanceVariable);
    }

    public static void main(String[] args) {
        // Calling static method directly
        staticMethod();

        // Creating an object to access instance variables and
methods
        StaticExample obj = new StaticExample();
        System.out.println("Accessing instance variable: " +
obj.instanceVariable);
        obj.instanceMethod(); // Calling instance method through
an object
    }
}
```

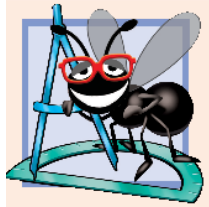
`static` Methods, `static` Fields and Class `Math` (Cont.)

`Math` Class Methods

- Class `Math` provides a collection of `static` methods that enable you to perform common mathematical calculations.
- Method arguments may be constants, variables or expressions.

Example:

```
// Using Math static methods
double squareRoot = Math.sqrt(25); // Calculates √25 = 5
double power = Math.pow(2, 3); // Calculates 2^3 = 8
```



Software Engineering Observation 6.4

Class `Math` is part of the `java.lang` package, which is implicitly imported by the compiler, so it's not necessary to import class `Math` to use its methods.

Static Methods, static Fields and Class Math (Cont.)

Method	Description	Example

`abs(`*x*`)`	absolute value of *x*	`abs(23.7)` is `23.7`
		`abs(0.0)` is `0.0`
		`abs(-23.7)` is `23.7`

`ceil(`*x*`)`	rounds *x* to the smallest integer not less than *x*	`ceil(9.2)` is `10.0` `ceil(-9.8)` is `-9.0`
`cos(`*x*`)`	trigonometric cosine of *x* (*x* in radians)	`cos(0.0)` is `1.0`
`exp(`*x*`)`	exponential method e^x	`exp(1.0)` is `2.71828` `exp(2.0)` is `7.38906`
`floor(`*x*`)`	rounds *x* to the largest integer not greater than *x*	`floor(9.2)` is `9.0` `floor(-9.8)` is `-10.0`

Fig. 6.2 | `Math` class methods. (Part 1 of 2.)

Static Methods, static Fields and Class Math (Cont.)

Method	Description	Example
`log(`*x*`)`	natural logarithm of *x* (base *e*)	`log(Math.E)` is `1.0` `log(Math.E * Math.E)` is `2.0`
`max(`*x*`, `*y*`)`	larger value of *x* and *y*	`max(2.3, 12.7)` is `12.7` `max(-2.3, -12.7)` is `-2.3`
`min(`*x*`, `*y*`)`	smaller value of *x* and *y*	`min(2.3, 12.7)` is `2.3` `min(-2.3, -12.7)` is `-12.7`
`pow(`*x*`, `*y*`)`	*x* raised to the power *y* (i.e., x^y)	`pow(2.0, 7.0)` is `128.0` `pow(9.0, 0.5)` is `3.0`
`sin(`*x*`)`	trigonometric sine of *x* (*x* in radians)	`sin(0.0)` is `0.0`
`sqrt(`*x*`)`	square root of *x*	`sqrt(900.0)` is `30.0`
`tan(`*x*`)`	trigonometric tangent of *x* (*x* in radians)	`tan(0.0)` is `0.0`

Fig. 6.2 | `Math` class methods. (Part 2 of 2.)

`static` Methods, `static` Fields and Class `Math` (Cont.)

`Math` Class `static` Constants `PI` and `E`

- `Math` fields for commonly used mathematical constants
  - `Math.PI` (3.141592653589793) // Used to calculate the circumference of a circle (2 * π * r).
  - `Math.E` (2.718281828459045) //Used in an exponential calculation
  - (e^x using Math.pow()).
- Declared in class `Math` with the modifiers `public`, `final` and `static`
  - `public` allows you to use these fields in your own classes.
  - A field declared with keyword `final` is *constant*—its value cannot change after the field is initialized.

`static` Methods, `static` Fields and Class `Math` (Cont.)

Why is method `main` declared `static`?

- The JVM calls the main method of the specified class when a program starts. At this point, no objects of the class exist. Declaring main as static allows the JVM to execute it without needing to create an instance of the class.

- **public class ClassName {**

```
    public static void main(String[] args) {
     // The body of the main method
    }
}
```

String concatenation

Assembling Strings with String Concatenation

- String concatenation
  - In Java, string concatenation allows us to assemble multiple Strings into a larger String using:
  - `The + operator`
  - `The += operator`
  - When both operands of operator + are `Strings`, operator + creates a new `String` object
  - Characters of the right operand are placed at the end of those in the left operand
- In Java, any primitive type (e.g., int, double, boolean) can be converted into a String and concatenated.
- When one of the + operator's operands is a `String`, the other is converted to a `String`, then the two are concatenated.

Example – String Concatenation

```
public class StringConcatTypes {
    public static void main(String[] args) {
        int age = 25;
        double height = 5.9;
        boolean isStudent = true;

        // Concatenating primitives with Strings using +
        String message = "Age: " + age + ", Height: " + height + ", Student: " + isStudent;
        System.out.println(message);

        // Using += to append more information
        message += ", Status: Graduated";
        System.out.println(message);

    }
}
```

Output:
Age: 25, Height: 5.9, Student: true
Age: 25, Height: 5.9, Student: true, Status: Graduated

Common Programming Error 6.3

Confusing the + operator used for string concatenation with the + operator used for addition can lead to strange results. Java evaluates the operands of an operator from left to right. For example, if integer variable y has the value 5, the expression `"y + 2 = " + y + 2` results in the string `"y + 2 = 52"`, not `"y + 2 = 7"`, because first the value of y (5) is concatenated to the string `"y + 2 = "`, then the value 2 is concatenated to the new larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the desired result `"y + 2 = 7"`.

6.12 Method Overloading

- Method overloading
  - **Methods of the same name** declared in the same class
  - **Must have different sets of parameters**
- Compiler selects the appropriate method to call by examining the **number**, **types** and **order of the arguments** in the call.
- Used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.
- Overloading improves code readability and flexibility.

```
1  // Fig. 6.10: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload {
5     // test overloaded square methods
6     public static void main(String[] args) {
7        System.out.printf("Square of integer 7 is %d%n", square(7));
8        System.out.printf("Square of double 7.5 is %f%n", square(7.5));
9     }
10
11    // square method with int argument
12    public static int square(int intValue) {
13       System.out.printf("%nCalled square with int argument: %d%n",
14          intValue);
15       return intValue * intValue;
16    }
17
18    // square method with double argument
19    public static double square(double doubleValue) {
20       System.out.printf("%nCalled square with double argument: %f%n",
21          doubleValue);
22       return doubleValue * doubleValue;
23    }
24 }
```

Fig. 6.10 | Overloaded method declarations. (Part 1 of 2.)

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.10 | Overloaded method declarations. (Part 2 of 2.)

Distinguishing Between Overloaded Methods

- The compiler differentiates overloaded methods based on:
  - **Method name** (must be the same)
  - **Number of parameters**
  - **Types of parameters**
  - **Order of parameters**

```
class OverloadExample {
   // Overloaded methods with different parameter lists
   void print(int a) {
      System.out.println("Integer: " + a);
   }

   void print(double a) {
      System.out.println("Double: " + a);
   }

   void print(int a, int b) {
      System.out.println("Two Integers: " + a + " and " + b);
   }

   public static void main(String[] args) {
      OverloadExample obj = new OverloadExample();
      obj.print(5);       // Calls print(int)
      obj.print(5.5);     // Calls print(double)
      obj.print(3, 7);    // Calls print(int, int)
   }
}
```

Output:
Integer: 5
Double: 5.5
Two Integers: 3 and 7

Distinguishing Between Overloaded Methods

- Return Types in Overloaded Methods
  - Return types are NOT considered for method overloading!
  - Methods cannot have the same parameter list but different return types!
  - Example:

```
class AB {
   int add(int a, int b) {
      return a + b;
   }

   double add(int a, int b) {
      return a + b;
   }
}
```

Error Reason: The compiler cannot distinguish the methods based on return type alone.

Questions?