

Programming 1

Lecture 5 – Switch, More about String, Static method

Contents

- The **switch** statement
- **String** operations
- Overview of methods

The `switch` statement



- The `switch` statement is often used to replace an `if...else` chain. Here is an example:

```
int n = sc.nextInt(); // read from keyboard
if (n == 1) {
    System.out.println("1st");
} else if (n == 2) {
    System.out.println("2nd");
} else if (n == 3) {
    System.out.println("3rd");
} else {
    System.out.println(n + "th");
}
```

The `switch` statement

- This `if...else` chain is used to specify the program's behavior for many cases of `n`.
- The first block whose condition returns a true value gets executed.

```
int n = sc.nextInt(); // read from keyboard
if (n == 1) {
    System.out.println("1st");
} else if (n == 2) {
    System.out.println("2nd");
} else if (n == 3) {
    System.out.println("3rd");
} else {
    System.out.println(n + "th");
}
```

The `switch` statement

- If no boolean expression evaluates to true, the `else` block will get executed.

```
int n = sc.nextInt(); // read from keyboard
if (n == 1) {
    System.out.println("1st");
} else if (n == 2) {
    System.out.println("2nd");
} else if (n == 3) {
    System.out.println("3rd");
} else {
    System.out.println(n + "th");
}
```

The `switch` statement

- The equivalent `switch` statement of that `if...else` chain.

```
int n = sc.nextInt(); // read from keyboard
switch (n) {
    case 1:
        System.out.println("1st");
        break;
    case 2:
        System.out.println("2nd");
        break;
    case 3:
        System.out.println("3rd");
        break;
    default:
        System.out.println(n + "th");
}
```

The `switch` statement

- However, the `switch` statement **does not** work the same way as an `if...else` chain.
- To see the difference, let's remove all `break` statements.

```
int n = 2;
switch (n) {
    case 1:
        System.out.println("1st");
    case 2:
        System.out.println("2nd");
    case 3:
        System.out.println("3rd");
    default:
        System.out.println(n + "th");
}
```

The `switch` statement

```
int n = 2;
switch (n) {
    case 1:
        System.out.println("1st");
    case 2:
        System.out.println("2nd");
    case 3:
        System.out.println("3rd");
    default:
        System.out.println(n + "th");
}
```

Output:

2nd
3rd
2th

But why?

`n` is 2 so shouldn't it print out only "2nd"?

The `switch` statement

```
int n = 2;
switch (n) {
    case 1:
        System.out.println("1st");
    case 2:
        System.out.println("2nd");
    case 3:
        System.out.println("3rd");
    default:
        System.out.println(n + "th");
}
```

Output:

2nd
3rd
2th

It turns out that it is called a **switch** for this reason.



The `switch` statement

```
int n = 2;
switch (n) {
    case 1:
        System.out.println("1st");
    case 2:
        System.out.println("2nd");
    case 3:
        System.out.println("3rd");
    default:
        System.out.println(n + "th");
}
```

Output:

2nd
3rd
2th

**The first case that is equal to `n` turns the switch on.
From that point, every statement is executed regardless of case.**

The `switch` statement

```
int n = 2; // a hard-coded value
switch (n) {
    case 1:
        System.out.println("1st");
        break;
    case 2:
        System.out.println("2nd");
        break;
    case 3:
        System.out.println("3rd");
        break;
    default:
        System.out.println(n + "th");
}
```

A `break` statement can prevent other cases to be executed.

The `break` statement breaks out of `switch`, just like what it does to a loop.

Part 3

More about String

Escaping special characters

- The backslash `\` is the escape character.
 - Some special characters must be escaped in strings.

Sequence	Meaning
<code>\t</code>	Tab character
<code>\b</code>	Backspace character, will remove a previous character
<code>\"</code>	The double quote
<code>\n</code>	Newline character
<code>\\</code>	The backslash character itself

"She said \"Hello!\" to me."

She said "Hello" to me.

"I'm gonna take\b\b\b\bgive your\b money."

?

"C:\\Temp\\Secret.txt"

C:\Temp\Secret.txt

String comparison

- Use **String.equals()** method to compare 2 Strings (case-sensitive)

```
String s = "Hello";  
boolean b1 = s.equals("Hello"); // true  
boolean b2 = s.equals("hello"); // false
```

- Use **String.equalsIgnoreCase()** method to compare 2 Strings (case-insensitive)

```
String s = "Hello";  
boolean b1 = s.equalsIgnoreCase("Hello"); // true  
boolean b2 = s.equalsIgnoreCase("hello"); // true
```

Find a String within another String

- **s1.indexOf(s2)** returns the position of String **s2** in String **s1**.

```
String s1 = "Welcome to FIT";  
int pos = s1.indexOf("come");  
System.out.println(pos);  
System.out.println(s1.indexOf("greeting"));  
System.out.println(s1.indexOf("fit"));  
System.out.println(s1.indexOf("Wel"));
```

Output:

```
3  
-1  
-1  
0
```

More String operations

Method name	Description
<code>startsWith(prefix)</code>	Tests if a string starts with the specified prefix.
<code>endsWith(suffix)</code>	Tests if a string ends with the specified suffix.

```
boolean b = "Hello".startsWith("He");
```

true

```
boolean b = "Hello".endsWith("abc");
```

false

More String operations

Method name	Description
<code>compareTo(str)</code>	Compares two strings alphabetically.

```
String s1 = "Apple";  
String s2 = "Banana";  
String s3 = "Cars";  
int order = s1.compareTo(s2); // -1, s1 < s2 by 1  
int order2 = s1.compareTo(s3); // -2, s1 < s3 by 2  
int order3 = s3.compareTo(s1); // 2, s3 > s1 by 2
```

Here, the comparison is based on the first character.

More String operations

Method name	Description
<code>trim()</code>	Returns a copy of the string, with leading and trailing whitespaces omitted.

```
String s = "    Hi, dad!    ";  
System.out.println(s + " I'm back!");  
System.out.println(s.trim() + " I'm back!");
```

Output:

```
    Hi, dad!    I'm back!  
Hi, dad! I'm back!
```

Java methods

- **Definition:** A method is a block of code which only runs when it is called.

Methods in a Java program

```
public class MyApp {  
    public static int getPositiveInt() {  
        Scanner sc = new Scanner(System.in);  
        int x = 0;  
        do {  
            try {  
                x = sc.nextInt();  
            } catch (Exception e) {  
                sc.nextLine();  
            }  
        } while (x <= 0);  
        return x;  
    }  
    public static void main(String[] args) {  
        int a = getPositiveInt();  
        System.out.println(a);  
    }  
}
```

About Java methods

- You can write your own methods!
- Writing a method is like inventing a new Java statement yourself!
- Methods in Java are similar to **functions** in other languages (similar, not the same).
- Divide a complex program into smaller modules
 - Much easier to debug each part than an entire big program.
 - Better organize your program, making development easier.
 - If a piece of code is used multiple times, putting it in a method results in shorter source code.

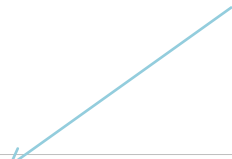
Method execution

- A method only runs when it is called.
 - Methods are called by their names.
 - A method can be called multiple times.
- The `main` method is an exception. It runs when the program starts.
 - Inside the `main` method, we can write statements to call other methods.

Using non-static methods

sc is an instance of class Scanner

Correct use:



```
Scanner sc = new Scanner(System.in);  
int x = sc.nextInt();
```

Incorrect use:

```
int x = Scanner.nextInt();
```

Notes on the methods we use

- Some familiar static methods:
 - `Math.sqrt()`, `Math.pow()`, `Integer.parseInt()`
- A `static` method is called from the **name** of the class that contains it.
- Some familiar non-static methods:
 - `Scanner.nextInt()`, `String.substring()`, `String.charAt()`...
- A non-static method is called from an **instance** of the class that contains it.

Notes on the methods we use

- Within a class, methods can be called using only the method name (without class name or instance).
 - In a `static` method (or `static context`), we can only directly call other `static` methods.
 - In a non-static method, we can call both static and non-static methods by name.
- The `main` method is `static`, so it can only call other `static` methods within the same class.
 - That's why, for the time being, for simplicity, all methods that we create will be `static`.

Return

- To run/trigger a method, write its name like so:

```
getPositiveInt(); // as a statement
int a = getPositiveInt(); // as an expression
System.out.println(getPositiveInt()); // expression
int b = 5 + getPositiveInt(); // expression
```

- Each time a method name appears is a **method call**.
- A **method call** makes two things happen:
 - All the statements inside the method are executed.
 - The **method call** itself is an **expression**. An **expression** is essentially a **value**. This **value** is specified inside the method with the **return** statement.

Return

- Consider an example:

```
public class MyApp {  
    public static int getPositiveInt() {  
        return 51;  
    }  
    public static void main(String[] args) {  
        System.out.println(getPositiveInt());  
    }  
}
```

Output:

51

So, `getPositiveInt()` is an `int` with value `51`

Producing method output

- A method usually produces a **value**.
 - With the exception of **void** methods which should not return a value.
- The output is produced with a **return** statement.

```
public class MyApp {  
    public static int getPositiveInt() {  
        return 51;  
    }  
    public static void main(String[] args) {  
        System.out.println(getPositiveInt());  
    }  
}
```

Method return type

- The data type of the `getPositiveInt()` method is `int`

```
public class MyApp {  
    public static int getPositiveInt() {  
        return 51;  
    }  
    public static void main(String[] args) {  
        System.out.println(getPositiveInt());  
    }  
}
```

- The value in the `return` statement must have the same data type as the method's. If you don't want a method to return a value, use the `void` type for your method.

Method return type

- A `void` method cannot contain a return statement with value, but can contain an empty return statement:

```
public static void printMenu() {  
    System.out.println("1. Option A");  
    System.out.println("2. Option B");  
    System.out.println("3. Option C");  
    return;  
}
```

- A `return` statement stops (terminates) the method.
- Any statement after the `return` statement will never be executed. The compiler will notify you if it discovers unreachable statements after a `return` statement.

Dataflow **into** and **out of** a method

- From where? By whom? To where?
- If there is a sender, there must be a receiver.

Method call (sender)

```
int result = gcd(5, 15);  
System.out.println(result);
```

Method (receiver)

```
public static int gcd(int a, int b) {  
    int c;  
    while (b != 0) {  
        c = a;  
        a = b;  
        b = c % b;  
    }  
    return a;  
}
```

Passing parameters to a method

- Parameters are specified after the method name, inside the parentheses.
 - Multiple parameters are separated by commas.
- Parameters act as variables inside the method.

```
static void printMax(int x, int y) {  
    if (x > y) {  
        System.out.println(x);  
    } else {  
        System.out.println(y);  
    }  
}
```


Method I/O

- A Java method receives inputs and produces an output.
- Inputs are received through parameters.
 - The list of parameters is given within a pair of parentheses (...) in the method header.
 - The list of parameters can be empty.
- The output is produced with the **return** statement.
 - With the exception of **void** methods, which do not produce output.