Lecture 5 — Control Statements

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

Control Statements

- We've seen `if` statements
- `if (`*`boolean-expression`*`)` *`statement-or-block`*
- They are a kind of *control* statement
- Control statements modify the flow of a program
- Without control statements, all Java programs would be executed line-by-line
- Let us refresh ourselves on the `if` statement

if statements

```
public class If {
    public static void main(String[] args) {
        double x = 2.3;
        if (x < 4/2.0) {
            System.out.println("Not executed");
        }
        if (x < 5/2.0) {
            System.out.println("Executed");
        }
    }
}
```

- The code inside an `if` statement is executed only when the condition *expression* is true
- Otherwise the program skips past that code and starts again on the next line after the `if` block

else

```
public class Else {
    public static void main(String[] args) {
        double x = 2.3;
        if (x < 4/2.0) {
            // Needs braces if there is more than one statement
            System.out.println("Not executed 1");
            x = 0;
        } else {
            System.out.println("Executed 1");
        }
        if (x < 5/2.0)
            System.out.println("Executed 2");
        else
            System.out.println("Not executed 2");
    }
}
```

- The `if` code is executed if the condition is true and then the `else` block is skipped
- Otherwise the `else` block is executed

if-else-if ladder

```
public class IfElseIfElse {
    public static void main(String[] args) {
        double x = 2.3;
        if (x < 4/2.0)
            System.out.println("x < 4/2");
        else if (x < 5/2.0)
            System.out.println("4/2 <= x < 5/2");
        else if (x < 6/2.0)
            System.out.println("5/2 <= x < 6/2");
        else
            System.out.println("x >= 6/2");
    }
}
```

- A full `if` statement can be followed by any number of `else if` statements
- Then, 0 or 1 `else` statements
- This is the `if-else-if` ladder and it is executed in order
- Once a single clause is executed, the program skips to the end

Nested ifs

```
public class NestedIf {
    public static void main(String[] args) {
        boolean x = true, y = false;
        if (x) {
            if (y) {
                System.out.println("x && y");
            } else {
                System.out.println("x && !y");
            }
        }
    }
}
```

- if statements can be nested
- A refresher on logical operators!

Scanner

```
public class Scanner {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = sc.next();
        System.out.println("Hello, " + name);
    }
}
```

- Using `args` allows some degree of user input
- Reading input from the terminal allows much more
- We will do this using a `class` from Java's API: `Scanner`
- We saw this in a previous lecture
- We create a new instance of a class using the `new` keyword

Scanner

```
import java.util.Scanner; // Allows us to use short-name Scanner
public class NestedIf2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // Note how these are initialised to in order
        boolean x = sc.nextBoolean(), y = sc.nextBoolean();
        if (x) {
            if (y) {
                System.out.println("x && y");
            } else {
                System.out.println("x && !y");
            }
        }
    }
}
```

- We can make our previous nested `if` example interactive

switch statements

```
switch (expression) {
    case constant1:
        statement sequence
        break;
    case constant2:
        statement sequence
        break;
    ...
    default:
        statement sequence
}
```

- switch statements are control statements like if statements
- They allow multiway branching
- Sometimes a better alternative to having lots of else if cases

switch statements

```
import java.util.Scanner;

public class Switch {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        switch (num) {
            case 1:
                System.out.println("One");
                break;
            case 2:
                System.out.println("Two");
                break;
            case 3:
                System.out.println("Three");
                break;
            default:
                System.out.println("Other");
        }
    }
}
```

`switch` statements

- `switch` statements jump to the case that matches to the expression
- Then, they execute lines of code until they hit a break
- If there is no matching case, they go to `default`
- The expression needs to be `byte`, `short`, `int`, `char`, an *Enumeration*, or a `String`
- We will learn about *Enumerations* later, but they are just fancy `int`s

switch statements

```
import java.util.Scanner;

public class Switch2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a word: ");
        String word = scanner.next();
        switch (word) {
            case "hello":
                System.out.println("Hello to you too!");
            case "goodbye":
                System.out.println("Goodbye!");
                break;
            default:
                System.out.println("I don't understand.");
        }
    }
}
```

- Missing `break` statements can cause bugs

switch statements

```java
import java.util.Scanner;

public class Switch3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        // Count of numbers >= num
        int countGtEq = 0;
        // Leaving out break can be useful for grouping cases together
        switch (num) {
            case 1:
                countGtEq++;
            case 2:
                countGtEq++;
            case 3:
                countGtEq++;
            case 4:
                countGtEq++;
            // default is optional
        }
        System.out.println("countGtEq=" + countGtEq);
    }
}
```

`switch` statements

- Why would you use a `switch` instead of an `if`?
- They are faster; `if` will check each case in order
- `if` statements are more flexible
- `switch` can only handle branching on a single expression to specific values
- A `switch` will struggle with conditions like `if (x > y/2)`
- How would you map this to specific values?

Mid-lecture break

while loops

```
import java.util.Scanner;

public class While {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        while (num > 0) {
            System.out.println("num=" + num);
            num--;
        }
    }
}
```

- As mentioned in a previous lecture, `while` loops are another control statement
- They work like an `if` that loops

break

```
import java.util.Scanner;

public class Break {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("Are you ready for this program to end? Enter true
                or false: ");
            boolean answer = scanner.nextBoolean();
            if (answer)
                break;
        }
    }
}
```

- `break` works in loops as well as `switch`
- It immediately exits the loop and continues execution after the loop

do-while loops

```
import java.util.Scanner;

public class DoWhile {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        boolean answer; // Note that this variable is declared outside the
            loop.
        do {
            System.out.print("Are you ready for this program to end? Enter true
                or false: ");
            answer = scanner.nextBoolean();
        } while (!answer);
    }
}
```

- Here we achieve the same thing with a `do-while` loop
- `do-while` always do at least one iteration

while loops again

```
import java.util.Scanner;

public class PowerOf2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int num = scanner.nextInt();
        int pow = 1;
        while (pow <= num) {
            pow *= 2;
        }
        System.out.println("The first power of 2 greater than your number is "
            + pow);
    }
}
```

- `while` loops are great when we want to iterate until some condition is achieved

for loops

```
public class For {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.println("i=" + i);
        }
    }
}
```

- `for` loops are often more convenient than `while` loops
- `for (initialisation; condition; iteration)`
- These three components happen before the loop starts; before each iteration; at the end of each iteration

for loops

```
public class ForWhile {
    public static void main(String[] args) {
        int i = 1;
        while (i <=10) {
            System.out.println("while i=" + i);
            i++;
        }
        // No need to redeclare i
        // We can omit {} since it's only one statement
        for (i = 1; i <= 10; i++)
            System.out.println("for i=" + i);
    }
}
```

- `for` loops compress a complex `while` onto a single line
- It is useful to realise that they are actually the same thing

Empty statements

```
public class EmptyStatement {
    public static void main(String[] args) {
        ;;; // Empty statements
        int x = 10;
        for (x = 0; x < 3; x++);
        System.out.println("x=" + x);
    }
}
```

- A semicolon alone ; is seen as an empty statement
- This allows us to write loops without any body

Empty statements

```
import java.util.Scanner;
public class EmptyForLoop {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int x = sc.nextInt();
        for (; x > 0; --x)
            System.out.println("x=" + x);
    }
}
```

- Empty statements can be used to omit parts of the `for` loop
- A minimal infinite loop: `for (;;);`

for loops and Arrays

```
public class ForArray {
    public static void main(String[] args) {
        int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int i = 0; i < nums.length; i++) {
            nums[i] *= 5;
        }
        for (int i = 0; i < nums.length; i++) {
            System.out.println("nums[" + i + "]=" + nums[i]);
        }
    }
}
```

- We will learn more about arrays in later lectures
- `for` loops are well suited to looping over array entries

for-each loops

```
public class ForEach {
    public static void main(String[] args) {
        int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int i = 0; i < nums.length; i++) {
            nums[i] *= 5;
        }
        for (int num : nums) {
            System.out.println(num);
        }
    }
}
```

- The `for-each` loop is a convenient way to loop over a collection
- We will see more of these later when we look at arrays and Java collections in general