

Department of Computer Science

# Conditional Execution & Patterns

Dr. Chia-Ling Tsai

## Outline

- The `if-else` statement
- The `switch` statement
- Comparing values
- Text processing
- Preconditions and postconditions
- Patterns

## The `if - else` Statement

- Conditional statements
- The simple `if` statement
- The `if-else` statement

## Conditional Statements

- Computers can be programmed to make *decisions*
- A decision is based on a *logical test*, or *condition* – an expression that evaluates to true or false
- In Java, a decision is programmed using a *conditional statement*
- The principal conditional statement in Java is called the `if` statement, because of the way it is written

## The Simple `if` Statement

- This statement is given a condition and an action
  - It checks the condition
  - If the condition is true, it performs the action
  - Otherwise, it does nothing, and the execution of the program continues
- Example

```
if (withDrawalAmount > balance)
    System.out.println("Account overdrawn");
```

- Follows as in English: If the amount being withdrawn is larger than the balance, then a message is printed that the account is overdrawn

## Statements Within Statements

- As in the program, a loop may be inside a conditional statement
  - Or vice versa
- Each of the `for` and `while` loops and the conditional `if` includes a block of statements
- Each of those statements may be *any* statement, including a `for`, `while`, or `if`
  - Each of which contains a block, which may contain *any* statement.

## Sample Program

```
import java.util.Scanner;

public class SumFinder1
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("This program will add any number of integers");
        System.out.print("How many integers will you enter? ");
        int number = keyboard.nextInt();

        int sum = 0;
        for (int i = 1; i <= number; i++)
        {
            System.out.print("Enter integer #" + i + ": ");
            int next = keyboard.nextInt();
            sum = sum + next;
        }
        System.out.println("Thank you.");
        System.out.println("The sum of the " + number + " integers is " + sum);
    }
}
```

What if user enters 0  
or -1?

```
This program will add any number of integers
How many integers will you enter? 0
Thank you.
The sum of the 0 integers is 0
Press any key to continue . . . _
```

```
This program will add any number of integers
How many integers will you enter? -1
Thank you.
The sum of the -1 integers is 0
Press any key to continue . . . _
```

## Sample Program Improved

```
import java.util.Scanner;

public class SumFinder1a
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("This program will add any number of integers");
        System.out.print("How many integers will you enter? ");
        int number = keyboard.nextInt();

        if (number < 0)
        {
            System.out.println("You can't enter " + number + " integers!");
        }

        int sum = 0;
        for (int i = 1; i <= number; i++)
        {
            System.out.print("Enter integer #" + i + ": ");
            int next = keyboard.nextInt();
            sum = sum + next;
        }
        System.out.println("Thank you.");
        System.out.println("The sum of the " + number + " integers is " + sum);
    }
}
```

```
This program will add any number of integers
How many integers will you enter? -1
You can't enter -1 integers!
Thank you.
The sum of the -1 integers is 0
```

## The `if-else` statement

- This statement is given a condition and *two* actions
  - If the condition is true, it performs the first action
  - Otherwise, it performs the second action

- Example

```

if (withDrawalAmount > balance)
{
    System.out.println("Account overdrawn");
}
else
{
    balance = balance - withDrawalAmount;
}

```

- Follows as in English: If the amount is larger than the balance, then a message is printed that the account is overdrawn. Otherwise, the amount is subtracted from the balance

## Sample Program Improved More

```

import java.util.Scanner;

public class SumFinder2
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("This program will add any number of integers");
        System.out.print("How many integers will you enter? ");
        int number = keyboard.nextInt();

        if (number < 0)
        {
            System.out.println("You can't enter " + number + " integers!");
        }
        else
        {
            int sum = 0;
            for (int i = 1; i <= number; i++)
            {
                System.out.print("Enter integer #" + i + ": ");
                int next = keyboard.nextInt();
                sum = sum + next;
            }
            System.out.println("Thank you.");
            System.out.println("The sum of the " + number + " integers is " + sum);
        }
    }
}

```

## Output

- Ordinary input

```
This program will add any number of integers
How many integers will you enter? 5
Enter integer #1: 123
Enter integer #2: 456
Enter integer #3: 789
Enter integer #4: 321
Enter integer #5: 654
Thank you.
The sum of the 5 integers is 2343
```

- Bad input

```
This program will add any number of integers
How many integers will you enter? -1
You can't enter -1 integers!
```

## Multiple Alternatives

- For more than two alternatives, you use *nested if statements*
- The safest way to nest statements is to use the form

```
if (condition1)
{
    statement-when-condition1-true
}
else if (condition2)
{
    statement-when-condition2-true
}
...
else
{
    statement-when -none-of-conditions-true
}
```

## Sample Program

```

public class SumFinder3
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("This program will add any number of integers");
        System.out.print("How many integers will you enter? ");
        int number = keyboard.nextInt();

        if (number < 0)
        {
            System.out.println("You can't enter " + number + " integers!");
        }
        else if (number == 0)
        {
            System.out.println("Well, if you have no numbers, no need for the program!");
            System.out.println("(The sum, by the way, is 0.)");
        }
        else
        {
            int sum = 0;
            for (int i = 1; i <= number; i++)
            {
                System.out.print("Enter integer #" + i + ": ");
                int next = keyboard.nextInt();
                sum = sum + next;
            }
            System.out.println("Thank you.");
            System.out.println("The sum of the " + number + " integers is " + sum);
        }
    }
}

```

## Output

### ■ Ordinary input

```

This program will add any number of integers
How many integers will you enter? 5
Enter integer #1: 123
Enter integer #2: 456
Enter integer #3: 789
Enter integer #4: 321
Enter integer #5: 654
Thank you.
The sum of the 5 integers is 2343

```

### ■ Input zero

```

This program will add any number of integers
How many integers will you enter? 0
Well, if you have no numbers, no need for the program!
(The sum, by the way, is 0.)

```

### ■ Bad input

```

This program will add any number of integers
How many integers will you enter? -1
You can't enter -1 integers!

```

## Even Better

```

public static void main(String[] args)
{
    Scanner keyboard = new Scanner(System.in);

    System.out.println("This program will add any number of integers");
    System.out.print("How many integers will you enter? ");
    int number = keyboard.nextInt();

    // Check input
    if (number > 0) ← Put most typical situation first
    {
        // Get numbers and find sum
        // Initialize accumulator
        int sum = 0;
        for (int i = 1; i <= number; i++)
        {
            // Get next item
            System.out.print("Enter integer #" + i + ": ");
            int next = keyboard.nextInt();

            // Accumulate
            sum = sum + next;
        }
        // Use correct accumulator value
        System.out.println("Thank you.");
        System.out.println("The sum of the " + number + " integers is " + sum);
    }
    else if (number < 0)
    {
        // Complain about negative input
        System.out.println("You said you would enter a negative number (" + number + ") of integers.");
        System.out.println("Please run the program again, and use a positive number.");
    }
    else //number == 0
    {
        System.out.println("You said you would enter zero integers, so the sum is 0.");
        System.out.println("If you meant to enter a positive number, please run the program again.");
    }
}

```

Be nice!

## Avoiding Complex Logic

- Keep in mind that the `else` part of the `if - else` executes only when the `if` part is not true
- Sometimes this fact can simplify things
- This is especially true for nested `if` statements, where each `else` is executed only when *all* the conditions above it are false
- If you decide instead to use complex conditions, it is easy to miss cases – try always to have a plain `else` at the end

## Sample 1

```

if (average >= 90 && average <= 100)
{
    grade = "A";
}
else if (average >= 80 && average < 90)
{
    grade = "B";
}
else if (average >= 70 && average < 80)
{
    grade = "C";
}
else if (average >= 60 && average < 70)
{
    grade = "D";
}
else if (average >= 0 && average < 60)
{
    grade = "E";
}
else
{
    grade = "not computable -- bad average of " + average;
}

```

## Sample 2

```

if (average > 100 || average < 0)
{
    grade = " not computable -- bad average of " + average;
}
else if (average >= 90)
{
    grade = "A";
}
else if (average >= 80)
{
    grade = "B";
}
else if (average >= 70)
{
    grade = "C";
}
else if (average >= 60)
{
    grade = "D";
}
else
{
    grade = "E";
}

```

## The `switch` statement

- Similar to `if-else`, but allowing you to choose from many statements based on an integer.
- Always include a `default` to catch bugs.
- Can always be replaced by `if-else`.

```
import java.util.Scanner;
//Just a silly program to demonstrate switch statement
public class Switch {
    public static void main(String[] args) {
        System.out.println("Please enter a number");
        Scanner keyboard = new Scanner(System.in);
        int day = keyboard.nextInt();
        switch (day) {
            case 1: System.out.println("Today is Monday");
                break;
            case 2: System.out.println("Today is Tuesday");
                break;
            default: System.out.println("Is it Friday?");
        }
    }
}
```

Error-prone!

## Comparing Values

- Comparing numerical values—revisited
- Floating-point numbers
- Comparing strings for equality
- Other string comparisons
- Comparing other objects

## Comparing Numerical Values—revisited

- In forming conditions, *comparisons* are often used

| Java | Math | Description           |
|------|------|-----------------------|
| >    | >    | Greater than          |
| >=   | ≥    | Greater than or equal |
| <    | <    | Less than             |
| <=   | ≤    | Less than or equal    |
| ==   | =    | Equal                 |
| !=   | ≠    | Not equal             |

## Floating-point Numbers

- Because floating-point numbers may be stored inaccurately, you must be careful when comparing them
- It is usually not a good idea to use == for comparison
- When you make a choice based on whether or not two floating-point numbers are equal, you may be surprised because of this inaccuracy

## Sample Program

```
public class FloatingPointPrecisionExample
{
    public static void main(String[] args)
    {
        double r = Math.sqrt(2);
        System.out.println("The square root of 2 is " + r);
        double rSquared = r * r;
        if (rSquared == 2.0)
        {
            System.out.println("OK: " + r + " * " + r + " = 2.0");
        }
        else
        {
            System.out.println("Oops: " + r + " * " + r + " = " + rSquared);
        }
    }
}
```

```
The square root of 2 is 1.4142135623730951
Oops: 1.4142135623730951 * 1.4142135623730951 = 2.0000000000000004
```

## Comparing Floating-Point Numbers

- When you want to compare floating-point numbers, *decide how close they have to be to be considered "the same"*
- Example
  - You might consider two numbers to be the same if they are the same in the first 7 decimal places
  - This will happen if they *differ by less than* 0.00000005 (i.e., 5 in the *eighth* decimal place)
- Mathematicians often denote a test value like this using the Greek letter  $\epsilon$  (epsilon)

## Sample Program

```

public class FloatingPointComparisonExample
{
    static final double EPSILON = 0.00000005; ← Criterion for "equality"

    public static void main(String[] args)
    {
        double r = Math.sqrt(2);
        System.out.printf("The square root of 2 is %.7f\n", r);
        double rSquared = r * r;

        double absoluteDifference = Math.abs(rSquared - 2.0); } Test
        if (absoluteDifference < EPSILON)
        {
            System.out.printf("OK: %.7f * %.7f = %.7f\n", r, r, rSquared);
        }
        else
        {
            System.out.printf("Oops: %.7f * %.7f = %.7f\n", r, r, rSquared);
        }
    }
}

```

Use formatted printing

```

The square root of 2 is 1.4142135623730951
OK: 1.4142135623730951 * 1.4142135623730951 = 2.0

```

## Comparing Objects

- Objects can be compared using `==`, but it is *almost never* a good idea
- If you use `==`, you are asking if the objects are *identical*
  - More precisely, when you ask if two *variables, which are object references*, are equal using `==`, you are asking if they *refer to the same object*
- There is a method, `equals`, which asks if the objects are the same from the point of view of *contents*
  - More precisely, if the two variables, though they may refer to different objects, refer to objects whose *attributes* are equal

## Sample Program

```

public class ComparingPoints
{
    public static void main(String[] args)
    {
        Point p1 = new Point(100, 200);
        Point p2 = p1;
        Point p3 = new Point(100, 200);
        Point p4 = new Point(100, 300);

        comparePoints(p1, p2);
        comparePoints(p1, p3);
        comparePoints(p1, p4);
    }

    static void comparePoints(Point first, Point second)
    {
        if (first == second)
        {
            System.out.println("The two points are identical");
        }
        else if (first.equals(second))
        {
            System.out.println("The two points have the same coordinates");
        }
        else
        {
            System.out.println("The two points are different");
        }
    }
}

```

```

The two points are identical
The two points have the same coordinates
The two points are different

```

## Text Processing

- Text is made of *strings*
- The String class has a number of methods that are helpful in *processing* text
  - Changing all letters to upper or lower case
  - Replacing each occurrence of one string of a string by another
  - Creating substrings – using part of a string
  - Making comparisons – do two strings
    - have the same contents?
    - have the same contents except for *case*?
    - compare with respect to *order*?

## Changing Case And Replacing

- Changing case is simple
  - `toUpperCase` replaces each lower case letter with the corresponding upper case letter
  - Similarly, `toLowerCase` converts to lower case
- The `replace` method has two arguments
  - The first argument is a part of the string to replace
  - The second argument is what to replace it with
  - All occurrences of the first are replaced by the second

## Sample Program

```
public class StringManipulationSample
{
    public static void main(String[] args)
    {
        String alphabet = "Abcdefg Hijklmnop Qrstuv Wxyz";
        String gibberish = "AZXD WZXR TZXJ PZXL";

        // Original information
        System.out.println("Two strings:");
        System.out.println(alphabet);
        System.out.println(gibberish);

        // Manipulate strings
        String changedAlphabet = alphabet.toUpperCase();
        changedAlphabet = changedAlphabet.replace(" ", "|");

        String changedGibberish = gibberish.toLowerCase();
        changedGibberish = changedGibberish.replace("zx", "IJK");

        // New information
        System.out.println("The two strings, changed:");
        System.out.println(changedAlphabet);
        System.out.println(changedGibberish);
    }
}
```

```
Two strings:
Abcdefg Hijklmnop Qrstuv Wxyz
AZXD WZXR TZXJ PZXL
The two strings, changed:
ABCDEFGH|IJKLMNOP|QRSTU|WXYZ
aIJKd wIJKr tIJKj pIJKl
```

## Substrings

- The method `substring` allows you to set one string to equal part of another
- You specify the part you want using the idea of an *index*
  - Each character in the string is numbered *starting from 0*
  - You give the index of the first character
  - You give the index of the character *following* the last character
- Example: If `s = "Your password is "`  
`s.substring(5,9)` **is** `"pass"`

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| Y | o | u | r |   | p | a | s | s | w | o  | r  | d  |    | i  | s  |    |

## Comparing Strings for Equality

- When you ask if strings are equal using `==`, you are *not* asking if the *text* is the same, but if the *objects* are *identical*
- This is (almost) *never* what you want
- Instead, use the method `equals` to test for equality of strings
- When you use this method, strings are considered equal if they look the same
  - This is usually what you want

## Sample Program

```
String string1 = "Iona Gaels";
String string2 = "Iona";
String string3 = string1.substring(0,4); // "Same as" string2
```

```
Using ==
string2(Iona) and string3(Iona) are not the same
string2(Iona) and Iona are the same
string3(Iona) and Iona are not the same
Using equals
string2(Iona) and string3(Iona) are the same
string2(Iona) and Iona are the same
string3(Iona) and Iona are the same
```

## Other String Comparisons

- *Lexicographic* comparison of strings

```
string1.compareTo(string2)
```

- Depends on "collating sequence"

- Not quite alphabetical order: all capitals before all lowercase, and includes numbers and punctuation

- Answer is a *number*

- Actual value doesn't matter – just whether negative, zero, or positive
- Negative means `string1` comes before `string2`
- Zero means `string1` equals `string2`
- Positive means `string1` comes after `string2`

- Case-insensitive comparison of strings

- Use `equalsIgnoreCase` instead of `equals`

## Preconditions And Postconditions

- The concepts
- The `assert` statement
- Writing a method with assertions

## The Concepts

- When a program operates, you make certain assumptions
- This is especially true for methods
  - Each method is written using certain assumptions, called *preconditions*
  - When a method is called, it is the *responsibility of the caller* to be sure that the preconditions are met
  - The writer of the method *assumes* the preconditions are met
- Assumptions about what a method *does* are called *postconditions*

## The `assert` Statement

- For a quick check of preconditions, you can use the `assert` statement
- This statement tells Java to check a precondition, and to stop the program with an error if the condition is not true
- For real-world, user-oriented programs, this approach is not good enough – but we'll use it to help us write methods that check their preconditions
- The idea is to state the preconditions as comments, then check them with `assert` statements

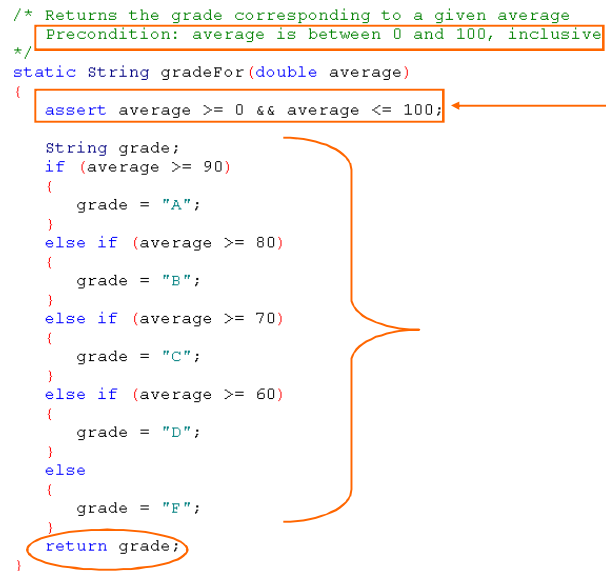
## Writing A Method With Assertions

- When you write a method, *don't look at the program that calls it*
  - Imagine that this method will be called many times, in many programs
  - Library methods, such as those in `Math`, work this way
- Clearly state the method's preconditions
- Use `assert` to check the preconditions
- There is a way to "turn off" assertions when you are sure they will be met – we won't worry about that

## Example

```
/* Returns the grade corresponding to a given average
 * Precondition: average is between 0 and 100, inclusive
 */
static String gradeFor(double average)
{
    assert average >= 0 && average <= 100;

    String grade;
    if (average >= 90)
    {
        grade = "A";
    }
    else if (average >= 80)
    {
        grade = "B";
    }
    else if (average >= 70)
    {
        grade = "C";
    }
    else if (average >= 60)
    {
        grade = "D";
    }
    else
    {
        grade = "F";
    }
    return grade;
}
```



## Patterns

- Looking for patterns
- Accumulator pattern
- Fencepost pattern

## Looking for Patterns

- As you program, *patterns* appear – you wind up doing very similar things in several programs
- If you identify the pattern, you can use it again
  - This is *almost* the same as reusing part of a program
  - The code is not quite the same, but the general organization is, and the algorithm is very close
- As examples, we look at two loop patterns
  - Accumulator
  - Fencepost

## Accumulator Pattern

- To add a sequence of numbers input by the user
  - You have to get the numbers one at a time
  - But you don't want to give each number a name
- So you keep a *running total* or *cumulative sum*
  - Initialize sum to zero
  - Loop
    - Get next number
    - Add it to the sum
  - When loop ends, sum is correct for all entered numbers

## Example

```
import java.util.Scanner;

public class CumulativeSum
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("This program will add a sequence of integers you enter");
        System.out.print("How many integers do you want to add? ");
        int count = keyboard.nextInt();

        int sum = 0;
        for (int i = 1; i <= count; i++)
        {
            System.out.print("Next integer (item #" + i + "): ");
            int nextItem = keyboard.nextInt();
            sum = sum + nextItem;
        }
        System.out.println("The sum of the " + count + " integers is " + sum);
    }
}
```

Initialize sum to 0  
 Loop:  
 get next number  
 add it to sum  
 After loop: sum is correct

## New Example – Same Pattern

- Multiply a sequence of numbers
- Changes in pattern details
  - Initialize *product* to 1
  - *Multiply* product by next number
  - When loop ends, *product* is correct
- Real pattern is the *accumulator pattern*
  - Initialize *accumulator*
  - Loop
    - Get next number
    - Perform operation to *accumulate* next number
  - When loop ends, *accumulator has correct value* for all entered numbers

## The Program

```

Initialize accumulator
Loop
  Get next number
  Perform operation to accumulate next number
After loop: accumulator has correct value
import java.util.Scanner;

public class CumulativeProduct
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("This program will multiply a sequence of integers you enter");
        System.out.print("How many integers do you want to multiply? ");
        int count = keyboard.nextInt();

        int product = 1;
        for (int i = 1; i <= count; i++)
        {
            System.out.print("Next integer (item #" + i + "): ");
            int nextItem = keyboard.nextInt();
            product = product * nextItem;
        }
        System.out.println("The product of the " + count + " integers is " + product);
    }
}

```

## Fencepost Pattern

- A "fencepost" loop pattern is one that resembles a fencepost design



- To print a fence in outline, we would use a loop
- But it's hard to get the loop right

- Desired output

```

|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

```

(Post on either end, rails between posts)

## Attempts – Off-By-One-Errors

```

for (int i = 1; i <= numberOfPosts; i++)
{
    post();
    rail();
}
System.out.println();

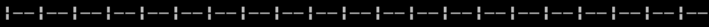
```

```

static void post()
{
    System.out.print("|");
}

static void rail()
{
    System.out.print("---");
}

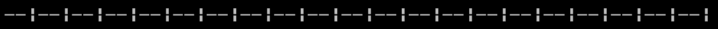
```



```

for (int i = 1; i <= numberOfPosts; i++)
{
    rail();
    post();
}

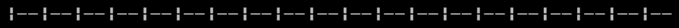
```



```

for (int i = 1; i <= numberOfPosts - 1; i++)
{
    post();
    rail();
}
System.out.println();

```

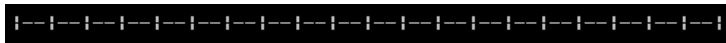


## Getting It Right – And Seeing A Pattern

```

post();
for (int i = 1; i <= numberOfPosts - 1; i++)
{
    rail();
    post();
}
System.out.println();

```



### Pattern

- First *post* *outside* loop
- Loop repeats *one less time* than number of posts (i.e., number of rails)
- In loop, first print *rail*, then *post*

## New Example – Same Pattern

- User chooses a count, then you print that many randomly selected numbers on one line, separated by commas

- We'll use integers between 0 and 99, inclusive
- Example: If the user chooses **5**, the output might be

```
87, 34, 57, 36, 31
```

- Here

- The "posts" are the numbers
- The "rails" are the commas (and following blanks)

## The Program

- Pattern:

- Post (Get and print a random number)
- Repeat one less time than number of posts:
  - Rail (Print a comma and a space)
  - Post (Get and print a random number)

```
// post
int nextNumber = randomizer.nextInt(100);
System.out.print(nextNumber);
for (int i = 1; i <= count - 1; i++)
{
    // rail
    System.out.print(", ");
    // post
    nextNumber = randomizer.nextInt(100);
    System.out.print(nextNumber);
}
System.out.println();
```