Introduction (1)

- Suppose we want to write a program to be used in cash registers in stores to compute the amount of money a customer must pay for his / her purchases
- The program prompts the cashier to enter the price of the next item, and then asks whether there are more items to be processed
- We know how to execute different statements depending on whether there are more items to be processed
- However, we do not know in advance how many items each customer will buy

Introduction (2)

- We could assume that no customer will ever buy more than a certain number of items, but that has certain disadvantages
  - The program is not very robust: no matter how high we set this threshold, someone could go over it
  - Do we really want to copy and paste the same code fragment 10 times? 20 times? 100 times? More?
- We need a way to repeat a (set of) statement(s) as long as some condition holds

Repetition Statements

- Repetition statements allow us to execute a (set of) statement(s) multiple times, repetitively; these statements are often called loops
- One execution of the (set of) statement(s) that is repeated by a repetition statement is called an iteration
- Like conditional statements, loops are controlled by conditions expressed as boolean expressions
- The (set of) statement(s) which is executed repetitively by the repetition statement is called the loop body
Repetition Statements in Java

- Java has three kinds of repetition statements: the **while loop**, the **do loop**, and the **for loop**
- These statements are equivalent
  - That is, one can repeat any (set of) statement(s) using any of these three statements
- However, in certain circumstances, choosing one of these statements over the other results in a clearer program
  - That is, one that is, easier for humans to read and understand

Part 1: The **while** Statement

The **while** Statement

- The **while** statement has the following syntax:
  
  ```java
  while ( condition )
  statement;
  ```

  while is a reserved word in Java

  Just like in `if`/`if-else` statements, **condition** MUST be:
  - a boolean expression; therefore, it must evaluate to either true or false
  - surrounded by parentheses

  If **condition** evaluates to true, then **statement** is executed and **condition** is evaluated again
  
  **statement** is executed repetitively until **condition** evaluates to false

Execution of the **while** Loop

- If the condition of a **while** loop evaluates to false initially (that is, it evaluates to false the first time it is evaluated), the statement is never executed
- Therefore, the body of a **while** loop will execute zero or more times
Logic of a **while** Statement

- **Condition** evaluated
  - false
  - true
- **Statement**
  - (rest of the program)

Statements in a Loop Body

- The statement(s) executed repeatedly by a loop (including a **while** loop) can be of any type:
  - Assignment statements
  - Method calls
  - Conditional statements (**if**, **if-else**, **switch**)
  - Loops (**while**, **do-while**, **for**)
  - Block statements

Block Statements and Loops

- Remember that a block statement (a set of statements delimited by `{`}) can be used wherever a single statement is called for in the Java syntax
- Therefore, a block statement can be used as the body of a **while** loop
- This enables us to write loops whose bodies consist of multiple statements instead of just one statement
  - Each of these statements can be of any type (assignment statement, method call, conditional statement, loop, ...)
- If you want to put more than one statement in a loop body, you *must* group these statements inside a block statement
  - Otherwise, you will get a semantic (logical or run-time) error

```java
public class Counter {
    public static void main(String[] args) {
        final int LIMIT = 3;
        int count = 1;
        while (count <= LIMIT) {
            System.out.println(count);
            count = count + 1;
        }
        System.out.println("Done.");
    }
}
```

What does this display?
Counter.java

```java
public class Counter {
    public static void main(String[] args) {
        final int LIMIT = 3;
        int count = 1;
        while (count <= LIMIT) {
            System.out.println(count);
            count = count + 1;
        }
        System.out.println("Done.");
    }
}
```

What does this display?

<table>
<thead>
<tr>
<th>count:</th>
<th>LIMIT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Console:

1
2
3
4

Done.

Infinite Loops

- The statements in the body of a `while` loop must eventually make the condition evaluate to `false`
- If not, it is an *infinite loop*, which will execute until the user interrupts (terminates) the program
- This is a common type of logical error
- You should always double check to ensure that your loops will terminate normally

Abyss.java

```java
public class Abyss {
    public static void main(String[] args) {
        int count = 1;
        System.out.println("I'm going in...");
        while (count <= Integer.MAX_VALUE) {
            System.out.println(count);
            count = count - 1;
        }
        System.out.println("Found the bottom of the abyss!");
    }
}
```

What is wrong here?

Block Statements vs. Indentation

- What will happen if the following code fragment is executed:
  ```java
  while (a < b) {
      System.out.println(a);
      a = a + 1;
  }
  ```
  - The value of `a` will be printed repeatedly, it will never change, and the loop will never stop
  - Once again, *syntax* determines whether a statement belongs to the body of a loop
    - Indentation has *nothing* to do with this
    - Advice: Always use block statements as loop bodies, even when the block statement contains only one statement
Off-By-One Errors

• The condition controlling a loop must be designed in such a way that the loop body is executed exactly as many times as necessary.
• However, it is a common logical error to write loop conditions that result in the loop body being executed one time too few, or one time too many.
• These are called off-by-one errors.
• You should always check that your loop conditions do not cause such errors.

Sum.java

```java
public class Sum {
    public static void main(String[] args) {
        final int MAX = 5;
        int sum = 0;
        int i = 1;
        while(i < MAX) {
            sum = sum + i;
            System.out.println("Sum: " + sum);
            i = i + 1;
            System.out.println("i: " + i);
            System.out.println();
        }
        System.out.println("The sum of the integers from 1 to " + MAX + " is: " + sum);
    }
}
```

What will be displayed? Why is it the wrong result?

AnotherSum.java

```java
public class AnotherSum {
    public static void main(String[] argv) {
        final int MAX = 5;
        int i = 0;
        int sum = 0;
        while(i <= MAX) {
            i = i + 1;
            System.out.println("i: " + i);
            sum = sum + i;
            System.out.println("sum: " + sum);
            System.out.println();
        }
        System.out.println("The sum of the integers from 1 to " + MAX + " is: " + sum);
    }
}
```

What will be displayed? Why is it the wrong result?

while Loop: Exercise 1

• Complete the `main()` method of the `BusPercentage` class by adding code that does the following:
  – Prompts the user to enter the number of passengers that want to board a bus.
  – Checks whether the number is greater than or equal to 0, and less than or equal to the number of seats on the bus.
    • If the number of passengers entered by the user is less than 0, then the completed program **MUST** display an appropriate message and prompt the user to enter again.
    • If the number of passengers entered by the user is greater than the capacity of the bus, the completed program **MUST** display an appropriate message and prompt the user to enter again.
  – Repeats the previous step until the user enters a valid number, then displays the percentage of occupied seats on the bus.
**BusPercentage.java**

```java
import java.util.Scanner;

public class BusPercentage {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        final int NUM_SEATS = 56;
        // Add your code here
    }
}
```

**while Loop: Exercise 2**

- Complete the `main()` method of the `BusSpeed` class by adding code that computes the total distance travelled by a bus during a trip, as well as the average speed of the bus during the trip
  - The user will enter the distance travelled during each hour of the trip individually
  - The user will not enter the number of hours travelled by the bus; instead, the completed program will calculate this value by counting the number of distance values entered by the user
  - The user will indicate that the trip is over by entering a negative value
  - After the user has entered the distance travelled in an hour, the completed program **MUST** display the total distance travelled up to that point

**BusSpeed.java**

```java
import java.util.Scanner;

public class BusSpeed {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        // Add your code here
    }
}
```

**Nested Loops**

- Like `if` and `if-else` statements, loops can be nested as well
- That is, the body of a loop could itself contain another loop
  - The loop contained in the body of another is called the *inner loop*
  - The loop which contains another loop in its body is called the *outer loop*
- Every time the body of the outer loop is executed, the inner loop will go through its entire set of iterations
NestedLoop.java

```java
public class NestedLoop {
    public static void main(String[] args) {
        final int MAX = 4;
        int outerCount, innerCount;
        outerCount = 1;
        while (outerCount <= MAX) {
            innerCount = 1;
            while (innerCount <= MAX) {
                System.out.print("(" + outerCount + "," + innerCount + ") ");
                innerCount++;
            }
            System.out.println();
            outerCount++;
        }
        System.out.println("Done.");
    }
}
```

What does this display?

AnotherNestedLoop.java

```java
public class AnotherNestedLoop {
    public static void main(String[] args) {
        final int MAX = 4;
        int outerCount, innerCount;
        outerCount = 1;
        while (outerCount <= MAX) {
            innerCount = 1;
            while (innerCount <= outerCount) {
                System.out.print("(" + outerCount + "," + innerCount + ") ");
                innerCount++;
            }
            System.out.println();
            outerCount++;
        }
        System.out.println("Done.");
    }
}
```

What does this display?

Nested Loops: Exercise

- Complete the `main()` method of the `PrimeFilter` class to add code that does the following:
  - Checks whether every number between 2 and the value entered by the user (inclusive) is a prime number
  - When the completed program determines that a number is prime, it MUST display it
  - The completed program MUST count how many prime numbers exist between 2 and the value entered by the user (inclusive), and display the total number of primes in that range

PrimeFilter.java

```java
import java.util.Scanner;

public class PrimeFilter {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        int upperBound;
        System.out.println("This program displays the prime numbers that exist between 2 and an upper limit.");
        System.out.println("It also counts them.");
        System.out.println("Enter the upper limit: ");
        upperBound = keyboard.nextInt();
        // Add your code here
    }
}
```
Part 2: The **do-while** Statement

**The do-while Statement**

- The *do-while* statement has the following syntax:
  
  ```
  do 
  statement; 
  while ( condition ); 
  ```

  - Uses both the `do` and `while` reserved words
  - `statement` is executed once initially; `condition` is evaluated only after `statement` has been executed at least once
  - `statement` is repetitively executed until `condition` evaluates to `false`
  - *do-while* loops *always* execute their bodies at least once

**Logic of a do-while Statement**

- **while vs. do-while Loops**
  - A *do-while* loop is similar to a *while* loop, except that the condition is evaluated after the body of the loop is executed
  - Therefore the body of a *do-while* loop will be executed at least once, whereas the body of a *while* loop may not be executed at all
### While Loops / do-while Loops

- **while loop:**
  - **Condition:** evaluated
  - **Statement:**
  - **False condition:**
  - (rest of the program)

- **do-while loop:**
  - **Condition:** evaluated
  - **Statement:**
  - **False condition:**
  - (rest of the program)

### AnotherCounter.java

```java
public class AnotherCounter {
    public static void main(String[] args) {
        final int LIMIT = 3;
        int count = 0;
        do {
            count = count + 1;
            System.out.println(count);
        } while (count < LIMIT);
        System.out.println("Done.");
    }
}
```

What does this display?

#### Console:

```
1
2
3
Done.
```

### do-while Loop: Exercise

- Complete the `main()` method of the `ReverseNumber` class by adding code that produces an integer value which consists of the digits of the number entered by the user, but in the reverse order.
  - For example, if the user enters the number 12345, the completed program MUST produce the number 54321.
  - In addition, the completed program MUST treat both the number entered by the user, and the number it produces, as integer values.
  - Finally, the completed program MUST use a do-while loop to reverse the number.
  - Can you modify the completed program so that it determines whether the number entered by the user is palindromic (that is, whether it reads the same way forwards and backwards)?
ReverseNumber.java

```java
import java.util.Scanner;

public class ReverseNumber {
    public static void main (String[] args) {
        Scanner keyboard = new Scanner (System.in);
        int number;
        System.out.print("Enter a positive integer: ");
        number = keyboard.nextInt();
        // Add your code here
    }
}
```

Part 3: The for Statement

The for Statement

- The for statement has the following syntax:

  ```
  for (initialization; condition; increment)
  statement;
  ```

  - The `initialization` portion is executed before the loop body is entered.
  - The `statement` is executed until the `condition` evaluates to `false`.
  - The `increment` portion is executed at the end of each iteration of the loop.

Logic of a for Statement
for Loops as while Loops

A for loop is equivalent to the following while loop structure:

```
initialization;
while (condition) {
    statement;
    increment;
}
```

Execution of a for Loop

Like in a while loop, the condition of a for loop is tested prior to entering the loop body.

- If the condition of a for loop evaluates to false initially (that is, it evaluates to false the first time it is evaluated), the statement is never executed.
- Therefore, the body of a for loop will be executed zero or more times.
- A for loop is well suited for executing a specific number of times that can be determined in advance.

```java
public class YetAnotherCounter {
    public static void main(String[] args) {
        final int LIMIT = 3;
        for (int count=1; count <= LIMIT; count++) {
            System.out.println(count);
        }
        System.out.println("Done.");
    }
}
```

What does this display?

```
count: 1 2 3 4
Console: 1 2 3 Done.
```

```java
public class YetAnotherCounter {
    public static void main(String[] args) {
        final int LIMIT = 3;
        for (int count=1; count <= LIMIT; count++) {
            System.out.println(count);
        }
        System.out.println("Done.");
    }
}
```

Limit: 3

What does this display?

Limit: 3

```
count: XXXX
```

```
Console:
1
2
3
Done.
```
**for Loop: Exercise 1**

- Complete the `main()` method of the `Multiples` class by adding code that displays all the multiples of a number entered by the user that exist between 1 and an upper limit also entered by the user
  - The completed program **MUST** display 5 multiples of the number entered by the user per line, except for the last line, which may contain less
  - In addition, the completed program **MUST** display a tab character (`	`) between every multiple displayed on a line
  - Finally, the completed program **MUST** use a `for` loop to generate and display the multiples of the number entered by the user

```java
import java.util.Scanner;

public class Multiples {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        final int PER_LINE = 5;
        int value;
        int limit;
        System.out.print("Enter a positive value: ");
        value = keyboard.nextInt();
        System.out.print("Enter an upper limit: ");
        limit = keyboard.nextInt();
        System.out.println("The multiples of " + value + " between " + value + " and " + limit + " (inclusive) are:");
        // Continued on next slide
    }
}
```

**Multiples.java (1 / 2)**

```java
// Continued from previous slide
// Add your code here
}
```

**Multiples.java (2 / 2)**

```
// Continued from previous slide
// Add your code here
}
```

**for Loop: Exercise 2**

- Complete the `main()` method of the `Approximation` class by adding code that approximates the number $e$ (the basis of the natural logarithm). The number $e$ can be approximated by the following sum: $e = (1 / 0!) + (1 / 1!) + (1 / 2!) + (1 / 3!) + ...$
  - The `main()` method of the `Approximation` class asks the user to enter the number of terms of be computed; your task is to add code which computes each of these terms and adds them together to form the approximation of $e$
  - The obvious way of solving this problem involves using nested loops, where the inner loop computes the required factorial during every iteration of the outer loop. Can you find a way to solve this problem using only one loop?
Approximation.java

```java
import java.util.Scanner;

public class Approximation {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        int n;
        System.out.print("Please enter the number of terms: ");
        n = keyboard.nextInt();
        // Add your code here
    }
}
```

Stars.java

```java
public class Stars {
    public static void main(String[] args) {
        final int MAX_ROWS = 10;
        for (int row = 1; row <= MAX_ROWS; row++) {
            for (int star = 1; star <= row; star++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

WhatIsThis.java (1 / 2)

```java
public class WhatIsThis {
    public static void main(String[] args) {
        final int MAX_ROWS = 10;
        for (int row = 1; row <= MAX_ROWS; row++) {
            for (int space = 1; space <= MAX_ROWS - row; space++) {
                System.out.print(" ");
            }
            for (int star = 1; star <= row * 2; star++) {
                System.out.print("*");
            }
            System.out.println();
        }
        for (int base = 3; base > 0; base--) {
            for (int space = 1; space <= MAX_ROWS-1; space++) {
                System.out.print(" ");
            }
            System.out.println("***");
        }
    }
}
```

WhatIsThis.java (2 / 2)

```
// Continued from previous slide
System.out.println("***");
```
Details on the for Statement

• Each expression in the header of a for loop is optional
  – If the initialization portion is left out, no initialization is performed
  – If the condition portion is left out, it is always considered to evaluate to true, and therefore creates an infinite loop
  – If the increment portion is left out, no increment operation is performed
• Both semi-colons are always required in the for loop header

Part 4: Top-Down Development

Top-Down Program Development

• Top-down development is a way of thinking when you try to solve a programming problem
  – It involves starting with the entire problem, and breaking it down into more manageable pieces
  – Each of these pieces can then be broken down again into smaller pieces
  – You then solve each of these pieces, and combine the solutions to these pieces to form the solution to the original problem

Top-Down Development: Steps

• Applying top-down development consists of doing the following:
  1. First, read and understand the problem
  2. Then, subdivide the problem into pieces. Each chunk is one task, like initializing variables, getting inputs, generating outputs, if-statements and loops
  3. Then sort all these elements in the correct order.
  4. Last, only now start writing your code
• Steps 1-3 are either done in your head or on scrap paper. They are not done using the language editor or compiler. They do not even need to be in Java, they could be in simple English / French / your first language / ...
Part 5: Exercises

Exercises (1)

1. Write a program which consists of a single class called `Factor` that asks the user to enter a positive integer (including zero). The program then displays that number and its greatest prime factor. The program repetitively does this until the user inputs a negative number.

Exercises (2)

2. Write a program which consists of single class called `Boxes` that asks the user for a positive integer number \( n \). The program then displays a solid square with side length \( n \) next to a hollow square with side length \( n \). The program does nothing if the user inputs 0 or a negative value. If example, if the user enters 4, the following will be displayed:

```
**** ****
**** *  *
**** *  *
**** ****
```