1 The while statement

Used to iterate through a sequence of statements a number of times.

Syntax:

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

- The `statement(s)` while be executed as long as the `expression` remains `true`.

Example:

```java
import java.util.*;

public class WhileExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int i = 0;

        while (i >= 0) {
            System.out.print("Please enter a positive number: ");
            i = sc.nextInt();
            System.out.println("You entered: "+i);
        }

        System.out.println("Bye...");
    }
}
```
The above program will prompt the user to enter a positive number. As long as the user inputs a positive number (or zero) the code displays the number. As soon as a negative number is entered, the program exits with a “bye” message.

2 The for loop

Syntax:

for (initialisation; test; step)
statement

where initialisation, test, step are expressions which control the loop. Any of these can be omitted.

1. The initialisation expression initialises some variables to some values.

2. The test expression is evaluated.
   (a) if true, the statement is executed.
   (b) if false, the loop terminates and the control of the program is transferred to the first statement after the loop.

3. The step expression is evaluated. This typically increases or decreases the value(s) of the variable(s) involved in Step 1.

4. Step 2 is executed.

Example:

Find all the numbers which are less than 100 and are divided exactly by 17.

```java
public class ForExample {
    boolean isDivisibleBy17(int number) {
        if ((number % 17) == 0)
            return true;
        else
            return false;
    }

    public static void main(String[] args) {
        // instantiate an object of class ForExample
        ForExample fe = new ForExample();

        for (int i=0; i <= 100; i++) {
            boolean result = fe.isDivisibleBy17(i);
            if (result)
```
When the above program is run, it displays:

0 is divisible by 17!
17 is divisible by 17!
34 is divisible by 17!
51 is divisible by 17!
68 is divisible by 17!
85 is divisible by 17!

3 The do-while statement

Another loop statement used as an alternative to while and for.

Syntax:

```
  do
    statement
  while (expression);
```

- The `statement(s)` will be executed as long as the `expression` remains `true`.
- The loop will execute at least once! If the `expression` is `false` in the first iteration, the `statement(s)` in the loop will already have been executed once.

Any code using a for, or while loop, can be expressed with the other form of the loop. For example, a loop written with a for can be rewritten using a while. However, do-while loops are not equivalent to for and while loops because the body of a do-while loop is always executed at least once, while the body of a for or while loop might not be executed at all.

Example:

Write a class which provides a method to calculate the number of digits in a given integer

```java
import java.util.Scanner;

class Counter {
    // count and return the number of digits in n
    int countDigits(int n) {
        int numDigits = 0;
        do {
            numDigits++;
            n = n / 10;
        } while (n != 0);
        return numDigits;
    }
}
```
numDigits++;
    n = n/10;
    }
while (n > 0);
    return numDigits;
}

public class DoExample {
    public static void main(String[] args) {
        // create an object of class Counter
        Counter c = new Counter();

        // get user input
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();

        // count how many digits there are in m
        int numDigits = c.countDigits(m);
        System.out.println(m + " contains " + numDigits + " digits.");
    }
}

4 The break and continue keywords

The break and continue can occur within loops (break can occur within a switch statement too).

Similarly with C++:

- A break causes the innermost enclosing loop, or switch to exit.
- When a continue statement is executed, the flow of control is passed to the next iteration of the innermost loop.

Example:

public class BreakContinueTest {
    public static void main(String[] args) {
        for (int i=1; i<=20; i++) {
            if (i == 5)
                continue;
            if (i == 7)
                break;
            System.out.println(i);
        }
    }
}
When the above example is run, it displays:

1
2
3
4
6

5 Overloading Methods

A class can have more than one methods with the same name, assuming that these methods have a different signature.

- The signature of a method consists of the combination of its name and its arguments (the specific order, number and type of arguments)

Note that both ordinary methods and constructors can be overloaded.

Example:

```java
public class Printer {
    int errorCode;

    // constructor 1
    public Printer() {
        System.out.println("Constructor with no arguments called!");
    }

    // constructor 2
    public Printer(int i) {
        errorCode = i;
        System.out.println("Constructor with an int argument called!");
    }

    public void display() {
        System.out.println(errorCode);
    }

    public void display(int i) {
        System.out.println(i);
    }
}
```
```java
public void display(String s) {
    System.out.println(s);
}

public void display(int i, String s) {
    System.out.println(i + s);
}

public void display(String s, int i) {
    System.out.println(i + s);
}
```

Note that the order of arguments is part of the signature of a method. Thus, both `display(int, String)` and `display(String, int)` can be present in class Printer because they are methods with different signatures (and the same name, i.e. `display` is overloaded).

## 6 Reading messages from thrown exceptions

Exceptions are thrown at runtime, when something unexpected happens. Exceptions will be discussed in detail in a future lecture.

```java
import java.util.Scanner;

public class ReadingExceptions {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Please enter an integer: ");
        int i = s.nextInt();
        System.out.println("You entered " + i);
    }
}
```

When the program is run:

```
> java ReadingExceptions
Please enter an integer: abc
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:819)
    at java.util.Scanner.next(Scanner.java:1431)
    at java.util.Scanner.nextInt(Scanner.java:2040)
    at java.util.Scanner.nextInt(Scanner.java:2000)
    at ReadingExceptions.main(ReadingExceptions.java:7)
```

The stacktrace, i.e. the sequence of method calls in which the error occurs, is displayed in reverse order.
To understand what happened, start at the bottom of the messages: the exception was thrown in class ReadingExceptions in the main method. More specifically the error occurred while executing line 7 of file ReadingExceptions.java. That line called line 2000 of file Scanner.java (method java.util.Scanner.nextInt), which in turn called line 2040 of Scanner.java.

Then line 1431 of Scanner.java was called which threw the exception java.util.InputMismatchException. This exception was thrown because a String was entered as input, while the Scanner.nextInt method expects an integer, and it cannot convert the string to an integer.