

JAVA

INTRODUCTION

What is Java?

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is used for:

- Mobile applications (specially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection
- And much, much more!

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming language in the world
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has a huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to [C++](#) and [C#](#), it makes it easy for programmers to switch to Java or vice versa

JAVA GETTING STARTED

Java Install

- Some PCs might have Java already installed.
- To check if you have Java installed on a Windows PC, search in the start bar for Java or type the following in Command Prompt (cmd.exe):

```
C:\Users\Your Name>java -version
```

If Java is installed, you will see something like this (depending on version):

```
java version "11.0.1" 2018-10-16 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.1+13-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.1+13-LTS, mixed mode)
```

If you do not have Java installed on your computer, you can download it for free at [oracle.com](https://www.oracle.com).

Note: In this tutorial, we will write Java code in a text editor. However, it is possible to write Java in an Integrated Development Environment, such as IntelliJ IDEA, Netbeans or Eclipse, which are particularly useful when managing larger collections of Java files.

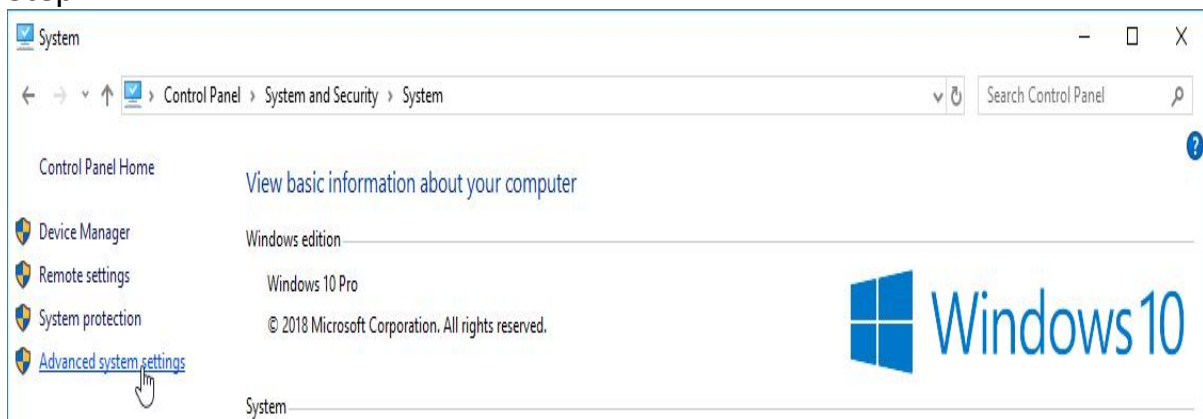
Setup for Windows

To install Java on Windows:

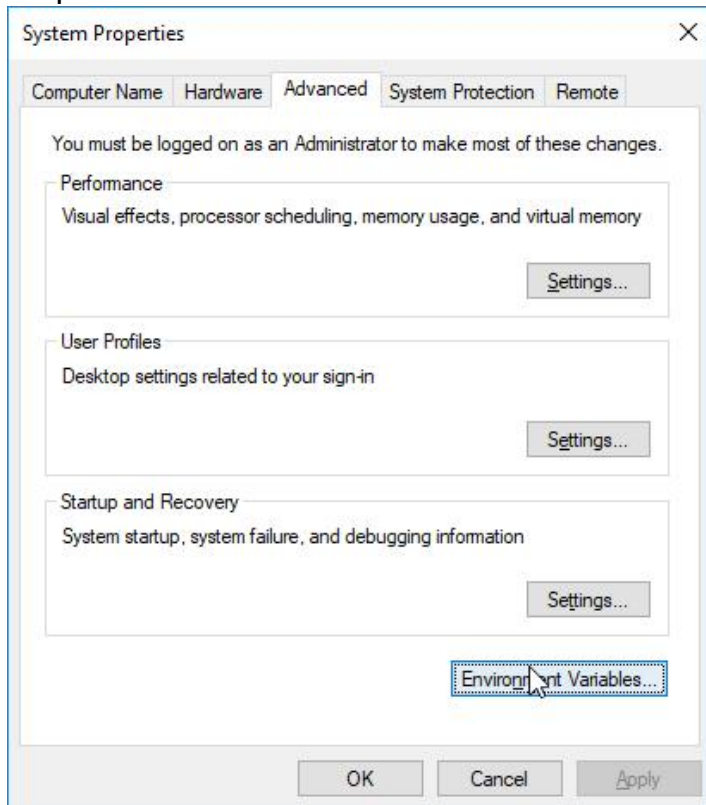
1. Go to "System Properties" (Can be found on Control Panel > System and Security > System > Advanced System Settings)
2. Click on the "Environment variables" button under the "Advanced" tab
3. Then, select the "Path" variable in System variables and click on the "Edit" button
4. Click on the "New" button and add the path where Java is installed, followed by `\bin`. By default, Java is installed in `C:\Program Files\Java\jdk-11.0.1` (If nothing else was specified when you installed it). In that case, You will have to add a new path with: **`C:\Program Files\Java\jdk-11.0.1\bin`**
Then, click "OK", and save the settings
5. At last, open Command Prompt (cmd.exe) and type **`java -version`** to see if Java is running on your machine

Show how to install Java step-by-step with images »

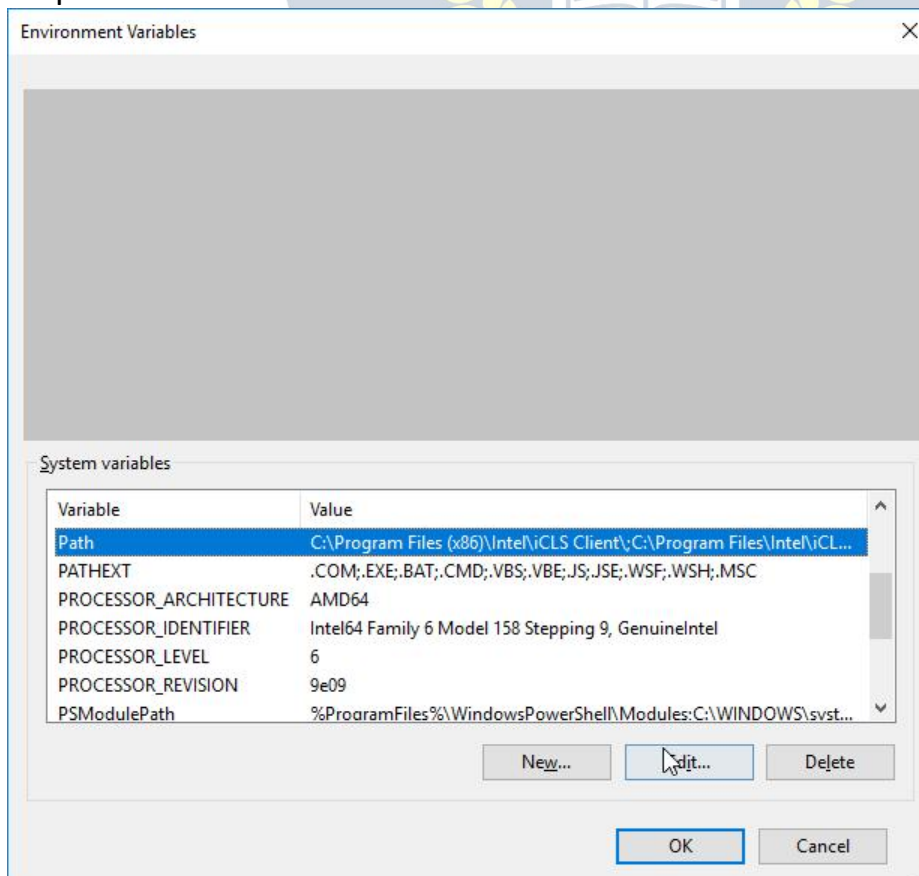
Step 1



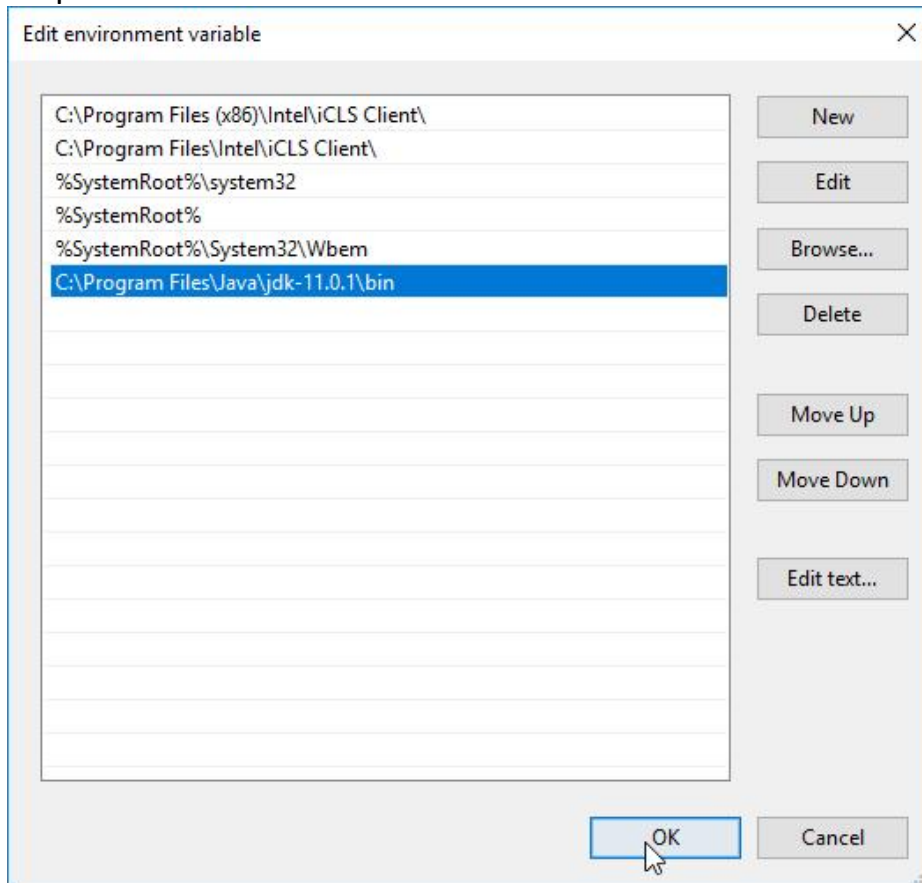
Step 2



Step 3



Step 4



Step 5

Write the following in the command line (cmd.exe):

```
C:\Users\Your Name>java -version
```

If Java was successfully installed, you will see something like this (depending on version):

```
java version "11.0.1" 2018-10-16 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.1+13-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.1+13-LTS, mixed mode)
```

Java Quickstart

- In Java, every application begins with a class name, and that class must match the filename.
- Let's create our first Java file, called Main.java, which can be done in any text editor (like Notepad).
- The file should contain a "Hello World" message, which is written with the following code:

Main.java

```
public class Main {  
    public static void main(String[] args)  
        {System.out.println("Hello World");  
    }  
}
```

- Don't worry if you don't understand the code above - we will discuss it in detail in later chapters. For now, focus on **how** to run the code above.
- Save the code in Notepad as "Main.java". Open Command Prompt (cmd.exe), navigate to the directory where you saved your file, and type "javac Main.java":

```
C:\Users\Your Name>javac Main.java
```

This will compile your code. If there are no errors in the code, the command prompt will take you to the next line. Now, type "java Main" to run the file:

```
C:\Users\Your Name>java Main
```

The output should read:

```
Hello World
```

Try it Yourself »

```
public class Main {  
    public static void main(String[] args)  
        {System.out.println("Hello World");  
    }  
}
```

Congratulations! You have written and executed your first Java program.

JAVA SYNTAX

Java Syntax

In the previous chapter, we created a Java file called **Main.java**, and we used the following code to print "Hello World" to the screen:

Main.java

```
public class Main {  
    public static void main(String[] args)  
        {System.out.println("Hello World");  
    }  
}
```

Example explained

- Every line of code that runs in Java must be inside a **class**. In our example, we named the class **Main**. A class should always start with an uppercase first letter.
- **Note:** Java is case-sensitive: "MyClass" and "myclass" has different meaning.
- The name of the java file **must match** the class name. When saving the file, save it using the class name and add ".java" to the end of the filename. To run the example above on your computer, make sure that Java is properly installed: Go to the Get Started Chapter for how to install Java. The output should be:

```
Hello World
```

The main Method

The **main()** method is required and you will see it in every Java program:

```
public static void main(String[] args)
```

- Any code inside the **main()** method will be executed. You don't have to understand the keywords before and after main. You will get to know them bit by bit while reading this tutorial.
- For now, just remember that every Java program has a **class** name which must match the filename, and that every program must contain the **main()** method.

System.out.println()

Inside the **main()** method, we can use the **println()** method to print a line of text to the screen:

```
public static void main(String[] args) {  
    System.out.println("Hello World");  
}
```

Note: The curly braces **{}** marks the beginning and the end of a block of code.

Note: Each code statement must end with a semicolon.

JAVA COMMENTS

Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line Comments

- Single-line comments start with two forward slashes (`//`).
- Any text between `//` and the end of the line is ignored by Java (will not be executed).
- This example uses a single-line comment before a line of code:

Example:

```
// This is a comment  
System.out.println("Hello World");
```

This example uses a single-line comment at the end of a line of code:

Example:

```
System.out.println("Hello World"); // This is a comment
```

Java Multi-line Comments

- Multi-line comments start with `/*` and ends with `*/`.
- Any text between `/*` and `*/` will be ignored by Java.
- This example uses a multi-line comment (a comment block) to explain the code:

Example:

```
/* The code below will print the words Hello World  
to the screen, and it is amazing */  
System.out.println("Hello World");
```

Single or multi-line comments?

It is up to you which you want to use. Normally, we use `//` for short comments, and `/* */` for longer.

JAVA VARIABLES

Java Variables

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

- **String** - stores text, such as "Hello". String values are surrounded by double quotes
- **int** - stores integers (whole numbers), without decimals, such as 123 or -123
- **float** - stores floating point numbers, with decimals, such as 19.99 or -19.99

- **char** - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- **boolean** - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

Syntax

type variable = value;

- Where *type* is one of Java's types (such as **int** or **String**), and *variable* is the name of the variable (such as **x** or **name**). The **equal sign** is used to assign values to the variable.
- To create a variable that should store text, look at the following example:

Example:

Create a variable called **name** of type **String** and assign it the value "John":

```
public class Main {
    public static void main(String[] args)
    {String name = "John";
    System.out.println(name);
    }
}
```

To create a variable that should store a number, look at the following example:

Example:

Create a variable called **myNum** of type **int** and assign it the value **15**:

```
public class Main {
    public static void main(String[] args)
    {int myNum = 15;
    System.out.println(myNum);
    }
}
```

You can also declare a variable without assigning the value, and assign the value later:

Example:

```
public class Main {
    public static void main(String[] args) {
```



```
int myNum;  
myNum = 15;  
System.out.println(myNum);  
}  
}
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of `myNum` from `15` to `20`:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int myNum = 15;  
        myNum = 20; // myNum is now 20  
        System.out.println(myNum);  
    }  
}
```

Final Variables

However, you can add the `final` keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        final int myNum = 15;  
        myNum = 20; // will generate an error  
        System.out.println(myNum);  
    }  
}
```

Other Types

A demonstration of how to declare variables of other types:

Example:

```
int myNum = 5;  
float myFloatNum = 5.99f;
```

```
char myLetter = 'D';
boolean myBool = true;
String myText = "Hello";
```

You will learn more about data types in the next chapter.

Display Variables

- The `println()` method is often used to display variables.
- To combine both text and a variable, use the `+` character:

Example:

```
public class Main {
    public static void main(String[] args)
    {String name = "John";
     System.out.println("Hello " + name);
    }
}
```

You can also use the `+` character to add a variable to another variable:

Example:

```
public class Main {
    public static void main(String[] args)
    {String firstName = "John ";
     String lastName = "Doe";
     String fullName = firstName + lastName;
     System.out.println(fullName);
    }
}
```

For numeric values, the `+` character works as a mathematical operator (notice that we use `int` (integer) variables here):

Example:

```
public class Main {
    public static void main(String[] args)
    {int x = 5;
     int y = 6;
     System.out.println(x + y); // Print the value of x + y
    }
}
```

From the example above, you can expect:

- x stores the value 5
- y stores the value 6
- Then we use the `println()` method to display the value of $x + y$, which is **11**

Declare Many Variables

To declare more than one variable of the **same type**, use a comma-separated list:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {int x = 5, y = 6, z = 50;  
    System.out.println(x + y + z);  
    }  
}
```

Java Identifiers

- All Java **variables** must be **identified** with **unique names**.
- These unique names are called **identifiers**.
- Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).
- **Note:** It is recommended to use descriptive names in order to create understandable and maintainable code:

Example:

```
public class Main {  
    public static void main(String[] args) {  
        // Good  
        int minutesPerHour = 60;  
  
        // OK, but not so easy to understand what m actually is  
        int m = 60;  
  
        System.out.println(minutesPerHour);  
        System.out.println(m);  
    }  
}
```

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names can also begin with \$ and _ (but we will not use it in this tutorial)
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as **int** or **boolean**) cannot be used as names

JAVA DATA TYPES

Java Data Types

As explained in the previous chapter, a variable in Java must be a specified data type:

Example:

```
public class Main {
    public static void main(String[] args) {
        int myNum = 5;           // Integer (whole number)
        float myFloatNum = 5.99f; // Floating point number
        char myLetter = 'D';     // Character
        boolean myBool = true;   // Boolean
        String myText = "Hello"; // String
        System.out.println(myNum);
        System.out.println(myFloatNum);
        System.out.println(myLetter);
        System.out.println(myBool);
        System.out.println(myText);
    }
}
```

Data types are divided into two groups:

- Primitive data types - includes **byte**, **short**, **int**, **long**, **float**, **double**, **boolean** and **char**
- Non-primitive data types - such as [String](#), [Arrays](#) and [Classes](#) (you will learn more about these in a later chapter)

Primitive Data Types

- A primitive data type specifies the size and type of variable values, and it has no additional methods.
- There are eight primitive data types in Java:

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Numbers

- Primitive number types are divided into two groups:
- **Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are **byte**, **short**, **int** and **long**. Which type you should use, depends on the numeric value.
- **Floating point types** represents numbers with a fractional part, containing one or more decimals. There are two types: **float** and **double**.

Even though there are many numeric types in Java, the most used for numbers are **int** (for whole numbers) and **double** (for floating point numbers). However, we will describe them all as you continue to read.

Integer Types

Byte

The **byte** data type can store whole numbers from -128 to 127. This can be used instead of **int** or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        byte myNum = 100;  
    }  
}
```

```
System.out.println(myNum);  
}  
}
```

Short

The **short** data type can store whole numbers from -32768 to 32767:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        short myNum = 5000;  
        System.out.println(myNum);  
    }  
}
```

Int

The **int** data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the **int** data type is the preferred data type when we create variables with a numeric value.

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int myNum = 100000;  
        System.out.println(myNum);  
    }  
}
```

Long

The **long** data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        long myNum = 15000000000L;  
        System.out.println(myNum);  
    }  
}
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The **float** data type can store fractional numbers from $3.4e-038$ to $3.4e+038$. Note that you should end the value with an "f":

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        float myNum = 5.75f;  
        System.out.println(myNum);  
    }  
}
```

Double

The **double** data type can store fractional numbers from $1.7e-308$ to $1.7e+308$. Note that you should end the value with a "d":

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        double myNum = 19.99d;  
        System.out.println(myNum);  
    }  
}
```

Use **float** or **double**?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of **float** is only six or seven decimal digits, while **double** variables have a precision of about 15 digits. Therefore it is safer to use **double** for most calculations.

Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        float f1 = 35e3f;  
        double d1 = 12E4d;  
    }  
}
```

```
System.out.println(f1);
System.out.println(d1);
}
}
```

Booleans

A boolean data type is declared with the **boolean** keyword and can only take the values **true** or **false**:

Example:

```
public class Main {
    public static void main(String[] args)
    {boolean isJavaFun = true;
    boolean isFishTasty = false;
    System.out.println(isJavaFun); // Outputs true
    System.out.println(isFishTasty); // Outputs false
    }
}
```

Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

Characters

The **char** data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

Example:

```
public class Main {
    public static void main(String[] args)
    {char myGrade = 'B';
    System.out.println(myGrade);
    }
}
```

Alternatively, you can use ASCII values to display certain characters:

Example:

```
public class Main {
    public static void main(String[] args)
    {char a = 65, b = 66, c = 67;
    System.out.println(a);
    System.out.println(b);
}
```



```
System.out.println(c);  
}  
}
```

Strings

The **String** data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {String greeting = "Hello World";  
    System.out.println(greeting);  
    }  
}
```

- The String type is so much used and integrated in Java, that some call it "the special **ninth** type".
- A String in Java is actually a **non-primitive** data type, because it refers to an object. The String object has methods that are used to perform certain operations on strings. **Don't worry if you don't understand the term "object" just yet.** We will learn more about strings and objects in a later chapter.

Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for **String**).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be **null**.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc. You will learn more about these in a later chapter.

JAVA TYPE CASTING

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
`byte -> short -> char -> int -> long -> float -> double`
- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
`double -> float -> long -> int -> char -> short -> byte`

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example:

```
public class Main {
    public static void main(String[] args)
    {
        int myInt = 9;
        double myDouble = myInt; // Automatic casting: int to double

        System.out.println(myInt); // Outputs 9
        System.out.println(myDouble); // Outputs 9.0
    }
}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example:

```
public class Main {
    public static void main(String[] args)
    {
        double myDouble = 9.78d;
        int myInt = (int) myDouble; // Manual casting: double to int

        System.out.println(myDouble); // Outputs 9.78
        System.out.println(myInt); // Outputs 9
    }
}
```

JAVA OPERATORS

Java Operators

- Operators are used to perform operations on variables and values.
- In the example below, we use the **+** operator to add together two values:

Example:

```
public class Main {  
    public static void main(String[] args)  
{int x = 100 + 50;  
System.out.println(x);  
}  
}
```

Although the **+** operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example:

```
public class Main {  
    public static void main(String[] args)  
{ int sum1 = 100 + 50;    // 150 (100 + 50)  
int sum2 = sum1 + 250;    // 400 (150 + 250)  
int sum3 = sum2 + sum2;    // 800 (400 + 400)  
    System.out.println(sum1);  
    System.out.println(sum2);  
    System.out.println(sum3);  
}  
}
```

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
----------	------	-------------	---------

+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Java Assignment Operators

- Assignment operators are used to assign values to variables.
- In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example:

```
public class Main {
    public static void main(String[] args)
    {int x = 10;
    System.out.println(x);
    }
}
```

The **addition assignment** operator (+=) adds a value to a variable:

Example:

```
public class Main {
    public static void main(String[] args)
    {int x = 10;
    x += 5;
    System.out.println(x);
    }
}
```

A list of all assignment operators:

Operator	Example	Same As
=	$x = 5$	$x = 5$
+=	$x += 3$	$x = x + 3$
-=	$x -= 3$	$x = x - 3$
*=	$x *= 3$	$x = x * 3$
/=	$x /= 3$	$x = x / 3$
%=	$x \% = 3$	$x = x \% 3$

&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Java Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Java Logical Operators

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
	Logical or	Returns true if one of the statements is true	x < 5 x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

WWW.VIDYAPITH.IN

JAVA STRINGS

Java Strings

- Strings are used for storing text.
- A **String** variable contains a collection of characters surrounded by double quotes:

Example:

Create a variable of type **String** and assign it a value:

```
public class Main {
    public static void main(String[] args)
    {String greeting = "Hello";
```

```
System.out.println(greeting);
}
}
```

String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the `length()` method:

Example:

```
public class Main {
    public static void main(String[] args) {
        String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        System.out.println("The length of the txt string is: " + txt.length());
    }
}
```

More String Methods

There are many string methods available, for example `toUpperCase()` and `toLowerCase()`:

Example:

```
public class Main {
    public static void main(String[] args)
    {String txt = "Hello World";
    System.out.println(txt.toUpperCase()); // Outputs "HELLO WORLD"
    System.out.println(txt.toLowerCase()); // Outputs "hello world"
    }
}
```



Finding a Character in a String

The `indexOf()` method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

Example:

```
public class Main {
    public static void main(String[] args) {
        String txt = "Please locate where 'locate' occurs!";
        System.out.println(txt.indexOf("locate")); // Outputs 7
    }
}
```

Java counts positions from zero.

0 is the first position in a string, 1 is the second, 2 is the third ...

String Concatenation

The **+** operator can be used between strings to combine them. This is called **concatenation**:

Example:

```
public class Main {
    public static void main(String args[])
    {String firstName = "John";
    String lastName = "Doe";
    System.out.println(firstName + " " + lastName);
    }
}
```

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

You can also use the **concat()** method to concatenate two strings:

Example:

```
public class Main {
    public static void main(String[] args)
    {String firstName = "John ";
    String lastName = "Doe";
    System.out.println(firstName.concat(lastName));
    }
}
```

Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

```
String txt = "We are the so-called "Vikings" from the north.";
```

- The solution to avoid this problem, is to use the **backslash escape character**.
- The backslash (****) escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote

\"	"	Double quote
\\	\	Backslash

The sequence `\"` inserts a double quote in a string:

Example:

```
public class Main {
    public static void main(String[] args) {
        String txt = "We are the so-called \"Vikings\" from the north.";
        System.out.println(txt);
    }
}
```

The sequence `'` inserts a single quote in a string:

Example:

```
public class Main {
    public static void main(String[] args)
    {String txt = "It's alright.";
    System.out.println(txt);
    }
}
```

The sequence `\\` inserts a single backslash in a string:

Example:

```
public class Main {
    public static void main(String[] args) {
        String txt = "The character \\ is called backslash.";
        System.out.println(txt);
    }
}
```

Six other escape sequences are valid in Java:

Code	Result
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed

Adding Numbers and Strings

WARNING!

Java uses the + operator for both addition and concatenation. Numbers are added. Strings are concatenated. If you add two numbers, the result will be a number:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int x = 10;  
        int y = 20;  
        int z = x + y  
        System.out.println(z);  
    }  
}
```

If you add two strings, the result will be a string concatenation:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        String x = "10";  
        String y = "20";  
        String z = x + y;  
        System.out.println(z);  
    }  
}
```

If you add a number and a string, the result will be a string concatenation:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        String x = "10";  
        int y = 20;  
        String z = x + y;  
        System.out.println(z);  
    }  
}
```

Complete String Reference

For a complete reference of String methods, go to our Java String Methods Reference.

The reference contains descriptions and examples of all string methods.

JAVA MATH

The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.max(x,y)

The **Math.max(x,y)** method can be used to find the highest value of x and y:

Example:

```
public class Main {  
    public static void main(String[] args)  
    { System.out.println Math.max(5,  
        10));  
    }  
}
```

Math.min(x,y)

The **Math.min(x,y)** method can be used to find the lowest value of x and y:

Example:

```
public class Main {  
    public static void main(String[] args)  
    { System.out.println(Math.min(5,  
        10));  
    }  
}
```

Math.sqrt(x)

The **Math.sqrt(x)** method returns the square root of x:

Example:

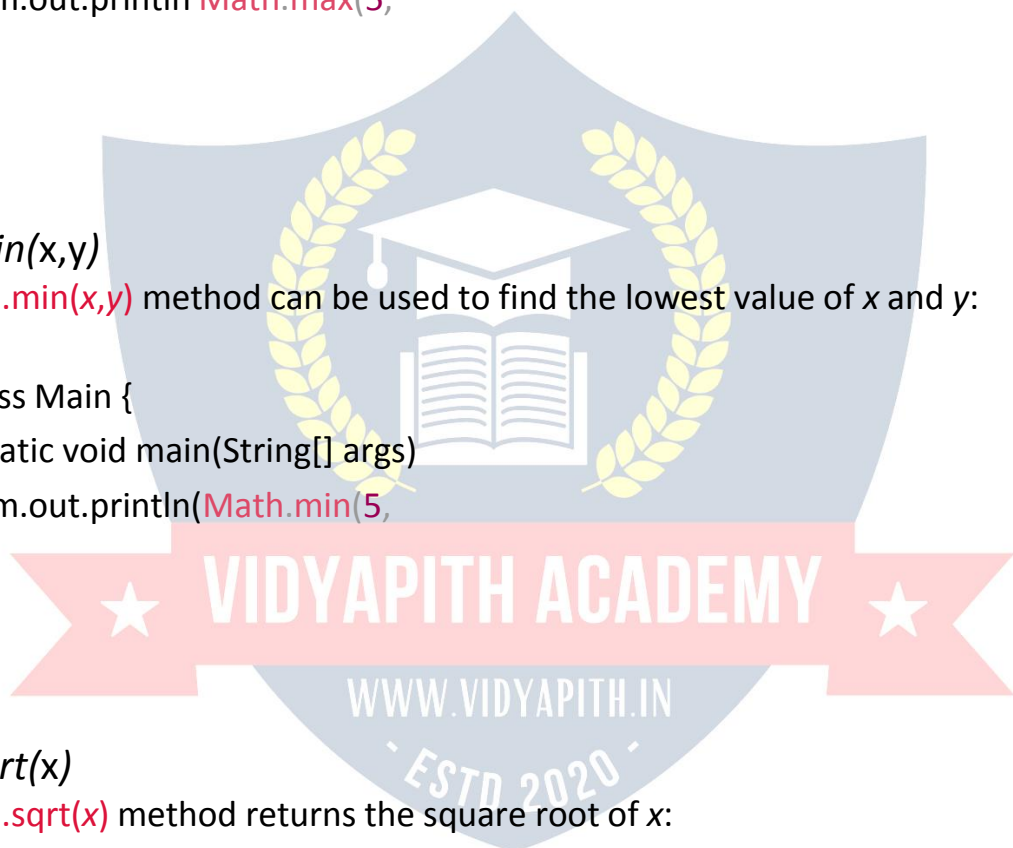
```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(Math.sqrt(64));  
    }  
}
```

Math.abs(x)

The **Math.abs(x)** method returns the absolute (positive) value of x:

Example:

```
public class Main {
```



```
public static void main(String[] args) {
```



```
System.out.println(Math.abs(-4.7));  
}  
}
```

Random Numbers

Math.random() returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

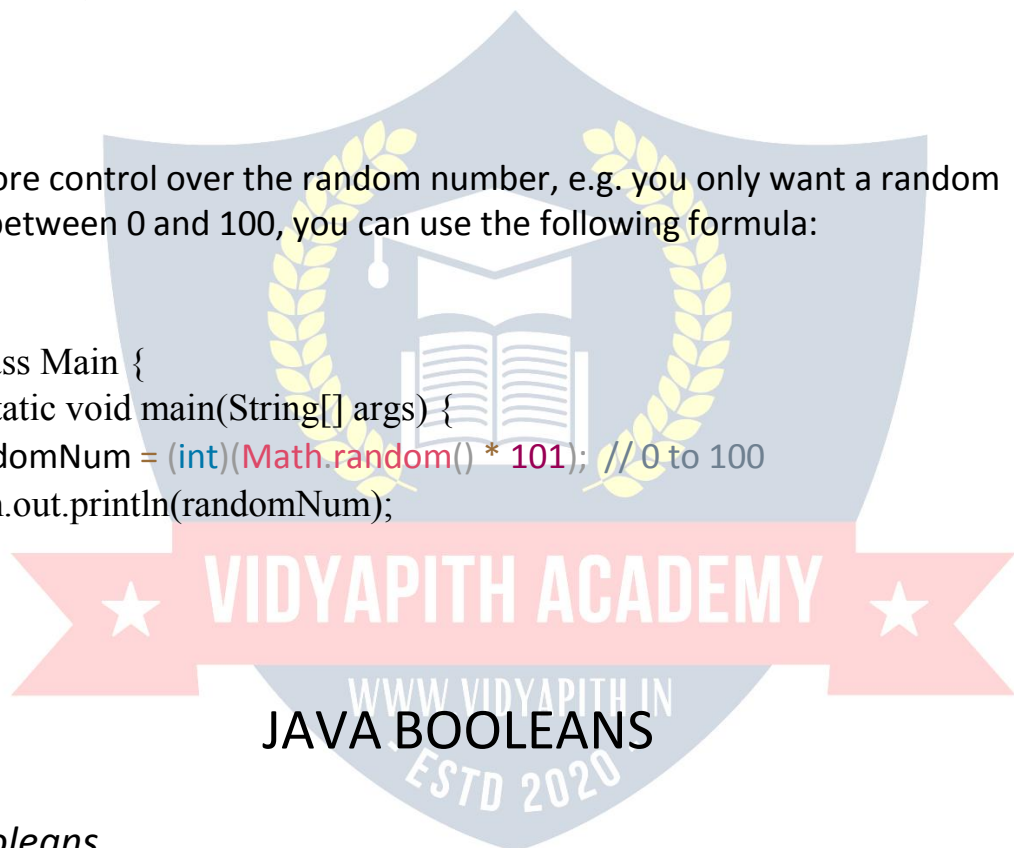
Example:

```
public class Main {  
    public static void main(String[] args)  
    {System.out.println(Math.random());  
    }  
}
```

To get more control over the random number, e.g. you only want a random number between 0 and 100, you can use the following formula:

Example:

```
public class Main {  
    public static void main(String[] args) {  
        int randomNum = (int)(Math.random() * 101); // 0 to 100  
        System.out.println(randomNum);  
    }  
}
```



JAVA BOOLEANS

Java Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, Java has a **boolean** data type, which can take the values **true** or **false**.

Boolean Values

A boolean type is declared with the **boolean** keyword and can only take the values **true** or **false**:

Example:

```
public class Main {
    public static void main(String[] args)
    {boolean isJavaFun = true;
    boolean isFishTasty = false;
    System.out.println(isJavaFun); // Outputs true
    System.out.println(isFishTasty); // Outputs false
    System.out.println(isJavaFun);
        System.out.println(isFishTasty);
    }
}
```

However, it is more common to return boolean values from boolean expressions, for conditional testing (see below).

Boolean Expression

- A **Boolean expression** is a Java expression that returns a Boolean value: **true** or **false**.
- You can use a comparison operator, such as the **greater than (>)** operator to find out if an expression (or a variable) is true:

Example:

```
public class Main {
    public static void main(String[] args)
    {int x = 10;
    int y = 9;
    System.out.println(x > y); // returns true, because 10 is higher than 9
    }
}
```

Or even easier:

Example:

```
public class Main {
    public static void main(String[] args) {
    System.out.println(10 > 9); // returns true, because 10 is higher than 9
    }
}
```

```
}
```

In the examples below, we use the **equal to** (`==`) operator to evaluate an expression:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int x = 10;  
        System.out.println(x == 10); // returns true, because the value of x is equal  
        to 10  
    }  
}
```

Example:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(10 == 15); // returns false, because 10 is not equal to 15  
    }  
}
```

JAVA IF ... ELSE

Java Conditions and If Statements

Java supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
- Use **else** to specify a block of code to be executed, if the same condition is false
- Use **else if** to specify a new condition to test, if the first condition is false

- Use **switch** to specify many alternative blocks of code to be executed

The if Statement

Use the **if** statement to specify a block of Java code to be executed if a condition is **true**.

Syntax:

```
if (condition) {  
    // block of code to be executed if the condition is true  
}
```

Note that **if** is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is **true**, print some text:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        if (20 > 18) {  
            System.out.println("20 is greater than 18");  
        }  
    }  
}
```

We can also test variables:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int x = 20;  
        int y = 18;  
        if (x > y) {  
            System.out.println("x is greater than y");  
        }  
    }  
}
```

Example explained

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the **>** operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

Use the **else** statement to specify a block of code to be executed if the condition is **false**.

Syntax:

```
if (condition) {  
    // block of code to be executed if the condition is true  
} else {  
    // block of code to be executed if the condition is false  
}
```

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int time = 20;  
        if (time < 18)  
            System.out.println("Good  
            day.");  
        else {  
            System.out.println("Good evening.");  
        }  
    }  
}
```

Example explained

In the example above, time (20) is greater than 18, so the condition is **false**. Because of this, we move on to the **else** condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

The else if Statement

Use the **else if** statement to specify a new condition if the first condition is **false**.



Syntax:

```
if (condition1) {  
    // block of code to be executed if condition1 is true  
} else if (condition2) {  
    // block of code to be executed if the condition1 is false and condition2 is true  
} else {  
    // block of code to be executed if the condition1 is false and condition2 is false  
}
```

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int time = 22;  
        if (time < 10)  
        {  
            System.out.println("Good  
morning.");  
        } else if (time < 20)  
        {  
            System.out.println("Good  
day.");  
        } else {  
            System.out.println("Good evening.");  
        }  
    }  
}
```

Example explained

In the example above, time (22) is greater than 10, so the **first condition** is **false**. The next condition, in the **else if** statement, is also **false**, so we move on to the **else** condition since **condition1** and **condition2** is both **false** - and print to the screen "Good evening". However, if the time was 14, our program would print "Good day."

Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax:

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Instead of writing:



Example:

```
public class Main {
    public static void main(String[] args)
    {int time = 20;
    if (time < 18)
    { System.out.println("Good
    day.");
    } else {
    System.out.println("Good evening.");
    }
    }
}
```

You can simply write:

Example:

```
public class Main {
    public static void main(String[] args)
    {int time = 20;
    String result = (time < 18) ? "Good day." : "Good evening.";
    System.out.println(result);
    }
}
```

★ VIDYAPITH ACADEMY ★

Java Switch Statements

Use the **switch** statement to select one of many code blocks to be executed.

Syntax:

```
switch(expression)
{case x:
    // code block
    break;
case y:
    // code block
break; default:
    // code block
}
```

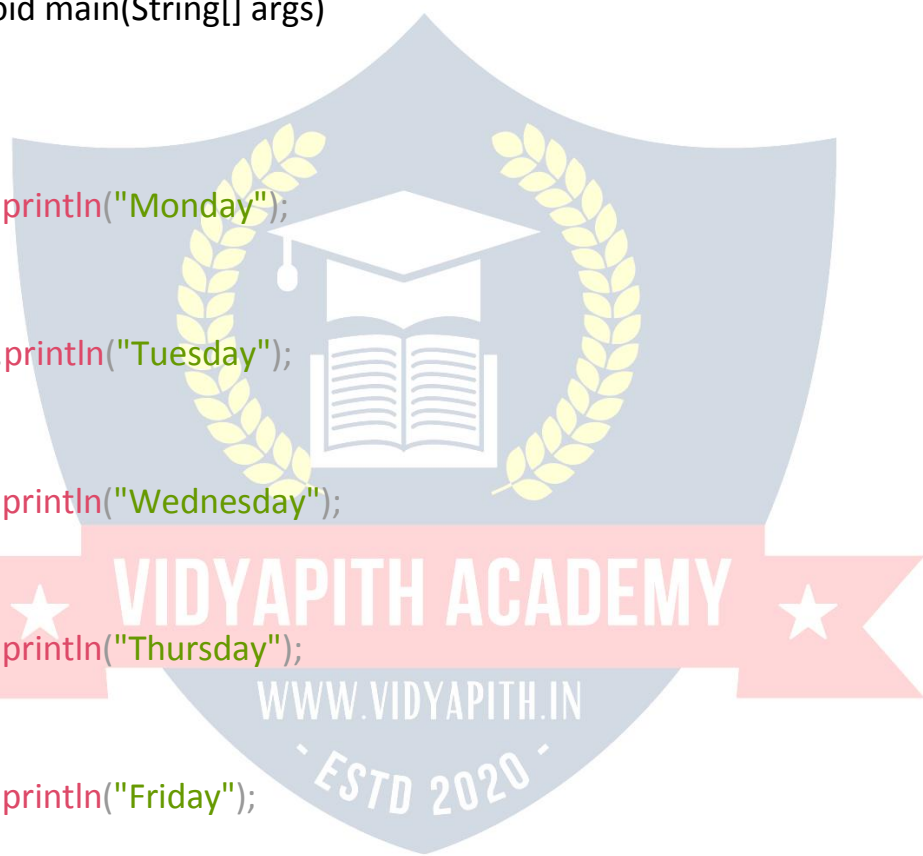
This is how it works:

- The **switch** expression is evaluated once.
- The value of the expression is compared with the values of each **case**.
- If there is a match, the associated block of code is executed.
- The **break** and **default** keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int day = 4;  
        switch (day)  
        {  
            case 1:  
                System.out.println("Monday");  
                break;  
            case 2:  
                System.out.println("Tuesday");  
                break;  
            case 3:  
                System.out.println("Wednesday");  
                break;  
            case 4:  
                System.out.println("Thursday");  
                break;  
            case 5:  
                System.out.println("Friday");  
                break;  
            case 6:  
                System.out.println("Saturday");  
                break;  
            case 7:  
                System.out.println("Sunday");  
                break;  
        }  
    }  
}
```



The break Keyword

- When Java reaches a **break** keyword, it breaks out of the switch block.
- This will stop the execution of more code and case testing inside the block.
- When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The **default** keyword specifies some code to run if there is no case match:

Example:

```
public class Main {  
    public static void main(String[] args) {  
        int day = 4;  
        switch (day)  
        {  
            case 6:  
                System.out.println("Today is Saturday");  
                break;  
            case 7:  
                System.out.println("Today is Sunday");  
                break;  
            default:  
                System.out.println("Looking forward to the Weekend");  
        }  
    }  
}
```

Note that if the **default** statement is used as the last statement in a switch block, it does not need a break.

JAVA WHILE LOOP

Loops

- Loops can execute a block of code as long as a specified condition is reached.

- Loops are handy because they save time, reduce errors, and they make code more readable.

Java While Loop

The **while** loop loops through a block of code as long as a specified condition is **true**:

Syntax:

```
while (condition) {  
    // code block to be executed  
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int i = 0;  
        while (i < 5)  
        {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The **do/while** loop is a variant of the **while** loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax:

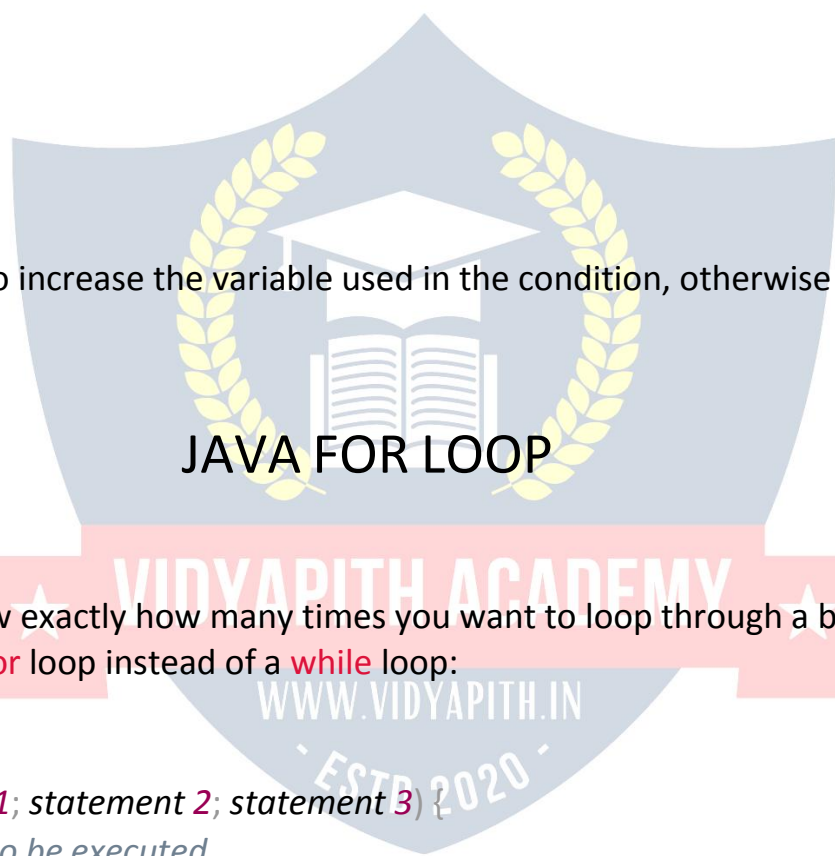
```
do {  
    // code block to be executed  
}  
while (condition);
```

The example below uses a **do/while** loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example:

```
public class Main {
    public static void main(String[] args)
    {int i = 0;
    do
    { System.out.println(i);
    i++;
    }
    while (i < 5);
    }
}
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!



JAVA FOR LOOP

Java For Loop

When you know exactly how many times you want to loop through a block of code, use the **for** loop instead of a **while** loop:

Syntax:

```
for (statement 1; statement 2; statement 3) {
    // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example:

```
public class Main {
    public static void main(String[] args) {
```

```

for (int i = 0; i < 5; i++)
    {System.out.println(i);
    }
}
}

```

Example explained

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Another Example

This example will only print even values between 0 and 10:

Example:

```

public class Main {
    public static void main(String[] args)
    {for (int i = 0; i <= 10; i = i + 2)
    { System.out.println(i);
    }
    }
}

```

For-Each Loop

There is also a "**for-each**" loop, which is used exclusively to loop through elements in an **array**:

Syntax:

```

for (type variableName : arrayName) {
    // code block to be executed
}

```

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

Example:

```

public class Main {
    public static void main(String[] args) {

```



```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
    System.out.println(i);
}
}
```

JAVA BREAK AND CONTINUE

Java Break

- You have already seen the **break** statement used in an earlier chapter of this tutorial. It was used to "jump out" of a **switch** statement.
- The **break** statement can also be used to jump out of a **loop**.
- This example stops the loop when i is equal to 4:

Example:

```
public class Main {
    public static void main(String[] args)
    {for (int i = 0; i < 10; i++) {
        if (i == 4)
            {break;
        }
        System.out.println(i);
    }
}
```

Java Continue

- The **continue** statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.
- This example skips the value of 4:

Example:

```
public class Main {
    public static void main(String[] args)
    {for (int i = 0; i < 10; i++) {
        if (i == 4)
            { continu
            e;
        }
        System.out.println(i);
    }
}
```

```
}  
}  
}
```

Break and Continue in While Loop

You can also use **break** and **continue** in while loops:

Break Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int i = 0;  
        while (i < 10)  
        {  
            System.out.println(i);  
            i++;  
            if (i == 4)  
            {  
                break;  
            }  
        }  
    }  
}
```

Continue Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        int i = 0;  
        while (i < 10)  
        {  
            if (i == 4)  
            {  
                i++;  
                continue;  
            }  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```



JAVA ARRAYS

Java Arrays

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- To declare an array, define the variable type with **square brackets**:

```
String[] cars;
```

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

- You access an array element by referring to the index number.
- This statement accesses the value of the first element in cars:

Example:

```
public class Main {  
    public static void main(String[] args) {  
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
        System.out.println(cars[0]);  
    }  
}
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example:

```
cars[0] = "Opel";
```

Example:

```
public class Main {
    public static void main(String[] args) {
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
        cars[0] = "Opel";
        System.out.println(cars[0]);
    }
}
```

Array Length

To find out how many elements an array has, use the **length** property:

Example:

```
public class Main {
    public static void main(String[] args) {
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
        System.out.println(cars.length);
    }
}
```

Loop Through an Array

- You can loop through the array elements with the **for** loop, and use the **length** property to specify how many times the loop should run.
- The following example outputs all elements in the **cars** array:

Example:

```
public class Main {
    public static void main(String[] args) {
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
        for (int i = 0; i < cars.length; i++)
        { System.out.println(cars[i]);
        }
    }
}
```

Loop Through an Array with For-Each

There is also a **"for-each"** loop, which is used exclusively to loop through elements in arrays:

Syntax

```
for (type variable : arrayname) {  
    ...  
}
```

The following example outputs all elements in the **cars** array, using a "for-each" loop:

Example:

```
public class Main {  
    public static void main(String[] args) {  
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
        for (String i : cars) {  
            System.out.println(i);  
        }  
    }  
}
```

- The example above can be read like this: **for each** **String** element (called **i** - as in index) in **cars**, print out the value of **i**.
- If you compare the **for** loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

Multidimensional Arrays

- A multidimensional array is an array containing one or more arrays.
- To create a two-dimensional array, add each array within its own set of **curly braces**:

Example:

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

- **myNumbers** is now an array with two arrays as its elements.
- To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of **myNumbers**:

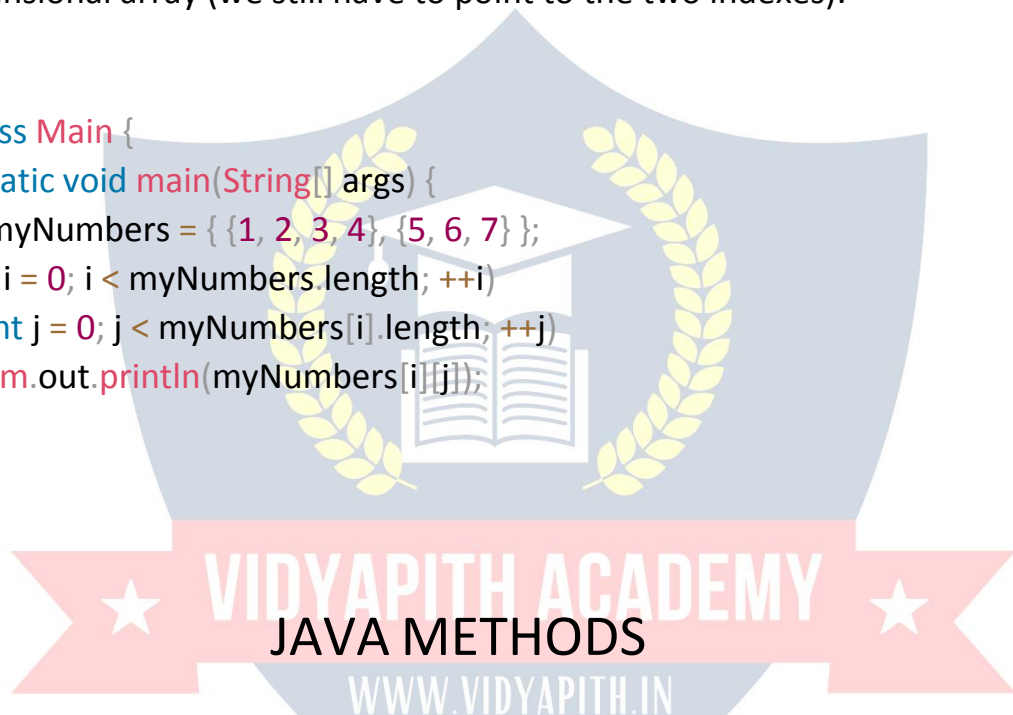
Example:

```
public class Main {  
    public static void main(String[] args) {  
        int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };  
        int x = myNumbers[1][2];  
        System.out.println(x);  
    }  
}
```

We can also use a **for loop** inside another **for loop** to get the elements of a two-dimensional array (we still have to point to the two indexes):

Example:

```
public class Main {  
    public static void main(String[] args) {  
        int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };  
        for (int i = 0; i < myNumbers.length; ++i)  
            { for(int j = 0; j < myNumbers[i].length; ++j)  
                {System.out.println(myNumbers[i][j]);  
                }  
            }  
    }  
}
```



- A **method** is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a method.
- Methods are used to perform certain actions, and they are also known as **functions**.
- Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as **System.out.println()**, but you can also create your own methods to perform certain actions:

Example:

Create a method inside Main:

```
public class Main {
    static void myMethod() {
        // code to be executed
    }
}
```

Example Explained

- **myMethod()** is the name of the method
- **static** means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- **void** means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

- To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon;
- In the following example, **myMethod()** is used to print a text (the action), when it is called:

Example:

Inside **main**, call the **myMethod()** method:

```
public class Main {
    static void myMethod()
    { System.out.println("I just got executed!");
    }

    public static void main(String[] args)
    {myMethod();
    }
}
```

```
// Outputs "I just got executed!"
```

A method can also be called multiple times:

Example:

```
public class Main {
    static void myMethod()
        { System.out.println("I just got executed!");
    }
}
```

```
public static void main(String[] args)
    {myMethod();
    myMethod();
    myMethod();
    }
}
```

```
// I just got executed!
// I just got executed!
// I just got executed!
```

JAVA METHOD PARAMETERS

Parameters and Arguments

- Information can be passed to methods as parameter. Parameters act as variables inside the method.
- Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.
- The following example has a method that takes a **String** called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example:

```
public class Main {
    static void myMethod(String fname)
        { System.out.println(fname + "
    Refsnes");
    }
    public static void main(String[] args) {
```



```
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
}
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: **fname** is a **parameter**, while **Liam**, **Jenny** and **Anja** are **arguments**.

Multiple Parameters

You can have as many parameters as you like:

Example:

```
public class Main {
    static void myMethod(String fname, int age)
    {System.out.println(fname + " is " + age);
    }
}
```

```
public static void main(String[] args)
{myMethod("Liam", 5);
myMethod("Jenny", 8);
myMethod("Anja", 31);
}
}
```

```
// Liam is 5
// Jenny is 8
// Anja is 31
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The **void** keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as **int**, **char**, etc.) instead of **void**, and use the **return** keyword inside the method:

Example:

```
public class Main {
    static int myMethod(int x) {
        return 5 + x;
    }

    public static void main(String[] args)
    {System.out.println(myMethod(3));
    }
}
// Outputs 8 (5 + 3)
```

This example returns the sum of a method's **two parameters**:

Example:

```
public class Main {
    static int myMethod(int x, int y)
    {return x + y;
    }

    public static void main(String[] args)
    { System.out.println(myMethod(5,
    3));
    }
}
// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example:

```
public class Main {
    static int myMethod(int x, int y) {
```

```

    return x + y;
}
public static void main(String[] args)
{int z = myMethod(5, 3);
 System.out.println(z);
}
}
// Outputs 8 (5 + 3)

```

A Method with If...Else

It is common to use **if...else** statements inside methods:

Example:

```

public class Main {

// Create a checkAge() method with an integer variable called age
static void checkAge(int age) {

// If age is less than 18, print "access denied"
if (age < 18) {
 System.out.println("Access denied - You are not old enough!");

// If age is greater than, or equal to, 18, print "access granted"
} else {
 System.out.println("Access granted - You are old enough!");
}

}

public static void main(String[] args) {
 checkAge(20); // Call the checkAge method and pass along an age of 20
}
}

// Outputs "Access granted - You are old enough!"

```

JAVA METHOD OVERLOADING

Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Example:

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

Consider the following example, which have two methods that add numbers of different type:

Example:

```
public class Main {
    static int plusMethodInt(int x, int y)
    {return x + y;
    }

    static double plusMethodDouble(double x, double y)
    {return x + y;
    }

    public static void main(String[] args)
    {
        int myNum1 = plusMethodInt(8, 5);
        double myNum2 = plusMethodDouble(4.3, 6.26);
        System.out.println("int: " + myNum1);
        System.out.println("double: " + myNum2);
    }
}
```

- Instead of defining two methods that should do the same thing, it is better to overload one.
- In the example below, we overload the **plusMethod** method to work for both **int** and **double**:

Example:

```
public class Main {  
    static int plusMethod(int x, int y)  
    {return x + y;  
    }  
    static double plusMethod(double x, double y)  
    {return x + y;  
    }
```

```
public static void main(String[] args)  
    {int myNum1 = plusMethod(8, 5);  
    double myNum2 = plusMethod(4.3, 6.26);  
    System.out.println("int: " + myNum1);  
    System.out.println("double: " + myNum2);  
    }  
}
```

JAVA SCOPE

Java Scope

In Java, variables are only accessible inside the region they are created. This is called **scope**.

Method Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared:

Example:

```
public class Main {  
    public static void main(String[] args) {  
  
        // Code here CANNOT use x  
  
        int x = 100;  
  
        // Code here can use x  
        System.out.println(x);  
    }  
}
```

```
}
```

Block Scope

A block of code refers to all of the code between curly braces `{}`. Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared:

Example:

```
public class Main {  
    public static void main(String[] args) {  
  
        // Code here CANNOT use x  
  
        { // This is a block  
  
            // Code here CANNOT use x  
  
            int x = 100;  
  
            // Code here CAN use x  
            System.out.println(x);  
  
        } // The block ends here  
  
        // Code here CANNOT use x  
  
    }  
}
```

A block of code may exist on its own or it can belong to an **if**, **while** or **for** statement. In the case of **for** statements, variables declared in the statement itself are also available inside the block's scope.

JAVA RECURSION

Java Recursion

- Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.
- Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Example:

Use recursion to add all of the numbers up to 10.

```
public class Main {  
    public static void main(String[] args)  
    {  
        int result = sum(10);  
        System.out.println(result);  
    }  
    public static int sum(int k)  
    {  
        if (k > 0) {  
            return k + sum(k - 1);  
        } else  
        {  
            return  
            0;  
        }  
    }  
}
```

Example Explained

When the `sum()` function is called, it adds parameter `k` to the sum of all numbers smaller than `k` and returns the result. When `k` becomes 0, the function just returns 0. When running, the program follows these steps:

10 + sum(9)

10 + (9 + sum(8))

10 + (9 + (8 + sum(7)))

...

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

Since the function does not call itself when **k** is 0, the program stops there and returns the result.

Halting Condition

- Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter **k** becomes 0.
- It is helpful to see a variety of different examples to better understand the concept. In this example, the function adds a range of numbers between a start and an end. The halting condition for this recursive function is when **end** is not greater than **start**:

Example:

Use recursion to add all of the numbers between 5 to 10.

```
public class Main {  
    public static void main(String[] args)  
    {  
        int result = sum(5, 10);  
        System.out.println(result);  
    }  
    public static int sum(int start, int end)  
    {  
        if (end > start) {  
            return end + sum(start, end - 1);  
        } else  
        {  
            return  
            end;  
        }  
    }  
}
```

JAVA OOP

Java What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

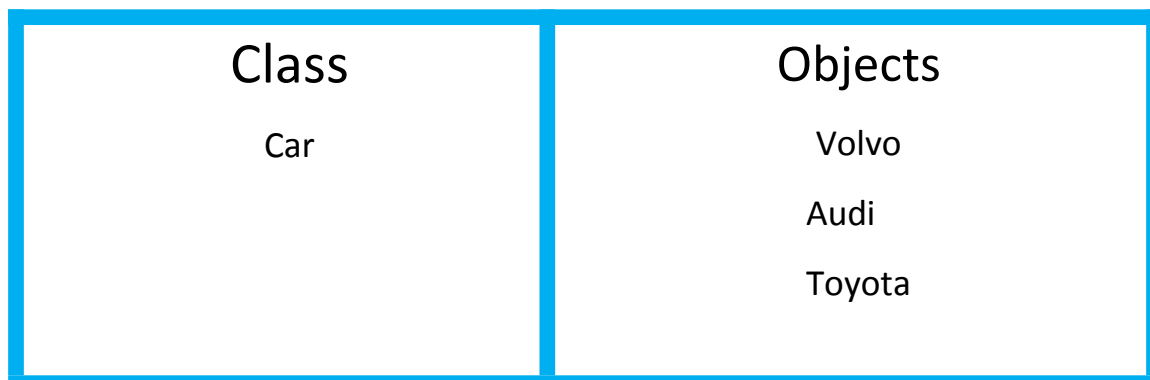
Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

C++ What are Classes and Objects?

- Classes and objects are the two main aspects of object-oriented programming.
- Look at the following illustration to see the difference between class and objects:



Another example:



- So, a class is a template for objects, and an object is an instance of a class.
- When the individual objects are created, they inherit all the variables and functions from the class.
- You will learn much more about classes and objects in the next chapter.

JAVA CLASSES AND OBJECTS

Java Classes/Objects

- Java is an object-oriented programming language.
- Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword **class**:

Main.java

Create a class named "**Main**" with a variable x:

```
public class Main {  
    int x = 5;  
}
```

Remember from the Java Syntax chapter that a class should always start with an uppercase first letter, and that the name of the java file should match the class name.

Create an Object

- In Java, an object is created from a class. We have already created the class named **Main**, so now we can use this to create objects.
- To create an object of **Main**, specify the class name, followed by the object name, and use the keyword **new**:

Example

Create an object called "**myObj**" and print the value of x:

```
public class Main  
{int x = 5;
```

```
public static void main(String[] args)
{
    Main myObj = new Main();
    System.out.println(myObj.x);
}
}
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of **Main**:

```
public class Main
{
    int x = 5;

    public static void main(String[] args)
    {
        Main myObj1 = new Main(); // Object 1
        Main myObj2 = new Main(); // Object 2
        System.out.println(myObj1.x);
        System.out.println(myObj2.x);
    }
}
```

Using Multiple Classes

- You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the **main()** method (code to be executed)).
- Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:
 - Main.java
 - Second.java

Main.java

```
public class Main
{
    int x = 5;
}
```

Second.java

```
class Second {
```

```
public static void main(String[] args)
{Main myObj = new Main();
System.out.println(myObj.x);
}
}
```

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
5
```

JAVA CLASS ATTRIBUTES

Java Class Attributes

In the previous chapter, we used the term "variable" for **x** in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

Example

Create a class called "**Main**" with two attributes: **x** and **y**:

```
public class Main
{int x = 5;
int y = 3;
}
```

Another term for class attributes is **fields**.

Accessing Attributes

- You can access attributes by creating an object of the class, and by using the dot syntax (.):
- The following example will create an object of the **Main** class, with the name **myObj**. We use the **x** attribute on the object to print its value:

Example:

Create an object called "**myObj**" and print the value of **x**:

```
public class Main
{int x = 5;

public static void main(String[] args)
{Main myObj = new Main();
System.out.println(myObj.x);
}
}
```

Modify Attributes

You can also modify attribute values:

Example:

Set the value of **x** to 40:

```
public class Main
{int x;

public static void main(String[] args)
{Main myObj = new Main();
myObj.x = 40;
System.out.println(myObj.x);
}
}
```

Or override existing values:

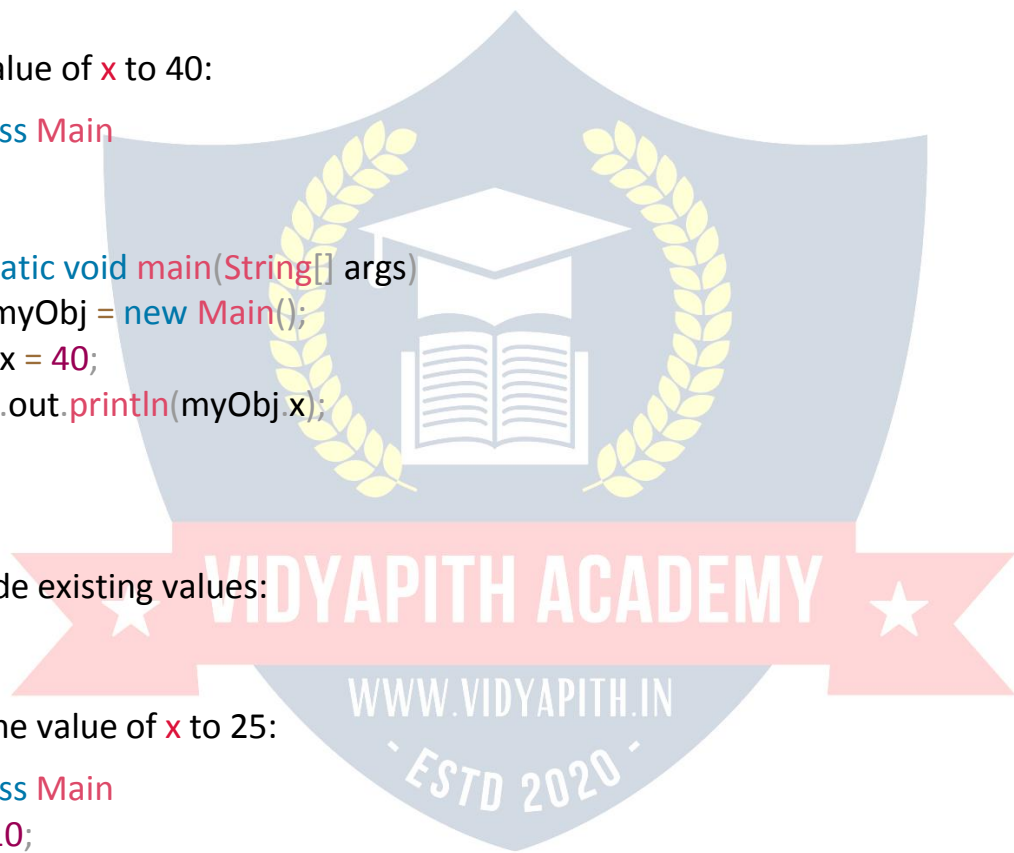
Example:

Change the value of **x** to 25:

```
public class Main
{int x = 10;

public static void main(String[] args)
{Main myObj = new Main();
myObj.x = 25; // x is now 25
System.out.println(myObj.x);
}
}
```

If you don't want the ability to override existing values, declare the attribute as **final**:



Example:

```
public class Main {  
    final int x = 10;  
  
    public static void main(String[] args)  
    {  
        Main myObj = new Main();  
        myObj.x = 25; // will generate an error: cannot assign a value to a final  
variable  
        System.out.println(myObj.x);  
    }  
}
```

The **final** keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

The **final** keyword is called a "modifier". You will learn more about these in the Java Modifiers Chapter.

Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

Example:

Change the value of **x** to 25 in **myObj2**, and leave **x** in **myObj1** unchanged:

```
public class Main {  
    int x = 5;  
  
    public static void main(String[] args)  
    {  
        Main myObj1 = new Main(); // Object 1  
        Main myObj2 = new Main(); // Object 2  
        myObj2.x = 25;  
        System.out.println(myObj1.x); // Outputs 5  
        System.out.println(myObj2.x); // Outputs 25  
    }  
}
```

Multiple Attributes

You can specify as many attributes as you want:

Example:

```
public class Main {
```

```
String fname = "John";
String lname = "Doe";
int age = 24;

public static void main(String[] args)
{
    Main myObj = new Main();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
}
}
```

The next chapter will teach you how to create class methods and how to access them with objects.

JAVA CLASS METHODS

Java Class Methods

You learned from the Java Methods chapter that methods are declared within a class, and that they are used to perform certain actions:

Example:

Create a method named `myMethod()` in `Main`:

```
public class Main {
    static void myMethod()
    {
        System.out.println("Hello
        World!");
    }
}
```

`myMethod()` prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses `()` and a semicolon;

Example:

Inside `main`, call `myMethod()`:

```
public class Main {
    static void myMethod()
    {
        System.out.println("Hello
        World!");
    }
}
```

```

public static void main(String[] args)
{
    myMethod();
}
}

```

// Outputs "Hello World!"

Static vs. Non-Static

- You will often see Java programs that have either **static** or **public** attributes and methods.
- In the example above, we created a **static** method, which means that it can be accessed without creating an object of the class, unlike **public**, which can only be accessed by objects:

Example

An example to demonstrate the differences between **static** and **public methods**:

```

public class Main {
    // Static method
    static void myStaticMethod() {
        System.out.println("Static methods can be called without creating objects");
    }

    // Public method
    public void myPublicMethod() {
        System.out.println("Public methods must be called by creating objects");
    }

    // Main method
    public static void main(String[] args)
    {
        myStaticMethod(); // Call the static method
        // myPublicMethod(); This would compile an error

        Main myObj = new Main(); // Create an object of Main
        myObj.myPublicMethod(); // Call the public method on the object
    }
}

```

Note: You will learn more about these keywords (called modifiers) in the Java Modifiers chapter.

Access Methods With an Object

Example:

Create a Car object named `myCar`. Call the `fullThrottle()` and `speed()` methods on the `myCar` object, and run the program:

```
// Create a Main class
public class Main {

    // Create a fullThrottle() method
    public void fullThrottle() {
        System.out.println("The car is going as fast as it can!");
    }

    // Create a speed() method and add a parameter
    public void speed(int maxSpeed)
    { System.out.println("Max speed is: " + maxSpeed);
    }

    // Inside main, call the methods on the myCar object
    public static void main(String[] args) {
        Main myCar = new Main(); // Create a myCar object
        myCar.fullThrottle(); // Call the fullThrottle() method
        myCar.speed(200); // Call the speed() method
    }
}

// The car is going as fast as it can!
// Max speed is: 200
```

Example explained

- 1) We created a custom `Main` class with the `class` keyword.
- 2) We created the `fullThrottle()` and `speed()` methods in the `Main` class.
- 3) The `fullThrottle()` method and the `speed()` method will print out some text, when they are called.
- 4) The `speed()` method accepts an `int` parameter called `maxSpeed` - we will use this in **8**).
- 5) In order to use the `Main` class and its methods, we need to create an **object** of the `Main` Class.
- 6) Then, go to the `main()` method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).

7) By using the **new** keyword we created an object with the name **myCar**.
8) Then, we call the **fullThrottle()** and **speed()** methods on the **myCar** object, and run the program using the name of the object (**myCar**), followed by a dot (**.**), followed by the name of the method (**fullThrottle();** and **speed(200);**). Notice that we add an **int** parameter of **200** inside the **speed()** method.

Remember that..

- The dot (**.**) is used to access the object's attributes and methods.
- To call a method in Java, write the method name followed by a set of parentheses (**()**), followed by a semicolon (**;**).
- A class must have a matching filename (**Main** and **Main.java**).

Using Multiple Classes

- Like we specified in the Classes chapter, it is a good practice to create an object of a class and access it in another class.
- Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:
 - Main.java
 - Second.java

Main.java

```
public class Main {  
    public void fullThrottle() {  
        System.out.println("The car is going as fast as it can!");  
    }  
  
    public void speed(int maxSpeed)  
    { System.out.println("Max speed is: " +  
        maxSpeed);  
    }  
}
```

Second.java

```
class Second {  
    public static void main(String[] args) {  
        Main myCar = new Main(); // Create a myCar object  
        myCar.fullThrottle(); // Call the fullThrottle() method  
        myCar.speed(200); // Call the speed() method  
    }  
}
```

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
The car is going as fast as it can!
Max speed is: 200
```

JAVA CONSTRUCTORS

Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Example:

Create a constructor:

```
// Create a Main class
```

```
public class Main {
    int x; // Create a class attribute
```

```
// Create a class constructor for the Main class
```

```
public Main() {
    x = 5; // Set the initial value for the class attribute x
}
```

```
public static void main(String[] args) {
```

```
    Main myObj = new Main(); // Create an object of class Main (This will call
the constructor)
```

```
    System.out.println(myObj.x); // Print the value of x
}
```

```
// Outputs 5
```

- Note that the constructor name must **match the class name**, and it cannot have a **return type** (like **void**).
- Also note that the constructor is called when the object is created.
- All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Constructor Parameters

- Constructors can also take parameters, which is used to initialize attributes.
- The following example adds an **int y** parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example:

```
public class Main
```

```
{int x;
```

```
public Main(int y)
```

```
{x = y;
```

```
}
```

```
public static void main(String[] args)
```

```
{Main myObj = new Main(5);
```

```
System.out.println(myObj.x);
```

```
}
```

```
}
```

```
// Outputs 5
```

You can have as many parameters as you want:

Example:

```
public class Main
```

```
{ int modelYear;
```

```
String modelName;
```

```
public Main(int year, String name)
```

```
{modelYear = year;
```

```
modelName = name
```



```

}

public static void main(String[] args) {
    Main myCar = new Main(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
}
}

// Outputs 1969 Mustang

```

JAVA MODIFIERS

Modifiers

By now, you are quite familiar with the **public** keyword that appears in almost all of our examples:

```
public class Main
```

The **public** keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- **Access Modifiers** - controls the access level
- **Non-Access Modifiers** - do not control access level, but provides other functionality

Access Modifiers

For **classes**, you can use either **public** or **default**:

Modifier	Description
public	The class is accessible by any other class
<i>default</i>	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the declared class

<i>default</i>	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter
protected	The code is accessible in the same package and subclasses . You will learn more about subclasses and superclasses in the Inheritance chapter

Non-Access Modifiers

For **classes**, you can use either **final** or **abstract**:

Modifier	Description
final	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)
abstract	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)

For **attributes and methods**, you can use the one of the following:

Modifier	Description
final	Attributes and methods cannot be overridden/modified
static	Attributes and methods belongs to the class, rather than an object
abstract	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example abstract void run() ; The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters
transient	Attributes and methods are skipped when serializing the object containing them
synchronized	Methods can only be accessed by one thread at a time
volatile	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Final

If you don't want the ability to override existing attribute values, declare attributes as **final**:

Example:

```
public class Main {  
    final int x = 10;  
    final double PI = 3.14;  
  
    public static void main(String[] args)  
    {Main myObj = new Main();  
    myObj.x = 50; // will generate an error: cannot assign a value to a final  
variable  
    myObj.PI = 25; // will generate an error: cannot assign a value to a final  
variable  
    System.out.println(myObj.x);  
    }  
}
```

Static

A **static** method means that it can be accessed without creating an object of the class, unlike **public**:

Example:

An example to demonstrate the differences between **static** and **public** methods:

```
public class Main {  
    // Static method  
    static void myStaticMethod() {  
        System.out.println("Static methods can be called without creating objects");  
    }  
  
    // Public method  
    public void myPublicMethod() {  
        System.out.println("Public methods must be called by creating objects");  
    }  
  
    // Main method  
    public static void main(String[ ] args)  
    { myStaticMethod(); // Call the static method  
    // myPublicMethod(); This would output an error  
  
    Main myObj = new Main(); // Create an object of Main  
    myObj.myPublicMethod(); // Call the public method
```

```
}  
}
```

Abstract

An **abstract** method belongs to an **abstract** class, and it does not have a body. The body is provided by the subclass:

Example:

```
// Code from filename: Main.java  
// abstract class  
abstract class Main {  
    public String fname = "John";  
    public int age = 24;  
    public abstract void study(); // abstract method  
}  
  
// Subclass (inherit from Main)  
class Student extends Main  
{ public int graduationYear = 2018;  
  public void study() { // the body of the abstract method is provided here  
    System.out.println("Studying all day long");  
  }  
}  
// End code from filename: Main.java  
  
// Code from filename: Second.java  
class Second {  
    public static void main(String[] args) {  
        // create an object of the Student class (which inherits attributes and  
        methods from Main)  
        Student myObj = new Student();  
  
        System.out.println("Name: " + myObj.fname);  
        System.out.println("Age: " + myObj.age);  
        System.out.println("Graduation Year: " + myObj.graduationYear);  
        myObj.study(); // call abstract method  
    }  
}
```


JAVA ENCAPSULATION

Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as **private**
- provide public **get** and **set** methods to access and update the value of a **private** variable

Get and Set

- You learned from the previous chapter that **private** variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.
- The **get** method returns the variable value, and the **set** method sets the value.
- Syntax for both is that they start with either **get** or **set**, followed by the name of the variable, with the first letter in upper case:

Example:

```
public class Person {  
    private String name; // private = restricted access
```

```
// Getter
```

```
public String getName()  
    {return name;  
}
```

```
// Setter
```

```
public void setName(String newName)  
    {this.name = newName;  
}  
}
```

Example explained

- The **get** method returns the value of the variable **name**.
- The **set** method takes a parameter (**newName**) and assigns it to the **name** variable. The **this** keyword is used to refer to the current object.

- However, as the **name** variable is declared as **private**, we **cannot** access it from outside this class:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        Person myObj = new Person();  
        myObj.name = "John"; // error  
        System.out.println(myObj.name); // error  
    }  
}
```

If the variable was declared as **public**, we would expect the following output:

```
John
```

However, as we try to access a **private** variable, we get an error:

```
MyClass.java:4: error: name has private access in Person  
    myObj.name = "John";  
        ^  
MyClass.java:5: error: name has private access in Person  
    System.out.println(myObj.name);  
                ^  
2 errors
```

Instead, we use the **getName()** and **setName()** methods to access and update the variable:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        Person myObj = new Person();  
        myObj.setName("John"); // Set the value of the name variable to "John"  
        System.out.println(myObj.getName());  
    }  
}
```

```
// Outputs "John"
```

Why Encapsulation?

- Better control of class attributes and methods

- Class attributes can be made **read-only** (if you only use the **get** method), or **write-only** (if you only use the **set** method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

JAVA PACKAGES

Java Packages & API

A package in Java is used to group related classes. Think of it as a **folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

- The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
- The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: <https://docs.oracle.com/javase/8/docs/api/>.
- The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
- To use a class or a package from the library, you need to use the **import** keyword:

Syntax:

```
import package.name.Class; // Import a single class
```

```
import package.name.*; // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the **Scanner** class, **which is used to get user input**, write the following code:

Example:

```
import java.util.Scanner;
```

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Example:

Using the `Scanner` class to get user input:

```
import java.util.Scanner;
```

```
class MyClass {  
    public static void main(String[] args)  
    {  
        Scanner myObj = new  
        Scanner(System.in);  
        System.out.println("Enter username");  
  
        String userName = myObj.nextLine();  
        System.out.println("Username is: " + userName);  
    }  
}
```

Import a Package

- There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date and time facilities, random-number generator and other utility classes.
- To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the `java.util` package:

Example:

```
import java.util.*;
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example:

```
└─ root  
  └─ mypack  
    └─ MyPackageClass.java
```

To create a package, use the `package` keyword:

MyPackageClass.java:

```
package mypack;  
class MyPackageClass {  
    public static void main(String[] args)  
    { System.out.println("This is my  
    package!");  
    }  
}
```

Save the file as **MyPackageClass.java**, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

- This forces the compiler to create the "mypack" package.
- The **-d** keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.
- **Note:** The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```

JAVA INHERITANCE

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the **extends** keyword.

In the example below, the **Car** class (subclass) inherits the attributes and methods from the **Vehicle** class (superclass):

Example:

```
class Vehicle {
    protected String brand = "Ford";    // Vehicle attribute
    public void honk() {                // Vehicle method
        System.out.println("Tuut, tuut!");
    }
}

class Car extends Vehicle {
    private String modelName = "Mustang"; // Car attribute
    public static void main(String[] args) {

        // Create a myCar object
        Car myCar = new Car();

        // Call the honk() method (from the Vehicle class) on the myCar object
        myCar.honk();

        // Display the value of the brand attribute (from the Vehicle class) and the
        // value of the modelName from the Car class
        System.out.println(myCar.brand + " " + myCar.modelName);
    }
}
```

Did you notice the **protected** modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a **protected** access modifier. If it was set to **private**, the Car class would not be able to access it.

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The final Keyword

If you don't want other classes to inherit from a class, use the **final** keyword:

If you try to access a **final** class, Java will generate an error:

```
final class Vehicle {  
    ...  
}
```

```
class Car extends Vehicle {  
    ...  
}
```

The output will be something like this:

```
Main.java:9: error: cannot inherit from final Vehicle
```

```
class Main extends Vehicle {  
    ^
```

```
1 error)
```

JAVA POLYMORPHISM

Java Polymorphism

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.
- For example, think of a superclass called **Animal** that has a method called **animalSound()**. Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example:

```
class Animal {  
    public void animalSound()  
    { System.out.println("The animal makes a sound");  
    }  
}
```

```
class Pig extends Animal  
{ public void animalSound()  
{  
    System.out.println("The pig says: wee wee");
```

```
}  
}
```

```
class Dog extends Animal  
{ public void animalSound()  
{  
    System.out.println("The dog says: bow wow");  
}  
}
```

Remember from the Inheritance chapter that we use the **extends** keyword to inherit from a class.

Now we can create **Pig** and **Dog** objects and call the **animalSound()** method on both of them:

Example:

```
class Animal {  
    public void animalSound()  
    { System.out.println("The animal makes a sound");  
    }  
}
```

```
class Pig extends Animal  
{ public void animalSound()  
{  
    System.out.println("The pig says: wee wee");  
}  
}
```

```
class Dog extends Animal  
{ public void animalSound()  
{  
    System.out.println("The dog says: bow wow");  
}  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Animal myAnimal = new Animal(); // Create a Animal object
```



```

Animal myPig = new Pig(); // Create a Pig object
Animal myDog = new Dog(); // Create a Dog object
myAnimal.animalSound();
myPig.animalSound();
myDog.animalSound();
}
}

```

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

JAVA INNER CLASSES

Java Inner Classes

- In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
- To access the inner class, create an object of the outer class, and then create an object of the inner class:

Example:

```
class OuterClass
```

```
{int x = 10;
```

```
class InnerClass
```

```
{int y = 5;
```

```
}
```

```
}
```

```
public class Main {
```

```
public static void main(String[] args)
```

```
{ OuterClass myOuter = new
```

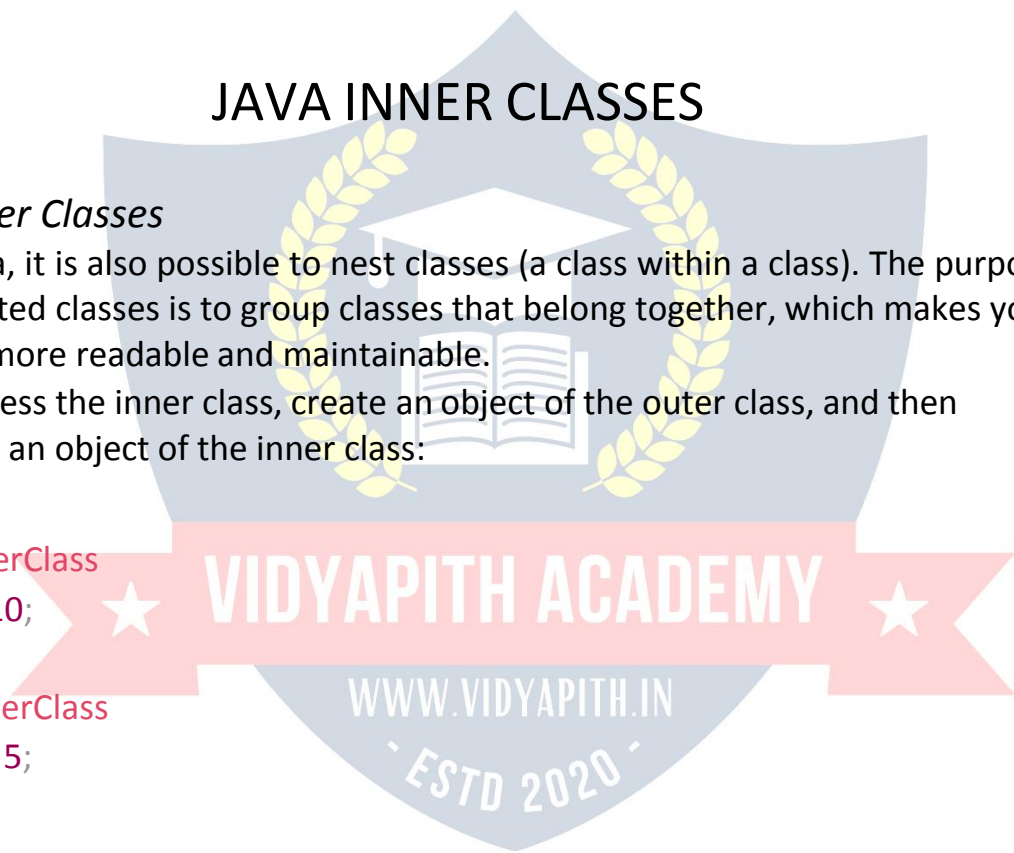
```
OuterClass();
```

```
OuterClass.InnerClass myInner = myOuter.new InnerClass();
```

```
System.out.println(myInner.y + myOuter.x);
```

```
}
```

```
}
```



```
// Outputs 15 (5 + 10)
```

Private Inner Class

Unlike a "regular" class, an inner class can be **private** or **protected**. If you don't want outside objects to access the inner class, declare the class as **private**:

Example:

```
class OuterClass
```

```
{int x = 10;
```

```
private class InnerClass
```

```
{int y = 5;
```

```
}
```

```
}
```

```
public class Main {
```

```
public static void main(String[] args)
```

```
{ OuterClass myOuter = new
```

```
OuterClass();
```

```
OuterClass.InnerClass myInner = myOuter.new InnerClass();
```

```
System.out.println(myInner.y + myOuter.x);
```

```
}
```

```
}
```

If you try to access a private inner class from an outside class, an error occurs:

```
Main.java:13: error: OuterClass.InnerClass has private access in OuterClass
OuterClass.InnerClass myInner = myOuter.new InnerClass();
```

```
^
```

Static Inner Class

An inner class can also be **static**, which means that you can access it without creating an object of the outer class:

Example:

```
class OuterClass
```

```
{int x = 10;
```

```
static class InnerClass
```

```
{int y = 5;
```

```

}
}

public class Main {
    public static void main(String[] args) {
        OuterClass.InnerClass myInner = new OuterClass.InnerClass();
        System.out.println(myInner.y);
    }
}

```

// Outputs 5

Note: just like **static** attributes and methods, a **static** inner class does not have access to members of the outer class.

Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

Example:

```

class OuterClass
{int x = 10;

```

```

class InnerClass {
    public int myInnerMethod()
    {return x;
    }
}
}

```

```

public class Main {
    public static void main(String[] args)
    { OuterClass myOuter = new
    OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
    }
}

```

// Outputs 10

JAVA ABSTRACTION

Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The **abstract** keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {  
    public abstract void animalSound();  
    public void sleep()  
    { System.out.println("Zzz");  
    }  
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class:

Remember from the Inheritance chapter that we use the **extends** keyword to inherit from a class.

Example:

```
// Abstract class  
abstract class Animal {  
    // Abstract method (does not have a body)  
    public abstract void animalSound();  
    // Regular method  
    public void sleep() {  
        System.out.println("Zzz");  
    }  
}
```

```

}
}

// Subclass (inherit from Animal)
class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}
}

```

```

class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}

```

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Note: Abstraction can also be achieved with Interfaces, which you will learn more about in the next chapter.

JAVA INTERFACE

Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An **interface** is a completely "**abstract class**" that is used to group related methods with empty bodies:

Example:

```

// interface
interface Animal
    public void animalSound(); // interface method (does not have a body)

```

```
public void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the **implements** keyword (instead of **extends**). The body of the interface method is provided by the "implement" class:

Example:

```
// Interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)
}
```

```
// Pig "implements" the Animal interface
class Pig implements Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
    public void sleep() {
        // The body of sleep() is provided here
        System.out.println("Zzz");
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```



Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default **abstract** and **public**
- Interface attributes are by default **public**, **static** and **final**
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

- 1) To achieve security - hide certain details and only show the important details of an object (interface).
- 2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example:

```
interface FirstInterface {  
    public void myMethod(); // interface method  
}
```

```
interface SecondInterface {  
    public void myOtherMethod(); // interface method  
}
```

```
class DemoClass implements FirstInterface, SecondInterface
```

```
{  
    public void myMethod() {  
        System.out.println("Some text..");  
    }  
}
```

```
public void myOtherMethod()  
{  
    System.out.println("Some other  
text..");  
}
```

```

}
}

class Main {
    public static void main(String[] args)
    { DemoClass myObj = new DemoClass();
      myObj.myMethod();
      myObj.myOtherMethod();
    }
}

```

JAVA ENUMS

Enums

An **enum** is a special "class" that represents a group of **constants** (unchangeable variables, like **final** variables).

To create an **enum**, use the **enum** keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

Example:

```

enum Level
{LOW,
MEDIUM,
HIGH
}

```

You can access **enum** constants with the **dot** syntax:

```
Level myVar = Level.MEDIUM;
```

Enum is short for "enumerations", which means "specifically listed".

Enum inside a Class

You can also have an **enum** inside a class:

Example:

```

public class Main
{enum Level {

```



```

LOW,
MEDIUM,
HIGH
}

public static void main(String[] args)
{
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
}
}

```

The output will be:

```
MEDIUM
```

Enum in a Switch Statement

Enums are often used in **switch** statements to check for corresponding values:

Example:

```

enum Level
{
LOW,
MEDIUM,
HIGH
}

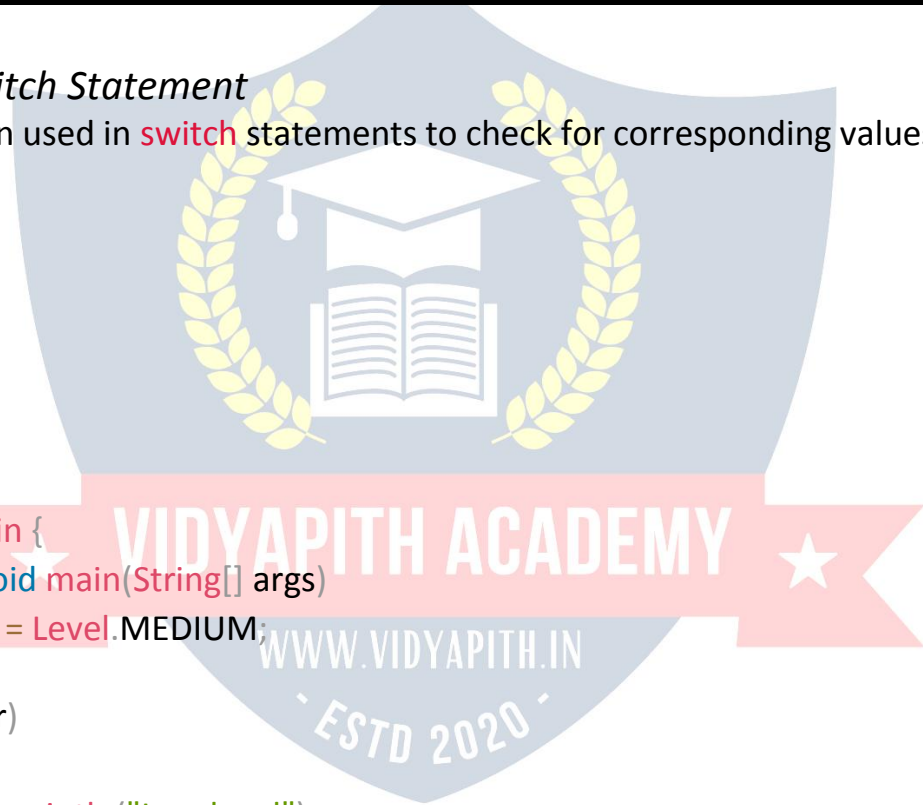
```

```

public class Main {
public static void main(String[] args)
{
    Level myVar = Level.MEDIUM;

switch(myVar)
{
case LOW:
    System.out.println("Low level");
    break;
case MEDIUM:
    System.out.println("Medium level");
    break;
case HIGH:
    System.out.println("High level");
    break;
}
}
}

```



```
}  
}  
}
```

The output will be:

```
Medium level
```

Loop Through an Enum

The enum type has a `values()` method, which returns an array of all enum constants. This method is useful when you want to loop through the constants of an enum:

Example:

```
enum Level  
{LOW,  
MEDIUM,  
HIGH  
}
```

```
public class Main {  
    public static void main(String[] args)  
        (for (Level myVar : Level.values())  
        { System.out.println(myVar);  
        }  
    }  
}
```

The output will be:

```
LOW  
MEDIUM  
HIGH
```

Difference between Enums and Classes

An `enum` can, just like a `class`, have attributes and methods. The only difference is that enum constants are `public`, `static` and `final` (unchangeable - cannot be overridden).

An `enum` cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why And When To Use Enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

JAVA USER INPUT (SCANNER)

Java User Input

The **Scanner** class is used to get user input, and it is found in the **java.util** package.

To use the **Scanner** class, create an object of the class and use any of the available methods found in the **Scanner** class documentation. In our example, we will use the **nextLine()** method, which is used to read Strings:

Example:

```
import java.util.Scanner; // Import the Scanner class
```

```
class Main {  
    public static void main(String[] args) {  
        Scanner myObj = new Scanner(System.in); // Create a Scanner object  
        System.out.println("Enter username");  
  
        String userName = myObj.nextLine(); // Read user input  
        System.out.println("Username is: " + userName); // Output user input  
    }  
}
```

Input Types

In the example above, we used the **nextLine()** method, which is used to read Strings. To read other types, look at the table below:

Method	Description
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

In the example below, we use different methods to read data of various types:

Example:

```
import java.util.Scanner;

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

        System.out.println("Enter name, age and salary:");

        // String input
        String name = myObj.nextLine();

        // Numerical input
        int age = myObj.nextInt();
        double salary = myObj.nextDouble();

        // Output input by user
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}
```

Note: If you enter wrong input (e.g. text in a numerical input), you will get an exception/error message (like "InputMismatchException").

You can read more about exceptions and how to handle errors in the Exceptions chapter.

JAVA DATE AND TIME

Java Dates

Java does not have a built-in Date class, but we can import the **java.time** package to work with the date and time API. The package includes many date and time classes. For example:

Class	Description
LocalDate	Represents a date (year, month, day (yyyy-MM-dd))

LocalTime	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
LocalDateTime	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
DateTimeFormatter	Formatter for displaying and parsing date-time objects

Display Current Date

To display the current date, import the `java.time.LocalDate` class, and use its `now()` method:

Example:

```
import java.time.LocalDate; // import the LocalDate class
```

```
public class Main {
    public static void main(String[] args) {
        LocalDate myObj = LocalDate.now(); // Create a date object
        System.out.println(myObj); // Display the current date
    }
}
```

The output will be:

```
2021-08-27
```

Display Current Time

To display the current time (hour, minute, second, and nanoseconds), import the `java.time.LocalTime` class, and use its `now()` method:

Example:

```
import java.time.LocalTime; // import the LocalTime class
```

```
public class Main {
    public static void main(String[] args)
    {
        LocalTime myObj =
        LocalTime.now();
        System.out.println(myObj);
    }
}
```

The output will be:

```
15:55:01.112130
```

Display Current Date and Time

To display the current date and time, import the `java.time.LocalDateTime` class, and use its `now()` method:

Example:

```
import java.time.LocalDateTime; // import the LocalDateTime class
```

```
public class Main {  
    public static void main(String[] args)  
    {  
        LocalDateTime myObj =  
            LocalDateTime.now();  
        System.out.println(myObj);  
    }  
}
```

The output will be:

```
2021-08-27T15:55:01.136549
```

Formatting Date and Time

The "T" in the example above is used to separate the date from the time. You can use the `DateTimeFormatter` class with the `ofPattern()` method in the same package to format or parse date-time objects. The following example will remove both the "T" and nanoseconds from the date-time:

Example:

```
import java.time.LocalDateTime; // Import the LocalDateTime class  
import java.time.format.DateTimeFormatter; // Import the  
DateTimeFormatter class
```

```
public class Main {  
    public static void main(String[] args)  
    {  
        LocalDateTime myDateObj =  
            LocalDateTime.now();  
        System.out.println("Before formatting: " + myDateObj);  
        DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM  
yyyy HH:mm:ss");
```

```
        String formattedDate = myDateObj.format(myFormatObj);
```

```
        System.out.println("After formatting: " + formattedDate);
```

```
    }  
}
```

The output will be:

Before Formatting: 2021-08-27T15:55:01.136965

After Formatting: 27-08-2021 15:55:01

The `ofPattern()` method accepts all sorts of values, if you want to display the date and time in a different format. For example:

Value	Example
<code>yyyy-MM-dd</code>	"1988-09-29"
<code>dd/MM/yyyy</code>	"29/09/1988"
<code>dd-MMM-yyyy</code>	"29-Sep-1988"
<code>E, MMM dd yyyy</code>	"Thu, Sep 29 1988"

JAVA ARRAYLIST

Java ArrayList

The `ArrayList` class is a resizable array, which can be found in the `java.util` package.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want. The syntax is also slightly different:

Example:

Create an `ArrayList` object called `cars` that will store strings:

```
import java.util.ArrayList; // import the ArrayList class
```

```
ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Add Items

The `ArrayList` class has many useful methods. For example, to add elements to the `ArrayList`, use the `add()` method:

Example:

```
import java.util.ArrayList;
```

```
public class Main {  
    public static void main(String[] args) {
```

```

ArrayList<String> cars = new ArrayList<String>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");
System.out.println(cars);
}
}

```

Access an Item

To access an element in the **ArrayList**, use the **get()** method and refer to the index number:

Example:

```

import java.util.ArrayList;

public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
      cars.add("Mazda");
      System.out.println(cars.get(0));
    }
}

```

Remember: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Item

To modify an element, use the **set()** method and refer to the index number:

Example:

```

import java.util.ArrayList;

public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");

```



```
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");
cars.set(0, "Opel");
System.out.println(cars);
}
}
```

Remove an Item

To remove an element, use the `remove()` method and refer to the index number:

Example:

```
import java.util.ArrayList;
```

```
public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
      cars.add("Mazda");
      cars.remove(0);
      System.out.println(cars);
    }
}
```

To remove all the elements in the `ArrayList`, use the `clear()` method:

Example:

```
import java.util.ArrayList;
```

```
public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
      cars.add("Mazda");
    }
}
```

```
cars.clear();
System.out.println(cars);
}
}
```

ArrayList Size

To find out how many elements an ArrayList have, use the **size** method:

Example:

```
import java.util.ArrayList;
```

```
public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
      cars.add("Mazda");
      System.out.println(cars.size());
    }
}
```

Loop Through an ArrayList

Loop through the elements of an **ArrayList** with a **for** loop, and use the **size()** method to specify how many times the loop should run:

Example:

```
public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
      cars.add("Mazda");
      for (int i = 0; i < cars.size(); i++)
          {System.out.println(cars.get(i));
            }
    }
}
```

You can also loop through an **ArrayList** with the **for-each** loop:

Example:

```
public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
      cars.add("Mazda");
      for (String i : cars)
      { System.out.println(i)
      ;
      }
    }
}
```

Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: **Integer**. For other primitive types, use: **Boolean** for boolean, **Character** for char, **Double** for double, etc:

Example:

Create an **ArrayList** to store numbers (add elements of type **Integer**):

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> myNumbers = new ArrayList<Integer>();
        myNumbers.add(10);
        myNumbers.add(15);
        myNumbers.add(20);
        myNumbers.add(25);
        for (int i : myNumbers)
        {System.out.println(i);
        }
    }
}
```

Sort an ArrayList

Another useful class in the `java.util` package is the `Collections` class, which include the `sort()` method for sorting lists alphabetically or numerically:

Example:

Sort an ArrayList of Strings:

```
import java.util.ArrayList;
import java.util.Collections; // Import the Collections class
```

```
public class Main {
    public static void main(String[] args)
    { ArrayList<String> cars = new ArrayList<String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
      cars.add("Mazda");
      Collections.sort(cars); // Sort cars
      for (String i : cars)
      { System.out.println(i);
      }
    }
}
```

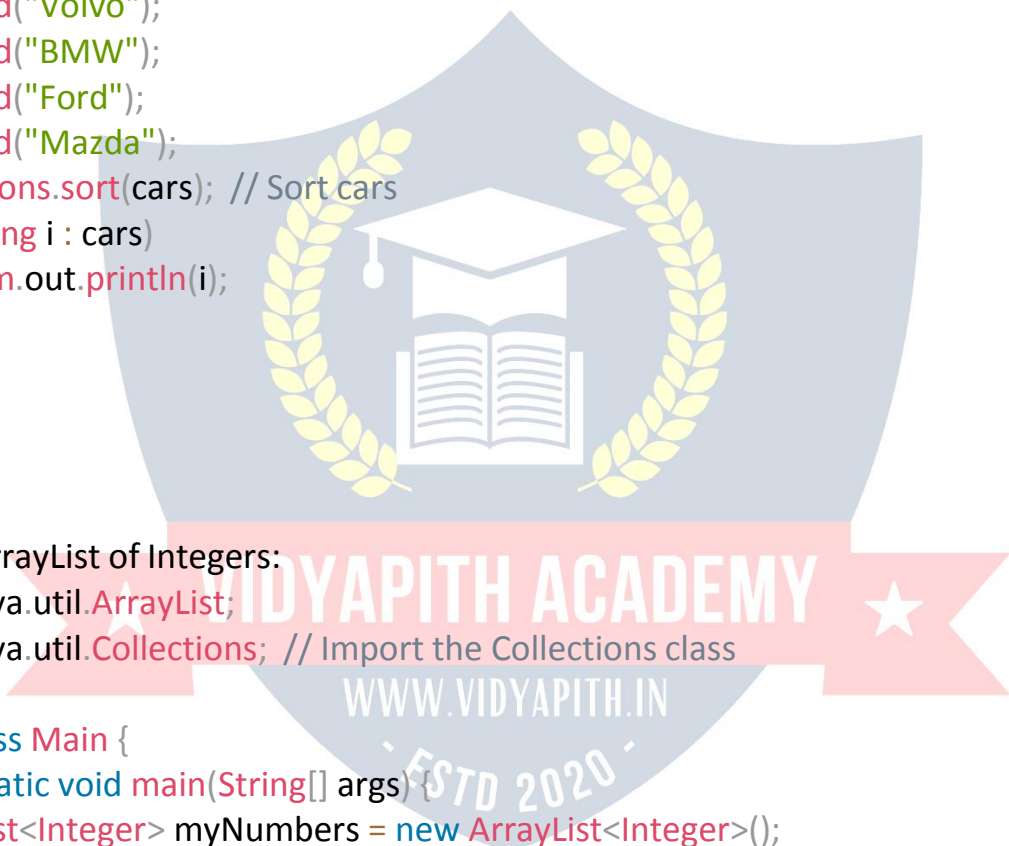
Example:

Sort an ArrayList of Integers:

```
import java.util.ArrayList;
import java.util.Collections; // Import the Collections class
```

```
public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> myNumbers = new ArrayList<Integer>();
        myNumbers.add(33);
        myNumbers.add(15);
        myNumbers.add(20);
        myNumbers.add(34);
        myNumbers.add(8);
        myNumbers.add(12);
```

```
        Collections.sort(myNumbers); // Sort myNumbers
        for (int i : myNumbers) {
            System.out.println(i);
```



```
}  
}  
}
```

JAVA LINKEDLIST

Java LinkedList

In the previous chapter, you learned about the `ArrayList` class.

The `LinkedList` class is almost identical to the `ArrayList`:

Example:

```
// Import the LinkedList class  
import java.util.LinkedList;  
public class Main {  
    public static void main(String[] args)  
    { LinkedList<String> cars = new LinkedList<String>();  
      cars.add("Volvo");  
      cars.add("BMW");  
      cars.add("Ford");  
      cars.add("Mazda");  
      System.out.println(cars);  
    }  
}
```

ArrayList vs. LinkedList

The `LinkedList` class is a collection which can contain many objects of the same type, just like the `ArrayList`.

The `LinkedList` class has all of the same methods as the `ArrayList` class because they both implement the `List` interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the `ArrayList` class and the `LinkedList` class can be used in the same way, they are built very differently.

How the ArrayList works

The `ArrayList` class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

How the LinkedList works

The **LinkedList** stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

When To Use

It is best to use an **ArrayList** when:

- You want to access random items frequently
- You only need to add or remove elements at the end of the list

It is best to use a **LinkedList** when:

- You only use the list by looping through it instead of accessing random items
- You frequently need to add and remove items from the beginning, middle or end of the list

LinkedList Methods

For many cases, the **ArrayList** is more efficient as it is common to need access to random items in the list, but the **LinkedList** provides several methods to do certain operations more efficiently:

Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

JAVA HASHMAP

Java HashMap

In the **ArrayList** chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (**int** type).

A **HashMap** however, store items in "**key/value**" pairs, and you can access them by an index of another type (e.g. a **String**).

One object is used as a key (index) to another object (value). It can store different types: **String** keys and **Integer** values, or the same type, like: **String** keys and **String** values:

Example:

Create a **HashMap** object called **capitalCities** that will store **String** keys and **String** values:

```
import java.util.HashMap; // import the HashMap class
```

```
HashMap<String, String> capitalCities = new HashMap<String, String>();
```

Add Items

The **HashMap** class has many useful methods. For example, to add items to it, use the **put()** method:

Example:

```
// Import the HashMap class
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        // Create a HashMap object called capitalCities
        HashMap<String, String> capitalCities = new HashMap<String, String>();

        // Add keys and values (Country, City)
        capitalCities.put("England", "London");
        capitalCities.put("Germany", "Berlin");
        capitalCities.put("Norway", "Oslo");
        capitalCities.put("USA", "Washington DC");
        System.out.println(capitalCities);
    }
}
```

Access an Item

To access a value in the **HashMap**, use the **get()** method and refer to its key:

Example:

```
import java.util.HashMap;
```

```

public class Main {
    public static void main(String[] args) {
        HashMap<String, String> capitalCities = new HashMap<String, String>();
        capitalCities.put("England", "London");
        capitalCities.put("Germany", "Berlin");
        capitalCities.put("Norway", "Oslo");
        capitalCities.put("USA", "Washington DC");
        System.out.println(capitalCities.get("England"));
    }
}

```

Remove an Item

To remove an item, use the `remove()` method and refer to the key:

Example:

```

import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, String> capitalCities = new HashMap<String, String>();
        capitalCities.put("England", "London");
        capitalCities.put("Germany", "Berlin");
        capitalCities.put("Norway", "Oslo");
        capitalCities.put("USA", "Washington DC");
        capitalCities.remove("England");
        System.out.println(capitalCities);
    }
}

```

To remove all items, use the `clear()` method:

Example:

```

import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, String> capitalCities = new HashMap<String, String>();
        capitalCities.put("England", "London");
        capitalCities.put("Germany", "Berlin");
        capitalCities.put("Norway", "Oslo");
    }
}

```



```

capitalCities.put("USA", "Washington DC");
capitalCities.clear();
System.out.println(capitalCities);
}
}

```

HashMap Size

To find out how many items there are, use the `size()` method:

Example:

```
import java.util.HashMap;
```

```

public class Main {
public static void main(String[] args) {
HashMap<String, String> capitalCities = new HashMap<String, String>();
capitalCities.put("England", "London");
capitalCities.put("Germany", "Berlin");
capitalCities.put("Norway", "Oslo");
capitalCities.put("USA", "Washington DC");
System.out.println(capitalCities.size());
}
}

```

Loop Through a HashMap

Loop through the items of a `HashMap` with a **for-each** loop.

Note: Use the `keySet()` method if you only want the keys, and use the `values()` method if you only want the values:

Example:

```
import java.util.HashMap;
```

```

public class Main {
public static void main(String[] args) {
HashMap<String, String> capitalCities = new HashMap<String, String>();
capitalCities.put("England", "London");
capitalCities.put("Norway", "Oslo");
capitalCities.put("USA", "Washington DC");

```

```
for (String i : capitalCities.keySet()) {
```

```
    System.out.println(i);
  }
}
}
```

Example:

```
import java.util.HashMap;
```

```
public class Main {
    public static void main(String[] args) {
        HashMap<String, String> capitalCities = new HashMap<String, String>();
        capitalCities.put("England", "London");
        capitalCities.put("Germany", "Berlin");
        capitalCities.put("Norway", "Oslo");
        capitalCities.put("USA", "Washington DC");

        for (String i : capitalCities.values())
        {System.out.println(i);
        }
    }
}
```

Example:

```
import java.util.HashMap;
```

```
public class Main {
    public static void main(String[] args) {

        HashMap<String, String> capitalCities = new HashMap<String, String>();
        capitalCities.put("England", "London");
        capitalCities.put("Germany", "Berlin");
        capitalCities.put("Norway", "Oslo");
        capitalCities.put("USA", "Washington DC");

        for (String i : capitalCities.keySet()) {
            System.out.println("key: " + i + " value: " + capitalCities.get(i));
        }
    }
}
```

```
}  
}
```

Other Types

Keys and values in a HashMap are actually objects. In the examples above, we used objects of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: **Integer**. For other primitive types, use: **Boolean** for boolean, **Character** for char, **Double** for double, etc:

Example:

Create a **HashMap** object called **people** that will store **String keys** and **Integer values**:

```
// Import the HashMap class  
import java.util.HashMap;  
  
public class Main {  
    public static void main(String[] args) {  
  
        // Create a HashMap object called people  
        HashMap<String, Integer> people = new HashMap<String, Integer>();  
  
        // Add keys and values (Name, Age)  
        people.put("John", 32);  
        people.put("Steve", 30);  
        people.put("Angie", 33);  
  
        for (String i : people.keySet()) {  
            System.out.println("key: " + i + " value: " + people.get(i));  
        }  
    }  
}
```

JAVA HASHSET

Java HashSet

A HashSet is a collection of items where every item is unique, and it is found in the **java.util** package:

Example:

Create a `HashSet` object called `cars` that will store strings:

```
import java.util.HashSet; // Import the HashSet class
```

```
HashSet<String> cars = new HashSet<String>();
```

Add Items

The `HashSet` class has many useful methods. For example, to add items to it, use the `add()` method:

Example:

```
// Import the HashSet class
```

```
import java.util.HashSet;
```

```
public class Main {  
    public static void main(String[] args)  
    {  
        HashSet<String> cars = new HashSet<String>();  
        cars.add("Volvo");  
        cars.add("BMW");  
        cars.add("Ford");  
        cars.add("BMW");  
        cars.add("Mazda");  
        System.out.println(cars);  
    }  
}
```

Note: In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

Check If an Item Exists

To check whether an item exists in a `HashSet`, use the `contains()` method:

Example:

```
// Import the HashSet class
```

```
import java.util.HashSet;
```

```
public class Main {  
    public static void main(String[] args)  
    {  
        HashSet<String> cars = new HashSet<String>();
```

```
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("BMW");
cars.add("Mazda");
System.out.println(cars.contains("Mazda"));
}
}
```

Remove an Item

To remove an item, use the **remove()** method:

Example:

```
// Import the HashSet class
```

```
import java.util.HashSet;
```

```
public class Main {
public static void main(String[] args)
{ HashSet<String> cars = new HashSet<String>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("BMW");
cars.add("Mazda");
cars.remove("Volvo");
System.out.println(cars);
}
}
```

To remove all items, use the **clear()** method:

Example:

```
// Import the HashSet class
```

```
import java.util.HashSet;
```

```
public class Main {
public static void main(String[] args)
{ HashSet<String> cars = new HashSet<String>();
cars.add("Volvo");
```

```

cars.add("BMW");
cars.add("Ford");
cars.add("BMW");
cars.add("Mazda");
cars.clear();
System.out.println(cars);
}
}

```

HashSet Size

To find out how many items there are, use the **size** method:

Example:

```
// Import the HashSet class
```

```
import java.util.HashSet;
```

```

public class Main {
public static void main(String[] args)
{ HashSet<String> cars = new HashSet<String>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("BMW");
cars.add("Mazda");
System.out.println(cars.size());
}
}

```

Loop Through a HashSet

Loop through the items of an **HashSet** with a **for-each** loop:

Example:

```
// Import the HashSet class
```

```
import java.util.HashSet;
```

```

public class Main {
public static void main(String[] args)
{ HashSet<String> cars = new HashSet<String>();
cars.add("Volvo");
}
}

```

```

cars.add("BMW");
cars.add("Ford");
cars.add("BMW");
cars.add("Mazda");
for (String i : cars)
{ System.out.println(i)
;
}
}
}

```

Other Types

Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: **Integer**. For other primitive types, use: **Boolean** for boolean, **Character** for char, **Double** for double, etc:

Example:

Use a **HashSet** that stores **Integer** objects:

```

import java.util.HashSet;

public class Main {
public static void main(String[] args) {

// Create a HashSet object called numbers
HashSet<Integer> numbers = new HashSet<Integer>();

// Add values to the set
numbers.add(4);
numbers.add(7);
numbers.add(8);

// Show which numbers between 1 and 10 are in the set
for(int i = 1; i <= 10; i++) {
if(numbers.contains(i)) {
System.out.println(i + " was found in the set.");
} else {
System.out.println(i + " was not found in the set.");
}
}
}

```

```
}  
}  
}
```

JAVA ITERATOR

Java Iterator

An **Iterator** is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the **java.util** package.

Getting an Iterator

The **iterator()** method can be used to get an **Iterator** for any collection:

Example:

```
// Import the ArrayList class and the Iterator class
```

```
import java.util.ArrayList;
```

```
import java.util.Iterator;
```

```
public class Main {  
    public static void main(String[] args) {
```

```
        // Make a collection
```

```
        ArrayList<String> cars = new ArrayList<String>();
```

```
        cars.add("Volvo");
```

```
        cars.add("BMW");
```

```
        cars.add("Ford");
```

```
        cars.add("Mazda");
```

```
        // Get the iterator
```

```
        Iterator<String> it = cars.iterator();
```

```
        // Print the first item
```

```
        System.out.println(it.next());
```

```
    }  
}
```



Looping Through a Collection

To loop through a collection, use the `hasNext()` and `next()` methods of the `Iterator`:

Example:

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
public class Main {
    public static void main(String[] args) {

        // Make a collection
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");

        // Get the iterator
        Iterator<String> it = cars.iterator();

        // Loop through a collection
        while(it.hasNext())
        { System.out.println(it.next());
        }
    }
}
```

Removing Items from a Collection

Iterators are designed to easily change the collections that they loop through. The `remove()` method can remove items from a collection while looping.

Example:

Use an iterator to remove numbers less than 10 from a collection:

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
```

```

numbers.add(12);
numbers.add(8);
numbers.add(2);
numbers.add(23);
Iterator<Integer> it = numbers.iterator();
while(it.hasNext()) {
    Integer i = it.next();
    if(i < 10) {
        it.remove();
    }
}
System.out.println(numbers);
}
}

```

Note: Trying to remove items using a **for loop** or a **for-each loop** would not work correctly because the collection is changing size at the same time that the code is trying to loop.

JAVA WRAPPER CLASSES

Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (**int**, **boolean**, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Sometimes you must use wrapper classes, for example when working with Collection objects, such as **ArrayList**, where primitive types cannot be used (the list can only store objects):

Example:

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid
```

```
ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

Creating Wrapper Objects

To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        Integer myInt = 5;  
        Double myDouble = 5.99;  
        Character myChar = 'A';  
        System.out.println(myInt);  
        System.out.println(myDouble);  
        System.out.println(myChar);  
    }  
}
```

Since you're now working with objects, you can use certain methods to get information about the specific object.

For example, the following methods are used to get the value associated with the corresponding wrapper object: `intValue()`, `byteValue()`, `shortValue()`, `longValue()`, `floatValue()`, `doubleValue()`, `charValue()`, `booleanValue()`.

This example will output the same result as the example above:

Example:

```
public class Main {  
    public static void main(String[] args)  
    {  
        Integer myInt = 5;  
        Double myDouble = 5.99;  
        Character myChar = 'A';  
        System.out.println(myInt.intValue());  
        System.out.println(myDouble.doubleValue());  
        System.out.println(myChar.charValue());  
    }  
}
```

Another useful method is the `toString()` method, which is used to convert wrapper objects to strings.

In the following example, we convert an `Integer` to a `String`, and use the `length()` method of the `String` class to output the length of the "string":

Example:

```
public class Main {
    public static void main(String[] args)
    {
        Integer myInt = 100;
        String myString = myInt.toString();
        System.out.println(myString.length());
    }
}
```

JAVA EXCEPTIONS - TRY...CATCH

Java Exceptions

- When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
- When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

Java try and catch

- The `try` statement allows you to define a block of code to be tested for errors while it is being executed.
- The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The `try` and `catch` keywords come in pairs:

Syntax:

```
try {
    // Block of code to try
}
catch(Exception e) {
    // Block of code to handle errors
}
```

Consider the following example:

This will generate an error, because **myNumbers[10]** does not exist.

```
public class Main {
    public static void main(String[] args)
    {int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
    }
}
```

The output will be something like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at Main.main(Main.java:4)
```

If an error occurs, we can use **try...catch** to catch the error and execute some code to handle it:

Example:

```
public class Main {
    public static void main(String[] args)
    {try {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]);
    } catch (Exception e)
    { System.out.println("Something went
    wrong.");
    }
    }
}
```

The output will be:

```
Something went wrong.
```

Finally

The **finally** statement lets you execute code, after **try...catch**, regardless of the result:

Example:

```
public class Main {
    public static void main(String[] args)
    {try {
        int[] myNumbers = {1, 2, 3};
        System.out.println(myNumbers[10]);
    } catch (Exception e)
    { System.out.println("Something went
    wrong.");
    } finally {
        System.out.println("The 'try catch' is finished.");
    }
    }
}
```

The output will be:

```
Something went wrong.
The 'try catch' is finished.
```

The throw keyword

- The **throw** statement allows you to create a custom error.
- The **throw** statement is used together with an **exception type**. There are many exception types available in Java: **ArithmeticException**, **FileNotFoundException**, **ArrayIndexOutOfBoundsException**, **SecurityException**, etc:

Example:

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {
    static void checkAge(int age)
    {if (age < 18) {
        throw new ArithmeticException("Access denied - You must be at least 18
    years old.");
    }
    else {
        System.out.println("Access granted - You are old enough!");
    }
    }
}
```

```
public static void main(String[] args) {  
    checkAge(15); // Set age to 15 (which is below 18...)  
}  
}
```

The output will be:

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You  
must be at least 18 years old.  
    at Main.checkAge(Main.java:4)  
    at Main.main(Main.java:12)
```

If age was 20, you would **not** get an exception:

Example:

```
public class Main {  
    static void checkAge(int age)  
    { if (age < 18) {  
        throw new ArithmeticException("Access denied - You must be at least 18  
years old.");  
    }  
    else {  
        System.out.println("Access granted - You are old enough!");  
    }  
}  
  
public static void main(String[] args)  
    { checkAge(20);  
    }  
}
```

The output will be:

```
Access granted - You are old enough!
```

JAVA REGULAR EXPRESSIONS

What is a Regular Expression?

- A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

- A regular expression can be a single character, or a more complicated pattern.
- Regular expressions can be used to perform all types of **text search** and **text replace** operations.
- Java does not have a built-in Regular Expression class, but we can import the `java.util.regex` package to work with regular expressions. The package includes the following classes:
 - **Pattern** Class - Defines a pattern (to be used in a search)
 - **Matcher** Class - Used to search for the pattern
 - **PatternSyntaxException** Class - Indicates syntax error in a regular expression pattern

Example:

Find out if there are any occurrences of the word "w3schools" in a sentence:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher("Visit W3Schools!");
        boolean matchFound = matcher.find();
        if(matchFound)
        { System.out.println("Match found");
        } else {
            System.out.println("Match not found");
        }
    }
}
// Outputs Match found
```

Example Explained

- In this example, The word "w3schools" is being searched for in a sentence.
- First, the pattern is created using the `Pattern.compile()` method. The first parameter indicates which pattern is being searched for and the second parameter has a flag to indicates that the search should be case-insensitive. The second parameter is optional.
- The `matcher()` method is used to search for the pattern in a string. It returns a `Matcher` object which contains information about the search that was performed.

- The `find()` method returns true if the pattern was found in the string and false if it was not found.

Flags

Flags in the `compile()` method change how the search is performed. Here are a few of them:

- `Pattern.CASE_INSENSITIVE` - The case of letters will be ignored when performing a search.
- `Pattern.LITERAL` - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
- `Pattern.UNICODE_CASE` - Use it together with the `CASE_INSENSITIVE` flag to also ignore the case of letters outside of the English alphabet

Regular Expression Patterns

The first parameter of the `Pattern.compile()` method is the pattern. It describes what is being searched for.

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find one character from the options between the brackets
[^abc]	Find one character NOT between the brackets
[0-9]	Find one character from the range 0 to 9

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
	Find a match for any one of the patterns separated by as in: cat dog fish
.	Find just one instance of any character
^	Finds a match as the beginning of a string as in: ^Hello
\$	Finds a match at the end of the string as in: World\$
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one <i>n</i>
n*	Matches any string that contains zero or more occurrences of <i>n</i>
n?	Matches any string that contains zero or one occurrences of <i>n</i>
n{x}	Matches any string that contains a sequence of <i>X n</i> 's
n{x,y}	Matches any string that contains a sequence of <i>X</i> to <i>Y n</i> 's
n{x,}	Matches any string that contains a sequence of at least <i>X n</i> 's

Note: If your expression needs to search for one of the special characters you can use a backslash (\) to escape them. In Java, backslashes in strings need to be escaped themselves, so two backslashes are needed to escape special characters. For example, to search for one or more question marks you can use the following expression: "\\?"

JAVA THREADS

Java Threads

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Creating a Thread

There are two ways to create a thread.

It can be created by extending the **Thread** class and overriding its **run()** method:

Extend Syntax:

```
public class Main extends Thread
{
    public void run() {
        System.out.println("This code is running in a thread");
    }
}
```

Another way to create a thread is to implement the **Runnable** interface:

Implement Syntax:

```
public class Main implements Runnable
{
    public void run() {
        System.out.println("This code is running in a thread");
    }
}
```

Running Threads

If the class extends the **Thread** class, the thread can be run by creating an instance of the class and call its **start()** method:

Extend Example:

```
public class Main extends Thread
{
    public static void main(String[] args)
    {
        Main thread = new Main();
        thread.start();
        System.out.println("This code is outside of the thread");
    }
    public void run() {
        System.out.println("This code is running in a thread");
    }
}
```

If the class implements the **Runnable** interface, the thread can be run by passing an instance of the class to a **Thread** object's constructor and then calling the thread's **start()** method:

Implement Example:

```
public class Main implements Runnable
{
    public static void main(String[] args)
    {
        Main obj = new Main();
        Thread thread = new Thread(obj);
        thread.start();
        System.out.println("This code is outside of the thread");
    }
}
```

```

public void run() {
    System.out.println("This code is running in a thread");
}
}

```

Differences between "extending" and "implementing" Threads

The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like: class **MyClass extends OtherClass implements Runnable**.

Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

Example:

A code example where the value of the variable **amount** is unpredictable:

```

public class Main extends Thread {
    public static int amount = 0;

    public static void main(String[] args)
    {Main thread = new Main();
    thread.start();
    System.out.println(amount);
    amount++;
    System.out.println(amount);
}

```

```

public void run()
{amount++;
}
}

```

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the **isAlive()** method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

Example:

Use `isAlive()` to prevent concurrency problems:

```
public class Main extends Thread {
    public static int amount = 0;

    public static void main(String[] args)
    {Main thread = new Main();
    thread.start();
    // Wait for the thread to finish
    while(thread.isAlive())
    { System.out.println("Waiting...")
    ;
    }
    // Update amount and print its value
    System.out.println("Main: " + amount);
    amount++;
    System.out.println("Main: " + amount);
    }
    public void run()
    {amount++;
    }
}
```

JAVA LAMBDA EXPRESSIONS

Java Lambda Expressions

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Syntax

The simplest lambda expression contains a single parameter and an expression:

parameter -> expression

To use more than one parameter, wrap them in parentheses:

(parameter1, parameter2) -> expression

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as **if** or **for**. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a **return** statement.

```
(parameter1, parameter2) -> { code block }
```

Using Lambda Expressions

Lambda expressions are usually passed as parameters to a function:

Example:

Use a lambda expression in the **ArrayList**'s **forEach()** method to print every item in the list:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.add(5);
        numbers.add(9);
        numbers.add(8);
        numbers.add(1);
        numbers.forEach( (n) -> { System.out.println(n); } );
    }
}
```

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the **Consumer** interface (found in the **java.util** package) used by lists.

Example:

Use Java's **Consumer** interface to store a lambda expression in a variable:

```
import java.util.ArrayList;
import java.util.function.Consumer;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.add(5);
    }
}
```

```

numbers.add(9);
numbers.add(8);
numbers.add(1);
Consumer<Integer> method = (n) -> { System.out.println(n); };
numbers.forEach( method );
}
}

```

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

Example:

Create a method which takes a lambda expression as a parameter:

```

interface StringFunction {
    String run(String str);
}

public class Main {
    public static void main(String[] args)
    { StringFunction exclaim = (s) -> s +
      "!";StringFunction ask = (s) -> s + "?";
      printFormatted("Hello", exclaim);
      printFormatted("Hello", ask);
    }
    public static void printFormatted(String str, StringFunction format)
    {String result = format.run(str);
      System.out.println(result);
    }
}

```

JAVA FILES

File handling is an important part of any application.

Java has several methods for creating, reading, updating, and deleting files.

Java File Handling

The **File** class from the **java.io** package, allows us to work with files.

To use the **File** class, create an object of the class, and specify the filename or directory name:

Example:

```
import java.io.File; // Import the File class
```

```
File myObj = new File("filename.txt"); // Specify the filename
```

If you don't know what a package is, read our [Java Packages Tutorial](#).

The **File** class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
<code>canRead()</code>	Boolean	Tests whether the file is readable or not
<code>canWrite()</code>	Boolean	Tests whether the file is writable or not
<code>createNewFile()</code>	Boolean	Creates an empty file
<code>delete()</code>	Boolean	Deletes a file
<code>exists()</code>	Boolean	Tests whether the file exists
<code>getName()</code>	String	Returns the name of the file
<code>getAbsolutePath()</code>	String	Returns the absolute pathname of the file
<code>length()</code>	Long	Returns the size of the file in bytes
<code>list()</code>	String[]	Returns an array of the files in the directory
<code>mkdir()</code>	Boolean	Creates a directory

You will learn how to create, write, read and delete files in the next chapters:

JAVA CREATE AND WRITE TO FILES

Create a File

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: **true** if the file was successfully created, and **false** if the file already exists. Note that the method is enclosed in a `try...catch` block. This is necessary because it throws an **IOException** if an error occurs (if the file cannot be created for some reason):

Example:

```
import java.io.File; // Import the File class
```

```
import java.io.IOException; // Import the IOException class to handle errors
```



```

public class CreateFile {
    public static void main(String[] args)
    {try {
        File myObj = new File("filename.txt");
        if (myObj.createNewFile()) {
            System.out.println("File created: " + myObj.getName());
        } else {
            System.out.println("File already exists.");
        }
    } catch (IOException e)
    { System.out.println("An error
    occurred.");e.printStackTrace();
    }
    }
}

```

The output will be:

```
File created: filename.txt
```

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

Example:

```

import java.io.File; // Import the File class
import java.io.IOException; // Import the IOException class to handle errors

```

```

public class CreateFile {
    public static void main(String[] args)
    {try {
        File myObj = new File("C:\\Users\\MyName\\filename.txt");
        if (myObj.createNewFile()) {
            System.out.println("File created: " + myObj.getName());
        } else {
            System.out.println("File already exists.");
        }
    } catch (IOException e) {

```

```
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
}
```

Write To a File

In the following example, we use the `FileWriter` class together with its `write()` method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the `close()` method:

Example:

```
import java.io.FileWriter; // Import the FileWriter class
import java.io.IOException; // Import the IOException class to handle errors
```

```
public class WriteToFile {
    public static void main(String[] args)
    {try {
        FileWriter myWriter = new FileWriter("filename.txt");
        myWriter.write("Files in Java might be tricky, but it is fun enough!");
        myWriter.close();
        System.out.println("Successfully wrote to the file.");
    } catch (IOException e)
    { System.out.println("An error
    occurred.");e.printStackTrace();
    }
}
}
```

The output will be:

```
Successfully wrote to the file.
```

JAVA READ FILES

Read a File

In the previous chapter, you learned how to create and write to a file.

In the following example, we use the **Scanner** class to read the contents of the text file we created in the previous chapter:

Example:

```
import java.io.File; // Import the File class
import java.io.FileNotFoundException; // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files
```

```
public class ReadFile {
    public static void main(String[] args)
    {try {
        File myObj = new File("filename.txt");
        Scanner myReader = new Scanner(myObj);
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            System.out.println(data);
        }
        myReader.close();
    } catch (FileNotFoundException e)
    { System.out.println("An error
    occurred.");e.printStackTrace();
    }
    }
}
```

The output will be:

```
Files in Java might be tricky, but it is fun enough!
```

Get File Information

To get more information about a file, use any of the **File** methods:

Example:

```
import java.io.File; // Import the File class
```

```
public class GetFileInfo {
    public static void main(String[] args)
    { File myObj = new
    File("filename.txt");if (myObj.exists())
    {
        System.out.println("File name: " + myObj.getName());
    }
}
```

```

System.out.println("Absolute path: " + myObj.getAbsolutePath());
System.out.println("Writable: " + myObj.canWrite());
System.out.println("Readable " + myObj.canRead());
System.out.println("File size in bytes " + myObj.length());
} else {
    System.out.println("The file does not exist.");
}
}
}
}

```

The output will be:

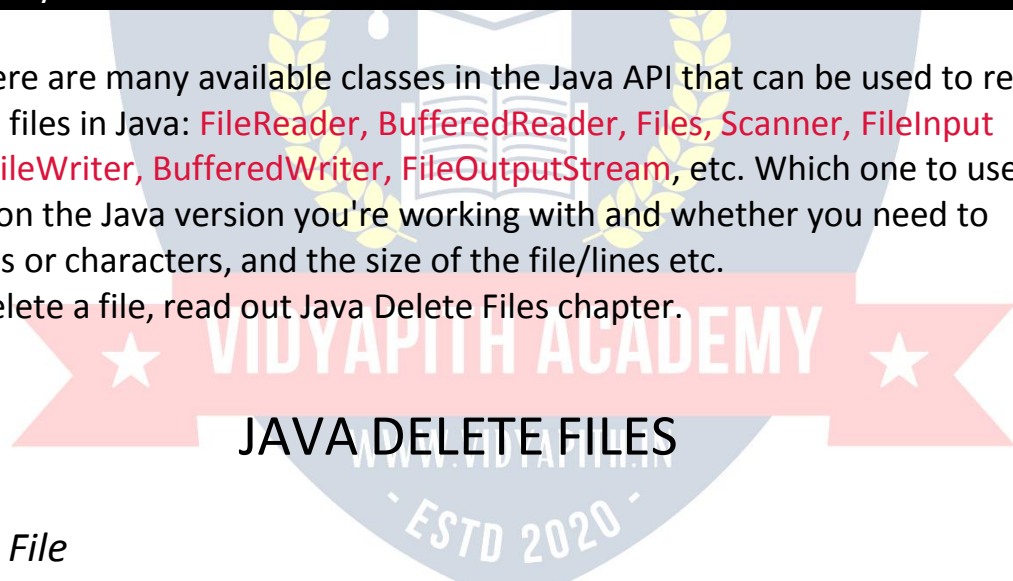
```

File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writable: true
Readable: true
File size in bytes: 0

```

Note: There are many available classes in the Java API that can be used to read and write files in Java: **FileReader, BufferedReader, Files, Scanner, FileInputStream, FileWriter, BufferedWriter, FileOutputStream**, etc. Which one to use depends on the Java version you're working with and whether you need to read bytes or characters, and the size of the file/lines etc.

Tip: To delete a file, read out Java Delete Files chapter.



JAVA DELETE FILES

Delete a File

To delete a file in Java, use the **delete()** method:

Example:

```
import java.io.File; // Import the File class
```

```

public class DeleteFile {
    public static void main(String[] args)
    { File myObj = new
      File("filename.txt");if (myObj.delete())
    {

```

```
    System.out.println("Deleted the file: " + myObj.getName());
} else {
    System.out.println("Failed to delete the file.");
}
}
}
```

The output will be:

```
Deleted the file: filename.txt
```

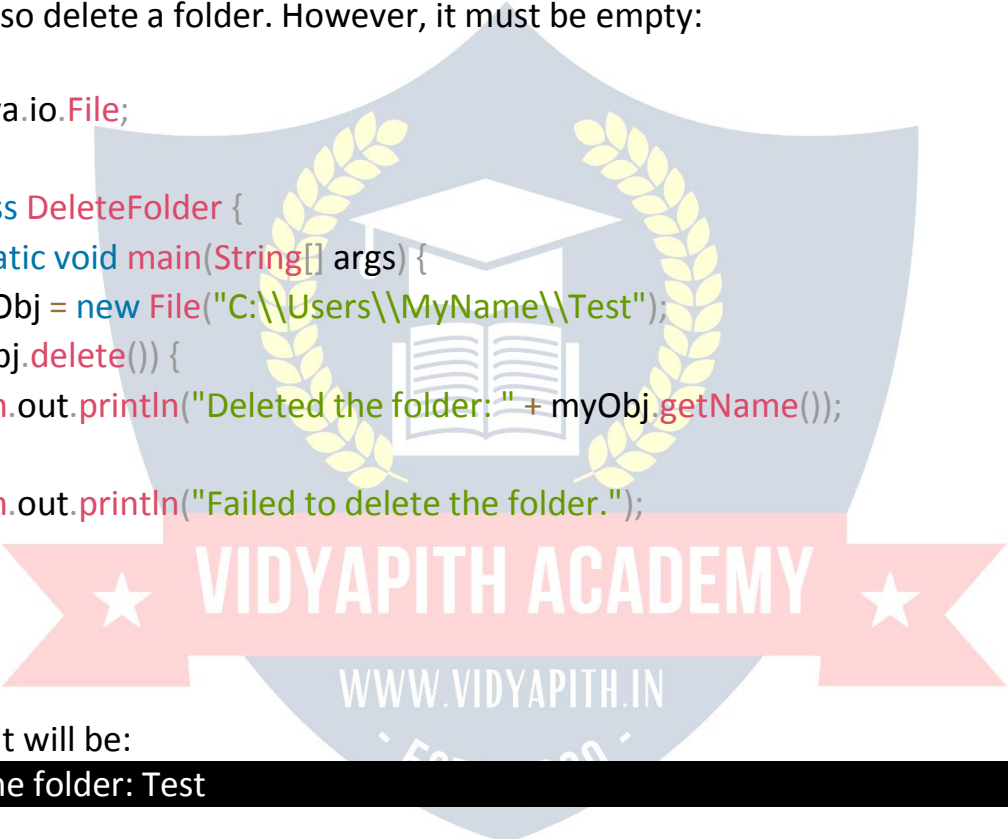
Delete a Folder

You can also delete a folder. However, it must be empty:

Example:

```
import java.io.File;

public class DeleteFolder {
    public static void main(String[] args) {
        File myObj = new File("C:\\Users\\MyName\\Test");
        if (myObj.delete()) {
            System.out.println("Deleted the folder: " + myObj.getName());
        } else {
            System.out.println("Failed to delete the folder.");
        }
    }
}
```



The output will be:

```
Deleted the folder: Test
```

JAVA HOW TO ADD TWO NUMBERS

Add Two Numbers

Learn how to add two numbers in Java:

Example:

```
public class Main {
    public static void main(String[] args) {
```

```
int x = 5;
int y = 6;
int sum = x + y;
System.out.println(sum); // Print the sum of x + y
}
}
```

Add Two Numbers with User Input

Learn how to add two numbers with user input:

Example:

```
import java.util.Scanner; // Import the Scanner class
```

```
class MyClass {
public static void main(String[] args)
{int x, y, sum;
Scanner myObj = new Scanner(System.in); // Create a Scanner object
System.out.println("Type a number:");
x = myObj.nextInt(); // Read user input

System.out.println("Type another number:");
y = myObj.nextInt(); // Read user input

sum = x + y; // Calculate the sum of x + y
System.out.println("Sum is: " + sum); // Print the sum
}
```

VIDYAPITH ACADEMY

A unit of **AITDC (OPC) PVT. LTD.**

IAF Accredited An ISO 9001:2015 Certified Institute.

Registered Under Ministry of Corporate Affairs

(CIN U80904AS2020OPC020468)

Registered Under MSME, Govt. of India. (UAN- AS04D0000207).

Registered Under MHRD (CR act) Govt. of India.

