

JAVA NOTES

❖ 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 languages in the world
- It has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has 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.

❖ 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.

Program :-

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

Output :-

Hello World

❖ Java Multi-line Comments

Multi-line comments start with `/*` and ends with `*/`.

Any text between `/*` and `*/` will be ignored by Java.

Program:-

```
public class Main {  
    public static void main (String[] args) {  
        /* The code below will print the words Hello World  
        to the screen, and it is amazing */  
        System.out.println("Hello World");  
    }  
}
```

Output:- Hello World

❖ 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.

❖ Identifiers

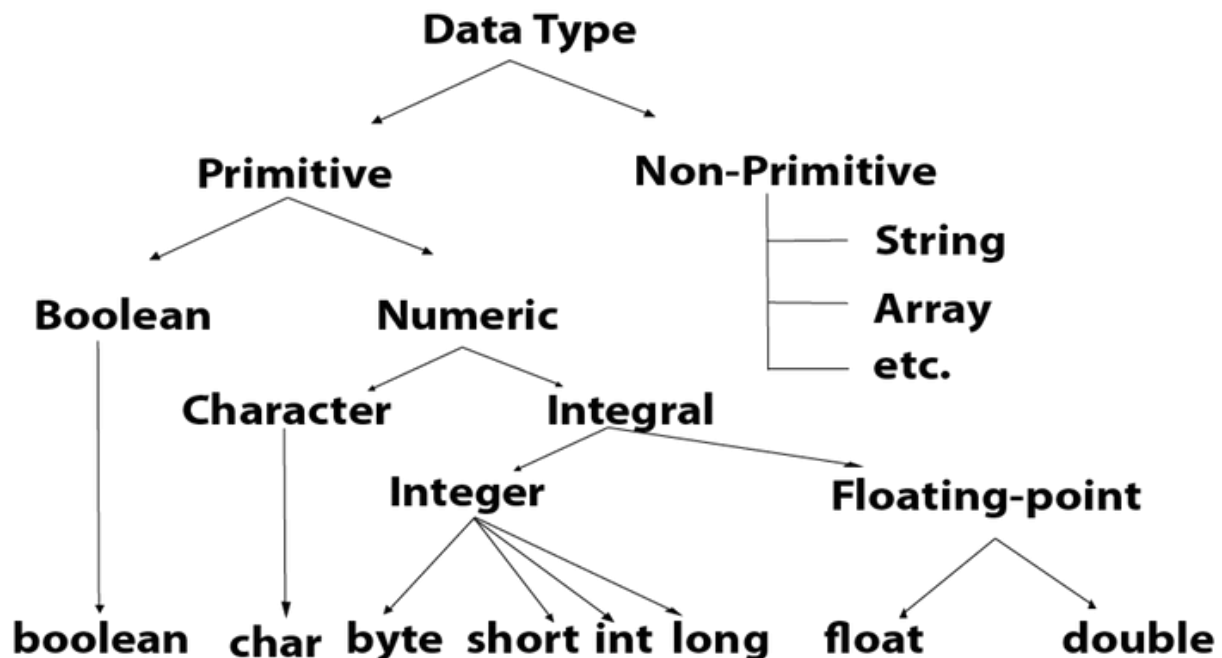
All Java **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, total Volume).

❖ Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:



Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include [Classes](#), [Interfaces](#), and [Arrays](#).

❖ 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.

❖ 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:

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

O/p:-

150
400
800

There are many types of operators in Java which are given below:

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

❖ Arithmetic Operators

These operators involve the mathematical operators that can be used to perform various simple or advanced arithmetic operations on the primitive data types referred to as the operands. These operators consist of various unary and binary operators that can be applied on a single or two operands. Let's look at the various operators that Java has to provide under the arithmetic operators.

Operators	Result
+	Addition of two numbers
-	Subtraction of two numbers
*	Multiplication of two numbers
/	Division of two numbers
%	(Modulus Operator) Divides two numbers and returns the remainder

❖ 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**:

```
public class Main {

    public static void main(String[] args) {

        int x = 10;

        System.out.println(x);

    }

}
```

```
}
```

o/p:-

10

❖ Java Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either true or false. These values are known as *Boolean values*, and you will learn more about them in the [Booleans](#) and [If..Else](#) chapter.

In the following example, we use the **greater than** operator (>) to find out if 5 is greater than 3:

```
public class Main {  
    public static void main(String[] args) {  
        int x = 5;  
        int y = 3;  
        System.out.println(x > y); // returns true, because 5 is higher than 3  
    }  
}
```

o/p :-

true

❖ Java Logical Operators

You can also test for true or false values with logical operators.

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

- **AND Operator (&&)** – if(a && b) [if true execute else don't]
- **OR Operator (||)** – if(a || b) [if one of them is true to execute else don't]
- **NOT Operator (!)** – !(a<b) [returns false if a is smaller than b]

❖ Bitwise Operators

Bitwise operators are used to performing the manipulation of individual bits of a number. They can be used with any integral type (char, short, int, etc.). They are used when performing update and query operations of the Binary indexed trees.

Now let's look at each one of the bitwise operators in Java:

1. Bitwise OR (|)

This operator is a binary operator, denoted by '|'. It returns bit by bit OR of input values, i.e., if either of the bits is 1, it gives 1, else it shows 0.

Example:

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7

```
0101
| 0111
```

0111 = 7 (In decimal)

2. Bitwise AND (&)

This operator is a binary operator, denoted by '&.' It returns bit by bit AND of input values, i.e., if both bits are 1, it gives 1, else it shows 0.

Example:

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise AND Operation of 5 and 7

0101
& 0111

0101 = 5 (In decimal)

3. Bitwise XOR (^)

This operator is a binary operator, denoted by '^.' It returns bit by bit XOR of input values, i.e., if corresponding bits are different, it gives 1, else it shows 0.

Example:

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7

0101
^ 0111

0010 = 2 (In decimal)

4. Bitwise Complement (~)

This operator is a unary operator, denoted by '~.' It returns the one's complement representation of the input value, i.e., with all bits inverted, which means it makes every 0 to 1, and every 1 to 0.

Example:

$a = 5 = 0101$ (In Binary)

Bitwise Complement Operation of 5

~ 0101

$1010 = 10$ (In decimal)

❖ Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](#) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements

- if statements
- switch statement

2. Loop statements

- do while loop
- while loop
- for loop
- for-each loop

3. Jump statements

- break statement
- continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements

evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

Backward Skip 10s Play Video Forward Skip 10s

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

1. `if(condition) {`
2. `statement 1; //executes when condition is true`
3. `}`

Consider the following example in which we have used the if statement in the java code.

Student.java

Student.java

```
public class Student {  
    public static void main(String[] args) {  
        int x = 10;  
        int y = 12;  
        if(x+y > 20) {  
            System.out.println("x + y is greater than 20");  
        }  
    }  
}
```

```
}  
}  
}
```

Output:

x + y is greater than 20

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {  
    statement 1; //executes when condition is true  
}  
else {  
    statement 2; //executes when condition is false  
}
```

Consider the following example.

Student.java

```
public class Student {  
    public static void main(String[] args) {  
        int x = 10;  
        int y = 12;  
        if(x+y < 10) {  
            System.out.println("x + y is less than 10");  
        } else {  
            System.out.println("x + y is greater than 20");  
        }  
    }  
}
```

Output:

x + y is greater than 20

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

ADVERTISEMENT

Syntax of if-else-if statement is given below.

```
if(condition 1) {  
    statement 1; //executes when condition 1 is true  
}  
else if(condition 2) {  
    statement 2; //executes when condition 2 is true  
}  
else {  
    statement 2; //executes when all the conditions are false  
}
```

Consider the following example.

Student.java

```
public class Student {  
    public static void main(String[] args) {  
        String city = "Delhi";  
        if(city == "Meerut") {  
            System.out.println("city is meerut");  
        }else if (city == "Noida") {  
            System.out.println("city is noida");  
        }  
    }  
}
```

```
    }else if(city == "Agra") {  
        System.out.println("city is agra");  
    }else {  
        System.out.println(city);  
    }  
}  
}
```

Output:

Delhi

4. Nested if-statement

In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {  
    statement 1; //executes when condition 1 is true  
    if(condition 2) {  
        statement 2; //executes when condition 2 is true  
    }  
    else{  
        statement 2; //executes when condition 2 is false  
    }  
}
```

Consider the following example.

Student.java

```
public class Student {  
    public static void main(String[] args) {  
        String address = "Delhi, India";
```

```
if(address.endsWith("India")) {  
    if(address.contains("Meerut")) {  
        System.out.println("Your city is Meerut");  
    }else if(address.contains("Noida")) {  
        System.out.println("Your city is Noida");  
    }else {  
        System.out.println(address.split(",")[0]);  
    }  
}else {  
    System.out.println("You are not living in India");  
}  
}  
}
```

Output:

Delhi

5. Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

ADVERTISEMENT

Points to be noted about switch statement:

ADVERTISEMENT

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.

- Break statement terminates the switch block when the condition is satisfied.
It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){  
    case value1:  
        statement1;  
    break;  
    case valueN:  
        statementN;  
    break;  
    default:  
        default statement;  
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {  
    public static void main(String[] args) {  
        int num = 2;  
        switch (num){  
            case 0:  
                System.out.println("number is 0");  
                break;  
            case 1:  
                System.out.println("number is 1");  
                break;  
            default:
```



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

Output:

2

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

❖ Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

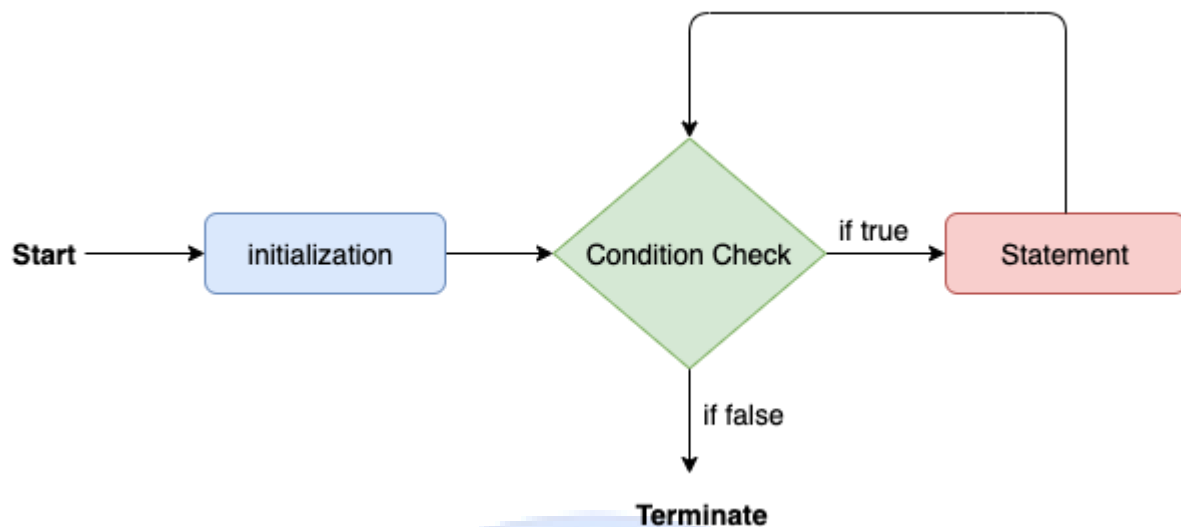
Let's understand the loop statements one by one.

❖ Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

1. for(initialization, condition, increment/decrement) {
2. //block of statements
3. }

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int sum = 0;  
        for(int j = 1; j<=10; j++) {  
            sum = sum + j;  
        }  
        System.out.println("The sum of first 10 natural numbers is " + sum);  
    }  
}
```

Output:

The sum of first 10 natural numbers is 55

❖ Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){  
    //statements  
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

```
Calculation.java  
  
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        String[] names = {"Java","C","C++","Python","JavaScript"};  
        System.out.println("Printing the content of the array names:\n");  
        for(String name:names) {  
            System.out.println(name);  
        }  
    }  
}
```

Output:

Printing the content of the array names:

Java

C

C++

Python

JavaScript

❖ Java while loop

The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for

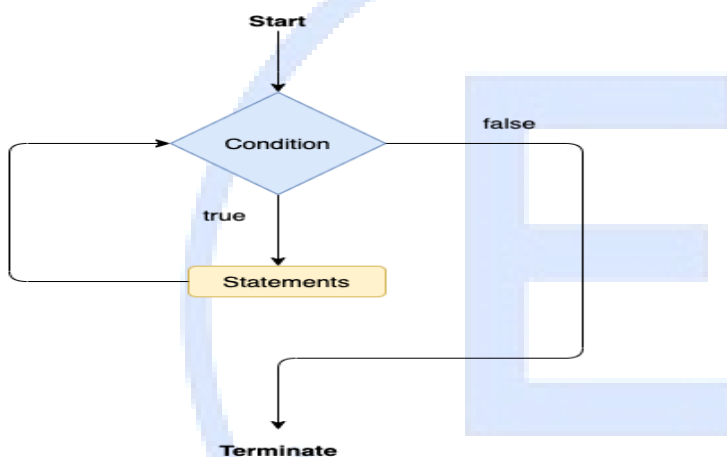
loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
while(condition){  
    //looping statements  
}
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int i = 0;  
        System.out.println("Printing the list of first 10 even numbers \n");  
  
        while(i<=10) {  
            System.out.println(i);  
            i = i + 2;  
        }  
    }  
}
```

```
}  
}
```

Output:

Printing the list of first 10 even numbers

```
0  
2  
4  
6  
8  
10
```

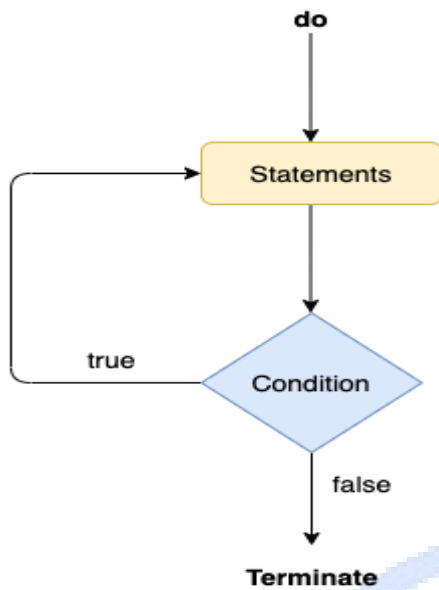
❖ Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
do  
{  
//statements  
} while (condition);
```

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int i = 0;  
        System.out.println("Printing the list of first 10 even numbers \n");  
        do {  
            System.out.println(i);  
            i = i + 2;  
        } while(i <= 10);  
    }  
}
```

Output:

Printing the list of first 10 even numbers

0

2

4

6

8

10

❖ Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```
public class BreakExample {  
    public static void main(String[] args) {  
        //TODO Auto-generated method stub  
        for(int i = 0; i<= 10; i++) {  
            System.out.println(i);  
            if(i==6) {  
                break;  
            }  
        }  
    }  
}
```

Output:

0

1
2
3
4
5
6

break statement example with labeled for loop

Calculation.java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        a:  
        for(int i = 0; i<= 10; i++) {  
            b:  
            for(int j = 0; j<=15;j++) {  
                c:  
                for (int k = 0; k<=20; k++) {  
                    System.out.println(k);  
                    if(k==5) {  
                        break a;  
                    }  
                }  
            }  
        }  
    }  
}
```


Output:

0
1
2
3
4
5

❖ Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class Continue Example {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        for(int i = 0; i<= 2; i++) {  
            for (int j = i; j<=5; j++) {  
                if(j == 4) {  
                    continue;  
                }  
                System.out.println(j);  
            }  
        }  
    }  
}
```

Output:

0

1
2
3
5
1
2
3
5
2
3
5

❖ 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, you can 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 can 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]);
}
}
```

Output :-

Volvo

❖ 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

```
public class Main {
    int x;

    // Create a class constructor for the Main class
    public Main() {
        x = 5;
    }

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

Output:-

5

❖ 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

❖ 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. However, it is possible to access them if we provide public **get** and **set** methods.

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 & 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: -

```
import java.util.Scanner; // import the Scanner class
class Main {
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);
        String userName;
        // Enter username and press Enter
        System.out.println("Enter username");
        userName = myObj.nextLine();
        System.out.println("Username is: " + userName);
    }
}
```

Output:-

Enter 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!");
    }
}
```

Output:-

This is my package!

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):

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

```
class Car extends Vehicle {  
    private String modelName = "Mustang";  
    public static void main(String[] args) {  
        Car myFastCar = new Car();  
        myFastCar.honk();  
        System.out.println(myFastCar.brand + " " + myFastCar.modelName);  
    }  
}
```

Tuut, tuut!

Ford Mustang

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.):

```
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();
        Animal myPig = new Pig();
        Animal myDog = new Dog();

        myAnimal.animalSound();
        myPig.animalSound();
        myDog.animalSound();
    }
}
```

Output:-

The animal makes a sound
The pig says: wee wee
The dog says: bow wow

❖ 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

```
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();  
    }  
}
```

Output:-

The pig says: wee wee
Zzz

❖ 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();
    }
}
```

Output: -The pig says: wee wee
Zzz

❖ 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.

Java ArrayList:-

The Array List 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:-

```
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);  
}  
}
```

Output:-

[Volvo, BMW, Ford, Mazda]

Access an Item:-

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

Example

```
cars.get(0);
```

Change an Item:-

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

Example

```
cars.set(0, "Opel");
```

Remove an Item:-

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

Example

```
cars.remove(0);
```

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

Example

```
cars.clear();
```

ArrayList Size:-

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

Example

```
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));  
        }  
    }  
}
```

❖ 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));  
        }  
    }  
}
```

Output:-

Volvo
BMW
Ford
Mazda

❖ 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);
        }
    }
}
```

Output:-

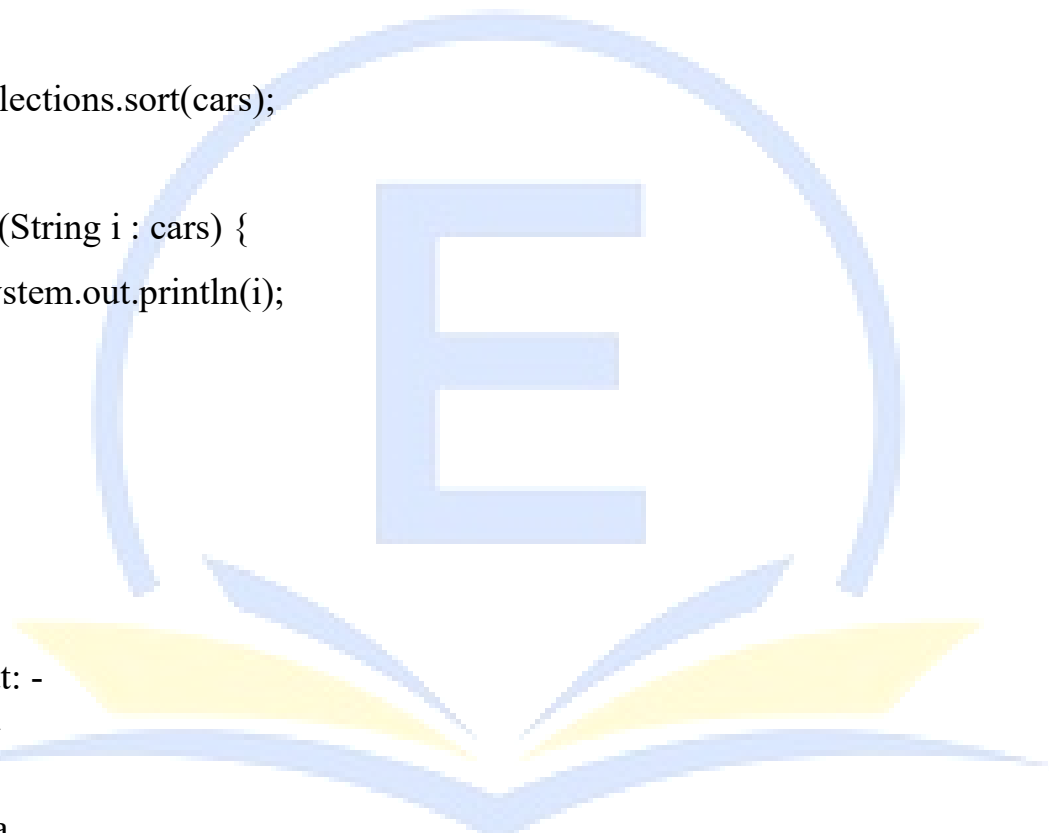
Sort an ArrayList

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

Example

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

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



Output: -

BMW
Ford
Mazda
Volvo

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.

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());  
}  
}
```

Output:- Volvo

❖ 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  
}
```

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.");  
        }  
    }  
}
```

❖ 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");  
    }  
}
```

Output:-

```
This code is outside of the thread  
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");  
}  
}
```

Output:-

This code is outside of the thread
This code is running in a thread

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
```

Java Create and Write To Files

[◀ PreviousNext ▶](#)

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