1. **How to create a class, object, method and its signature**

**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 MyClass, so now we can use this to create objects.

To create an object of MyClass, 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);

}

}

**Java Methods**

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.

void means that this method does not have a return value.

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 Signatures**

About:

A signature is a list that specifies a class constructor, an instance method, or a static method, thereby distinguishing it from other constructors, instance methods, or static methods.

Two forms of signatures are accepted: simple and full. A simple signature is a single element list containing the name of the method or constructor. In most cases a simple signature is all that is needed as the Java method resolver is able to disambiguate overloaded Java methods based on the types of Java object arguments. There are some cases where the Java method resolver is unable to determine which Java method you intended to invoke so you will need to use the full signature for the method or constructor. The full signature is used to distinguish between two or more methods or constructors that have the same number of arguments. The full signature of a method is a Tcl list containing the method name followed by the name of the Java object type for each parameter of the method.

**3. Write a program for a Single line comment, multi-line and documentation comments**

**Java Comments**

The Java comments are the statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code.

**Types of Java Comments**

There are three types of comments in Kava.

Single Line Comment

Multi Line Comment

Documentation Comment

**1) Java Single Line Comment**

The single line comment is used to comment only one line.

Syntax:

//This is single line comment

**2) Java Multi Line Comment**

The multi line comment is used to comment multiple lines of code.

Syntax:

/\* This is multi line comment \*/

**3) Java Documentation Comment**

The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

Syntax:

/\*\* This is documentation comment \*/

This kind of Java comments is utilized by large code for a programming bundle since it produces a documentation page for reference, which can be utilized for getting data about strategies, its parameters, and so forth.

This type of comments are used generally when writing code for a project/software package, since it helps to generate a documentation page for reference, which can be used for getting information about methods present, its parameters, etc.

Syntax

/\*\*Comment start

\*

\*tags are used in order to specify a parameter

\*or method or heading

\*HTML tags can also be used

\*such as <h1>

\*

\*comment ends\*/

**4. Define variables for different Data Types int, Boolean, char, float, double and print on the Console?**

**Data Types in Java**

**int**

The int data type is used to store integers. Integers are numbers which don’t have decimal. For example, -5, 0, 6, etc.

class Test {

public static void main(String[] args) {

int num;

num = 10;

System.out.println(num);

}

}

Output

10

The variable num is declared of type int and is assigned an integer value 10.

**double**

The double data type is used to store double-precision 64-bit floating point numbers. Floating point numbers are numbers which have decimal. In other words, the double data type is used to store numbers having decimal. For example, -5.64, 10.228, etc.

class Test {

public static void main(String[] args) {

double num;

num = 10.5;

System.out.println(num);

}

}

Output

10.5

The variable num is declared of type double and is assigned a floating point value 10.5.

Note that 8 is an int but 8.0 is a double.

**float**

The float data type is used to store single-precision 32-bit floating point numbers. A float value should always end with f or F. For example, -5.64f, 10.228F, etc.

class Test {

public static void main(String[] args) {

float num;

num = 10.5f;

System.out.println(num);

}

}

Output

10.5

In this example, we assigned the value 10.5f and not 10.5 to the variable num because 10.5 is a double. To tell the compiler to consider 10.5 as a float, we assigned 10.5f to the variable.

**char**

The char data type is used to store a character. A character value must be written within single quotes ' '. For example, ‘a’, ‘B’, ‘@’, etc.

class Test {

public static void main(String[] args) {

char ch;

ch = 'e';

System.out.println(ch);

}

}

Output

e

Here, ch is the name of a variable of type char which is assigned a character value 'e'.

**String**

The String data type is used to store a string. A string is a sequence of characters. For example, “Hello” is a string having characters ‘H’, ‘e’, ‘l’, ‘l’ and ‘o’.

A string value must be enclosed within double quotes " ". In fact, any value enclosed within double quotes " " is a string. Some examples of string are “Hello World”, “Hello123”, "123" and “Name: John”.

class Test {

public static void main(String[] args) {

String msg;

msg = "Let's learn Java";

System.out.println(msg);

}

}

Output

Let's learn Java

In this example, the variable msg is declared of type String and is assigned the string "Let's learn Java".

As we know that any value enclosed within double quotes “ “ is a string, so values like “10” and “10+2” are also strings.

Note that 10 is an int but “10” is a String.

Look at the following example.

class Test {

public static void main(String[] args) {

int a = 10 + 2;

String b = "10 + 2";

System.out.println(a);

System.out.println(b);

}

}

Output

12

10 + 2

In the above example, the integer variable a is assigned the expression 10 + 2. This expression first gets evaluated to 12 and then 12 is assigned to a. On the other hand, the string variable b is assigned the string "10 + 2".

We will learn more about String later in the topic Strings.

**boolean**

The boolean data type consists of two values - true and false.

class Test {

public static void main(String[] args) {

boolean a, b;

a = true;

b = false;

System.out.println(a);

System.out.println(b);

}

}

Output

true

false

The variables a and b are declared of type boolean and are assigned the values true and false respectively.

So these were all the basic data types. Now, let’s look at the range of values that different data types can take.

The following table states different data types along with the maximum and minimum value they can take.

Data Type Maximum Value Minimum Value

int 2,147,483,647

- 2,147,483,648

float 3.4028235E38

1.4E-45

double 1.7976931348623157E308

4.9E-324

char 65,535

0

short 32767

-32767

long 9223372036854775807

-9223372036854775808

10E5 means 105 i.e. 100000.

Let's see an example of double, char and boolean values.

class Test {

public static void main(String[] args) {

double b = 123.43555;

char c = 'e';

boolean d = true;

System.out.println("Double: " + b);

System.out.println("Character: " + c);

System.out.println("Boolean: " + d);

}

}

Output

Double: 123.43555

Character: e

Boolean: true

You must have understood the code. While printing, + joined a string and the value of a variable in each System.out.println() method.

In the first System.out.println() method, the string "Double: " and the value of b (because b is not inside " ") are combined and printed.

Look at another example.

class Test {

public static void main(String[] args) {

int x = 1, y = 5;

System.out.println("x");

System.out.println("y");

System.out.println("x" + "y");

System.out.println(x);

System.out.println(y);

System.out.println(x + y);

}

}

Output

x

y

xy

1

5

6

Here, whatever is written within " " got printed as it is, without getting evaluated. Whatever is not inside " " got evaluated first and then their values got printed. For example, "x"+"y" got printed as xy (without evaluation) but x+y got evaluated first as 1+5 i.e., 6 and then 6 got printed.

**Primitive and Non-Primitive Data types**

Using data types is so easy, isn't it?

All the data types are broadly classified into primitive and non-primitive.

**Primitive Data type**

Primitive data types are predefined (already defined) data types in Java.

There are eight predefined data types in Java, which are

int, float, double, short, long, char, boolean, byte

**Non-Primitive Data type**

Non-primitive data types are defined by the programmer. Some examples of non-primitive data types are Array, Class and Interface.

At present, there is no need to go into the details of non-primitive data types as we will learn about them later.

**Type Casting**

Suppose we are writing a program and we have an integer variable having a value 10 (an integer) and at some point of time we want it to be a string i.e., “10”. Or a more practical case would be to convert a double (10.2) to an integer (10). We can easily do so in Java using type casting.

Type Casting is the conversion of a value from one data type to another data type. For example, we can convert a double value to an int value or a char value to an int value.

Type Conversions are of two types - implicit and explicit.

**Implicit Conversion**

Suppose we are adding two numbers. The first number is of type int and the second number is of type double. We cannot add an int and a double because both the numbers have to be of the same data type i.e. either both are int or both are double. Since double is a larger data type than int, therefore while adding, the int variable automatically gets converted into double and then both the double variables add up.

Order of size of data types:

double > float > long > int > char > short

From the above order, we can see that double is the largest data type and short is the smallest data type. Any smaller data type gets implicitly converted into a larger data type when performing arithmetic operations or in any such other expression.

For example, when adding a value of type int and a value of type long, the value of type int gets automatically converted to long and then both the values get added.

Similarly, a char variable gets converted into an int while performing some arithmetic operation.

class Test {

public static void main(String[] args) {

int n = 10;

char ch = 'h';

int sum = n + ch;

System.out.println(sum);

}

}

Output

114

In the above program, when the variables n and ch are added, the integer value (ASCII value) of ch i.e. 104 is added to the value of n to produce a sum of 114. Note that every character has an ASCII value. You can get the ASCII chart from here.

**Explicit Conversion**

We know that a smaller data type can be implicitly converted to a larger data type. But what if we want to convert a larger data type to a smaller data type?

We can also convert values from one data type to another as shown below:

( data-type ) expression;

For example, a double value 10.5 can be converted to int as shown below.

(int)10.5;

Consider an example.

class D3 {

public static void main(String[] args) {

int sum = 23;

int n = 7;

double avg;

avg = (double) sum / n;

System.out.println("Average = " + avg);

}

}

Output

Average = 3.2857142857142856

In this example, since the variable avg is declared of type double, we are converting sum/n to type double by writing (double)sum/n (since int/int gives int in Java).

**Console.** Java programs often write to the console window. We use System.out—we read input, and write output, with this stream. And this fills many program requirements.

**Console usage**. We can use println with no argument for a blank line. Different argument types, like arrays, strings, and ints are supported. The methods are versatile.

**5. Define the local and Global variables with the same name and print both variables and understand the scope of the variables?**

**Difference between Local variable and Global Variable**

Variables in any programming language have a crucial role. Variables are classified into Global variables and Local variables based on their scope. The main difference between Global and local variables is that global variables can be accessed globally in the entire program, whereas local variables can be accessed only within the function or block in which they are defined. In this topic, we will first understand what are the variables and scope, along with local variables, global variables, and then differences between both the variables.

**What is a Variable?**

A variable is a name given to a memory location to store values in a computer program. It is used to store information that can be referenced and manipulated in a program.

We can choose any name for the variable, but it must follow the programming semantics. Such as it can be, a, b, x, y, z, sub, div, total, avg, etc.

Let's say there are two values, 10 and 20, that we want to store and use in our program. For this, we need to use a variable, and we will do the below steps:

First, we will create or declare a variable with a suitable name.

Assign those values to the variables to store them.

Once these values are stores, we can use these variables with their name in our program.

Local variable vs Global Variable

As we can see in the above image, there are two memory slots, 001 and 002, and we have given names to these locations as A and B. A is containing 10, and B is containing 20.

Different programming languages have different ways to declare the variable. For example, in C language, we can declare the variable in the following manner:

Syntax: (Variable declaration syntax in C language)

datatype v1, v2, v3,....;

Example:

#include <stdio.h>

void main(){

int a;

int b;

int sum;

}

**Scope of Variable**

Each variable is defined and can be used within its scope and determines that wherein the program this variable is available to use. The scope means the lifetime of that variable. It means the variable can only be accessed or visible within its scope.

The scope of variables can be defined with their declaration, and variables are declared mainly in two ways:

Global Variable: Outside of all the functions

Local Variable: Within a function block:

**What is a Global Variable?**

Global variables are those variables which are declared outside of all the functions or block and can be accessed globally in a program.

It can be accessed by any function present in the program.

Once we declare a global variable, its value can be varied as used with different functions.

The lifetime of the global variable exists till the program executes. These variables are stored in fixed memory locations given by the compiler and do not automatically clean up.

Global variables are mostly used in programming and useful for cases where all the functions need to access the same data.

Example:

#include<stdio.h>

int a=50, b=40;

void main()

{

printf("a = %d and b=%d",a,b);

}

In the above example, a and b are the global variables.

**Advantages of Global Variable**

Global variables can be accessed by all the functions present in the program.

Only a single declaration is required.

Very useful if all the functions are accessing the same data.

**Disadvantages of Global Variable**

The value of a global variable can be changed accidently as it can be used by any function in the program.

If we use a large number of global variables, then there is a high chance of error generation in the program.

**What is a Local Variable?**

Variables that are declared within or inside a function block are known as Local variables.

These variables can only be accessed within the function in which they are declared.

The lifetime of the local variable is within its function only, which means the variable exists till the function executes. Once function execution is completed, local variables are destroyed and no longer exist outside the function.

The reason for the limited scope of local variables is that local variables are stored in the stack, which is dynamic in nature and automatically cleans up the data stored within it.

But by making the variable static with "static" keyword, we can retain the value of local variable.

Example:

#include<stdio.h>

void main()

{

int x=50, y=40;

printf("x = %d and y=%d",x, y);

}

In the above example, we have declared x and y two variables inside the main function. Hence these are local variables.

Advantages of Local Variable

The same name of a local variable can be used in different functions as it is only recognized by the function in which it is declared.

Local variables use memory only for the limited time when the function is executed; after that same memory location can be reused.

Disadvantages of Local Variables

The scope of the local variable is limited to its function only and cannot be used by other functions.

Data sharing by the local variable is not allowed.

Comparison Chart between Global Variable and Local Variable

Global Variable Local Variable

Global variables are declared outside all the function blocks. Local Variables are declared within a function block.

The scope remains throughout the program. The scope is limited and remains within the function only in which they are declared.

Any change in global variable affects the whole program, wherever it is being used. Any change in the local variable does not affect other functions of the program.

A global variable exists in the program for the entire time the program is executed. A local variable is created when the function is executed, and once the execution is finished, the variable is destroyed.

It can be accessed throughout the program by all the functions present in the program. It can only be accessed by the function statements in which it is declared and not by the other functions.

If the global variable is not initialized, it takes zero by default. If the local variable is not initialized, it takes the garbage value by default.

Global variables are stored in the data segment of memory. Local variables are stored in a stack in memory.

We cannot declare many variables with the same name. We can declare various variables with the same name but in other functions.

Examples to understand differences between Local and Global Variable

Now let's understand examples in different programming languages to better understand the difference between local and global variables.

**6. Write a function to print your name and call the function from main method?**

**Java Methods**

A method is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

A method to draw the circle

A method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

**In Java, there are two types of methods:**

**User-defined Methods:** We can create our own method based on our requirements.

**Standard Library Methods:** These are built-in methods in Java that are available to use.

**Let's first learn about user-defined methods.**

**Declaring a Java Method**

**The syntax to declare a method is:**

returnType methodName() {

// method body

}

Here,

**returnType -** It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value.

If the method does not return a value, its return type is void.

**methodName** - It is an identifier that is used to refer to the particular method in a program.

**method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }.

**For example,**

int addNumbers() {

// code

}

In the above example, the name of the method is adddNumbers(). And, the return type is int. We will learn more about return types later in this tutorial.

This is the simple syntax of declaring a method. However, the complete syntax of declaring a method is

modifier static returnType nameOfMethod (parameter1, parameter2, ...) {

// method body

}

Here,

**modifier -** It defines access types whether the method is public, private, and so on. To learn more, visit Java Access Specifier.

**static -** If we use the static keyword, it can be accessed without creating objects.

**For example,** the sqrt() method of standard Math class is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.

parameter1/parameter2 - These are values passed to a method. We can pass any number of arguments to a method.

**Calling a Method in Java**

In the above example, we have declared a method named addNumbers(). Now, to use the method, we need to call it.

Here's is how we can call the addNumbers() method.

// calls the method

addNumbers();

Call a method in Java using the name the method followed by a parenthesis

Working of Java Method Call

Example 1: Java Methods

class Main {

**// create a method**

public int addNumbers(int a, int b) {

int sum = a + b;

// return value

return sum;

}

public static void main(String[] args) {

int num1 = 25;

int num2 = 15;

**// create an object of Main**

Main obj = new Main();

// calling method

int result = obj.addNumbers(num1, num2);

System.out.println("Sum is: " + result);

}

}

**Output**

Sum is: 40

In the above example, we have created a method named addNumbers(). The method takes two parameters a and b. Notice the line,

int result = obj.addNumbers(num1, num2);

Here, we have called the method by passing two arguments num1 and num2. Since the method is returning some value, we have stored the value in the result variable.

Note: The method is not static. Hence, we are calling the method using the object of the class**.**

**Java Method Return Type**

A Java method may or may not return a value to the function call. We use the return statement to return any value. For example,

int addNumbers() {

...

return sum;

}

Here, we are returning the variable sum. Since the return type of the function is int. The sum variable should be of int type. Otherwise, it will generate an error.

**Example 2: Method Return Type**

class Main {

// create a method

public static int square(int num) {

// return statement

return num \* num;

}

public static void main(String[] args) {

int result;

// call the method

// store returned value to result

result = square(10);

System.out.println("Squared value of 10 is: " + result);

}

}

Output:

Squared value of 10 is: 100

In the above program, we have created a method named square(). The method takes a number as its parameter and returns the square of the number.

Here, we have mentioned the return type of the method as int. Hence, the method should always return an integer value.

Java method returns a value to the method call

Representation of the Java method returning a value

Note: If the method does not return any value, we use the void keyword as the return type of the method. For example,

public void square(int a) {

int square = a \* a;

System.out.println("Square is: " + a);

}

**Method Parameters in Java**

A method parameter is a value accepted by the method. As mentioned earlier, a method can also have any number of parameters. For example,

// method with two parameters

int addNumbers(int a, int b) {

// code

}

// method with no parameter

int addNumbers(){

// code

}

If a method is created with parameters, we need to pass the corresponding values while calling the method. For example,

// calling the method with two parameters

addNumbers(25, 15);

// calling the method with no parameters

addNumbers()

**Example 3: Method Parameters**

class Main {

// method with no parameter

public void display1() {

System.out.println("Method without parameter");

}

// method with single parameter

public void display2(int a) {

System.out.println("Method with a single parameter: " + a);

}

public static void main(String[] args) {

// create an object of Main

Main obj = new Main();

// calling method with no parameter

obj.display1();

// calling method with the single parameter

obj.display2(24);

}

}

**Output**

Method without parameter

Method with a single parameter: 24

Here, the parameter of the method is int. Hence, if we pass any other data type instead of int, the compiler will throw an error. It is because Java is a strongly typed language.

Note: The argument 24 passed to the display2() method during the method call is called the actual argument.

The parameter num accepted by the method definition is known as a formal argument. We need to specify the type of formal arguments. And, the type of actual arguments and formal arguments should always match.

**Standard Library Methods**

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (\*.jar) file with JVM and JRE.

For example,

print() is a method of java.io.PrintSteam. The print("...") method prints the string inside quotation marks.

sqrt() is a method of Math class. It returns the square root of a number.

Here's a working example:

**Example 4: Java Standard Library Method**

public class Main {

public static void main(String[] args) {

// using the sqrt() method

System.out.print("Square root of 4 is: " + Math.sqrt(4));

}

}

Output:

Square root of 4 is: 2.0

To learn more about standard library methods, visit Java Library Methods.

**What are the advantages of using methods?**

1. The main advantage is code reusability. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".

Example 5: Java Method for Code Reusability

public class Main {

// method defined

private static int getSquare(int x){

return x \* x;

}

public static void main(String[] args) {

for (int i = 1; i <= 5; i++) {

// method call

int result = getSquare(i);

System.out.println("Square of " + i + " is: " + result);

}

}

}

Output:

Square of 1 is: 1

Square of 2 is: 4

Square of 3 is: 9

Square of 4 is: 16

Square of 5 is: 25

In the above program, we have created the method named getSquare() to calculate the square of a number. Here, the method is used to calculate the square of numbers less than 6.

Hence, the same method is used again and again.

2. Methods make code more readable and easier to debug. Here, the getSquare() method keeps the code to compute the square in a block. Hence, makes it more readable.

1. **Operators**
2. **Write a function for arithmetic operators(+,-,\*,/)**

Java too provides many types of operators which can be used according to the need to perform various calculation and functions be it logical, arithmetic, relational etc. They are classified based on the functionality they provide. Here are a few types:

Arithmetic Operators

Unary Operators

Assignment Operator

Relational Operators

Logical Operators

Ternary Operator

Bitwise Operators

Shift Operators

**Arithmetic Operators**

These operators involve the mathematical operators that can be used to perform various simple or advance 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 respectively. Let’s look at the various operators that Java has to provide under the arithmetic operators.

Addition(+): This operator is a binary operator and is used to add two operands.

Syntax:

num1 + num2

Subtraction(-): This operator is a binary operator and is used to subtract two operands.

Syntax:

num1 - num2

Multiplication(\*): This operator is a binary operator and is used to multiply two operands.

Syntax:

num1 \* num2

Division(/): This is a binary operator that is used to divide the first operand(dividend) by the second operand(divisor) and give the quotient as result.

Syntax:

num1 / num2

Modulus(%): This is a binary operator that is used to return the remainder when the first operand(dividend) is divided by the second operand(divisor).

Syntax:

num1 % num2

Increment(++): This is a unary operator that acts on one operand, unlike the previous operations. It is used to increment the value of an integer. It can be used in two ways:

Post-increment operator: When placed after the variable name, the value of the operand is incremented but the previous value is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

Syntax:

num++

Pre-increment operator: When placed before the variable name, the operand’s value is incremented instantly.

Syntax:

++num

Decrement(–): This is also a unary operator that acts on one operand. It is used to decrement the value of an integer. It can be used in two ways:

Post-decrement operator: When placed after the variable name, the value of the operand is decremented but the previous values is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

Syntax:

num--

Pre-decrement operator: When placed before the variable name, the operand’s value is decremented instantly.

Syntax:

--num

**Unary Operators**

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by 1. That is, ++5 will return 6.

Different types of unary operators are:

Operator Meaning

+ Unary plus: not necessary to use since numbers are positive without using it

- Unary minus: inverts the sign of an expression

++ Increment operator: increments value by 1

-- Decrement operator: decrements value by 1

! Logical complement operator: inverts the value of a Boolean

**Assignment Operator**

Assignment operators are used in Java to assign values to variables. For example,

int age;

age = 5;

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, 5 is assigned to the variable age.

Let's see some more assignment operators available in Java.

Operator Example Equivalent to

= a = b; a = b;

+= a += b; a = a + b;

-= a -= b; a = a - b;

\*= a \*= b; a = a \* b;

/= a /= b; a = a / b;

%= a %= b; a = a % b;

**Relational Operators**

Relational operators are used to check the relationship between two operands. For example,

// check is a is less than b

a < b;

Here, > operator is the relational operator. It checks if a is less than b or not.

It returns either true or false.

**Logical Operators**

Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator Example Meaning

&& (Logical AND) expression1 && expression2 true only if both expression1 and expression2 are true

|| (Logical OR) expression1 || expression2 true if either expression1 or expression2 is true

! (Logical NOT) !expression true if expression is false and vice versa

**Ternary Operator**

**Ternary Operator in Java**

**A ternary operator evaluates the test condition and executes a block of code based on the result of the condition. if condition is true , expression1 is executed. And, if condition is false , expression2 is executed.**

The ternary operator is a part of Java’s conditional statements. As the name ternary suggests, it is the only operator in Java consisting of three operands.

The ternary operator can be thought of as a simplified version of the if-else statement with a value to be returned.

**The ternary operator take three arguments**:

The first is a comparison argument.

The second is the result upon a true comparison.

The third is the result upon a false comparison.

**Syntax**

The three operands in a ternary operator include:

A boolean expression that evaluates to either true or false.

A value to be assigned if the expression is evaluated to true.

A value to be assigned if the expression is evaluated to false.

1 variable var = (booleanExpression) ? value1 if true : value2 if false

The variable var on the left-hand side of the = (assignment) operator will be assigned:

value1 if the booleanExpression evaluates to true

value2 if the booleanExpression evaluates to false

**Bitwise Operators**

In Java, an operator is a symbol that performs the specified operations. In this section, we will discuss only the bitwise operator and its types with proper examples.

Types of Bitwise Operator

There are six types of the bitwise operator in Java:

Bitwise AND

Bitwise exclusive OR

Bitwise inclusive OR

Bitwise Compliment

Bit Shift Operators

Operators Symbol Uses

Bitwise AND & op1 & op2

Bitwise exclusive OR ^ op1 ^ op2

Bitwise inclusive OR | op1 | op2

Bitwise Compliment ~ ~ op

Bitwise left shift << op1 << op2

Bitwise right shift >> op1 >> op2

Unsigned Right Shift Operator >>> op >>> number of places to shift

Bitwise Operator in Java

In Java, an operator is a symbol that performs the specified operations. In this section, we will discuss only the bitwise operator and its types with proper examples.

Types of Bitwise Operator

There are six types of the bitwise operator in Java:

Bitwise AND

Bitwise exclusive OR

Bitwise inclusive OR

Bitwise Compliment

Bit Shift Operators

Operators Symbol Uses

Bitwise AND & op1 & op2

Bitwise exclusive OR ^ op1 ^ op2

Bitwise inclusive OR | op1 | op2

Bitwise Compliment ~ ~ op

Bitwise left shift << op1 << op2

Bitwise right shift >> op1 >> op2

Unsigned Right Shift Operator >>> op >>> number of places to shift

**Bitwise AND (&)**

It is a binary operator denoted by the symbol &. It returns 1 if and only if both bits are 1, else returns 0.

**Bitwise exclusive OR (^)**

It is a binary operator denoted by the symbol ^ (pronounced as caret). It returns 0 if both bits are the same, else returns 1.

**Bitwise inclusive OR (|)**

It is a binary operator denoted by the symbol | (pronounced as a pipe). It returns 1 if either of the bit is 1, else returns 0.

**Bitwise Complement (~)**

It is a unary operator denoted by the symbol ~ (pronounced as the tilde). It returns the inverse or complement of the bit. It makes every 0 a 1 and every 1 a 0.

**Bit Shift Operators**

Shift operator is used in shifting the bits either right or left. We can use shift operators if we divide or multiply any number by 2. The general format to shift the bit is as follows:

variable << or >> number of places to shift;

**Signed Right Shift Operator (>>)**

The signed right shift operator shifts a bit pattern of a number towards the right with a specified number of positions and fills 0. The operator is denoted by the symbol >>. It also preserves the leftmost bit (sign bit). If 0 is presented at the leftmost bit, it means the number is positive. If 1 is presented at the leftmost bit, it means the number is negative.

**Signed Left Shift Operator (<<)**

The signed left shift operator (<<) shifts a bit pattern to the left. It is represented by the symbol <<. It also preserves the leftmost bit (sign bit). It does not preserve the sign bit.

**Unsigned Right Shift Operator (>>>)**

It shifts a zero at the leftmost position and fills 0. It is denoted by the symbol >>>. Note that the leftmost position after >> depends on the sign bit. It does not preserve the sign bit.

**2. Write a method for increment and decrement operators(++, --)** Incrementing and decrementing are such common operations that Java provides special operators for them. The ++ operator adds one to the current value of an int or char. -- subtracts one. Neither operator works on doubles, booleans or Strings.

Increment and decrement operators each have two forms: pre and post. In above example we have used the post form of increment and decrement operator

Both the pre- and post-increment operators increment the value of the variable by 1. Similarly, the pre- and post-decrement operators decrement the value of the variable by 1. The difference becomes apparent when the variable using these operators is employed in an expression.

**Operators**

‘Logical AND’ Operator(&&): This operator returns true when both the conditions under consideration are satisfied or are true. If even one of the two yields false, the operator results false. For example, cond1 && cond2 returns true when both cond1 and cond2 are true (i.e. non-zero).

Syntax:

condition1 && condition2

'Logical OR' Operator(||): This operator returns true when one of the two conditions under consideration are satisfied or are true. If even one of the two yields true, the operator results true. To make the result false, both the constraints need to return false.

Syntax:

condition1 || condition2

'Logical NOT' Operator(!): Unlike the previous two, this is a unary operator and returns true when the condition under consideration is not satisfied or is a false condition. Basically, if the condition is false, the operation returns true and when the condition is true, the operation returns false.

Syntax:

!(condition)

**Armstrong Number in Java**

Let's write a java program to check whether the given number is armstrong number or not.

Armstrong Number in Java: A positive number is called armstrong number if it is equal to the sum of cubes of its digits for example 0, 1, 153, 370, 371, 407 etc.

Let's try to understand why 153 is an Armstrong number.

153 = (1\*1\*1)+(5\*5\*5)+(3\*3\*3)

where:

(1\*1\*1)=1

(5\*5\*5)=125

(3\*3\*3)=27

So:

1+125+27=153

Let's try to understand why 371 is an Armstrong number.

371 = (3\*3\*3)+(7\*7\*7)+(1\*1\*1)

where:

(3\*3\*3)=27

(7\*7\*7)=343

(1\*1\*1)=1

So:

27+343+1=371

**Loops in Java**

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

for loop

while loop

do-while loop

Java For Loop vs While Loop vs Do While Loop

Comparison for loop while loop do while loop

Introduction The Java for loop is a control flow statement that iterates a part of the programs multiple times.

The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.

The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.

When to use

If the number of iteration is fixed, it is recommended to use for loop.

If the number of iteration is not fixed, it is recommended to use while loop.

If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

Syntax

for(init;condition;incr/decr){

// code to be executed

}

while(condition){

//code to be executed

}

do{

//code to be executed

}while(condition);

Example

**//for loop**

for(int i=1;i<=10;i++){

System.out.println(i);

}

**//while loop**

int i=1;

while(i<=10){

System.out.println(i);

i++;

}

**//do-while loop**

int i=1;

do{

System.out.println(i);

i++;

}while(i<=10);

Syntax for infinitive loop

for(;;){

//code to be executed

}

while(true){

//code to be executed

}

do{

//code to be executed

}while(true);

**4. Arrays**

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

advantages

Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.

Random access: We can get any data located at an index position.

Disadvantages

Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

ADVERTISEMENT

Single Dimensional Array

Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

Instantiation of an Array in Java

arrayRefVar=new datatype[size];

Example of Java Array

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

int a[]={33,3,4,5};//declaration, instantiation and initialization

For-each Loop for Java Array

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

for(data\_type variable:array){

//body of the loop

}

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

//Java Program to demonstrate the case of

//ArrayIndexOutOfBoundsException in a Java Array.

public class TestArrayException{

public static void main(String args[]){

int arr[]={50,60,70,80};

for(int i=0;i<=arr.length;i++){

System.out.println(arr[i]);

}

}}

Test it Now

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

int[][] arr=new int[3][3];//3 row and 3 column

Example to initialize Multidimensional Array in Java

arr[0][0]=1;

arr[0][1]=2;

arr[0][2]=3;

arr[1][0]=4;

arr[1][1]=5;

arr[1][2]=6;

arr[2][0]=7;

arr[2][1]=8;

arr[2][2]=9;

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.

1. **Write a function to add integer values of an array**

// Function to add x in arr

public static int[] addX(int n, int arr[], int x)

{

int i;

1. **Write a function to calculate the average value of an array of integers**

import java.util.Scanner;

public class Sum\_Average

{

public static void main(String[] args)

{

int n, sum = 0;

float average;

Scanner s = new Scanner(System.in);

System.out.print("Enter no. of elements you want in array:");

n = s.nextInt();

int a[] = new int[n];

System.out.println("Enter all the elements:");

for(int i = 0; i < n ; i++)

{

a[i] = s.nextInt();

sum = sum + a[i];

}

System.out.println("Sum:"+sum);

average = (float)sum / n;

System.out.println("Average:"+average);

}

}

**Output:**

$ javac Sum\_Average.java

$ java Sum\_Average

Enter no. of elements you want in array:5

Enter all the elements:

4

7

6

9

3

Sum:29

Average:5.8

**4.Write a function to test if array contains a specific value**

public static boolean contains(int[] arr, int item) {

for (int n : arr) {

if (item == n) {

return true;

}

}

return false;

}

public static void main(String[] args) {

int[] my\_array1 = {

1789, 2035, 1899, 1456, 2013,

1458, 2458, 1254, 1472, 2365,

1456, 2265, 1457, 2456};

System.out.println(contains(my\_array1, 2013));

System.out.println(contains(my\_array1, 2015));

}

}

Output

$javac Exercise5.java

$java -Xmx128M -Xms16M Exercise5

true

false

**5. Write a function to remove a specific element from an array**

import java.util.Arrays;

public class Exercise7 {

public static void main(String[] args) {

int[] my\_array = {25, 14, 56, 15, 36, 56, 77, 18, 29, 49};

System.out.println("Original Array : "+Arrays.toString(my\_array));

// Remove the second element (index->1, value->14) of the array

int removeIndex = 1;

for(int i = removeIndex; i < my\_array.length -1; i++){

my\_array[i] = my\_array[i + 1];

}

// We cannot alter the size of an array , after the removal, the last and second last element in the array will exist twice

System.out.println("After removing the second element: "+Arrays.toString(my\_array));

}

}

Output

Original Array : [25, 14, 56, 15, 36, 56, 77, 18, 29, 49]

After removing the second element: [25, 56, 15, 36, 56, 77, 18, 29, 49, 49]

**6. Write a function to copy an array to another array**

public class CopyArray {

public static void main(String[] args) {

//Initialize array

int [] arr1 = new int [] {1, 2, 3, 4, 5};

//Create another array arr2 with size of arr1

int arr2[] = new int[arr1.length];

//Copying all elements of one array into another

for (int i = 0; i < arr1.length; i++) {

arr2[i] = arr1[i];

}

//Displaying elements of array arr1

System.out.println("Elements of original array: ");

for (int i = 0; i < arr1.length; i++) {

System.out.print(arr1[i] + " ");

}

System.out.println();

//Displaying elements of array arr2

System.out.println("Elements of new array: ");

for (int i = 0; i < arr2.length; i++) {

System.out.print(arr2[i] + " ");

}

}

}

Output

Elements of original array

1 2 3 4 5

Elements of new array:

1 2 3 4 5

**7. Write a function to insert an element at a specific position in the array**

import java.util.Arrays;

public class Exercise9 {

public static void main(String[] args) {

int[] my\_array = {25, 14, 56, 15, 36, 56, 77, 18, 29, 49};

// Insert an element in 3rd position of the array (index->2, value->5)

int Index\_position = 2;

int newValue = 5;

System.out.println("Original Array : "+Arrays.toString(my\_array));

for(int i=my\_array.length-1; i > Index\_position; i--){

my\_array[i] = my\_array[i-1];

}

my\_array[Index\_position] = newValue;

System.out.println("New Array: "+Arrays.toString(my\_array));

}

}

Output

Original Array : [25, 14, 56, 15, 36, 56, 77, 18, 29, 49]

New Array: [25, 14, 5, 56, 15, 36, 56, 77, 18, 29]

**8. Write a function to find the minimum and maximum value of an array**

public class MinAndMax {

public int max(int [] array) {

int max = 0;

for(int i=0; i<array.length; i++ ) {

if(array[i]>max) {

max = array[i];

}

}

return max;

}

public int min(int [] array) {

int min = array[0];

for(int i=0; i<array.length; i++ ) {

if(array[i]<min) {

min = array[i];

}

}

return min;

}

public static void main(String args[]) {

int[] myArray = {23, 92, 56, 39, 93};

MinAndMax m = new MinAndMax();

System.out.println("Maximum value in the array is::"+m.max(myArray));

System.out.println("Minimum value in the array is::"+m.min(myArray));

}

}

Output

Maximum value in the array is::93

Minimum value in the array is::23

**9. Write a function to reverse an array of integer values**

import java.util.Arrays;

public class Exercise11 {

public static void main(String[] args){

int[] my\_array1 = {

1789, 2035, 1899, 1456, 2013,

1458, 2458, 1254, 1472, 2365,

1456, 2165, 1457, 2456};

System.out.println("Original array : "+Arrays.toString(my\_array1));

for(int i = 0; i < my\_array1.length / 2; i++)

{

int temp = my\_array1[i];

my\_array1[i] = my\_array1[my\_array1.length - i - 1];

my\_array1[my\_array1.length - i - 1] = temp;

}

System.out.println("Reverse array : "+Arrays.toString(my\_array1));

}

}

Output

Original array : [1789, 2035, 1899, 1456, 2013, 1458, 2458, 1254, 1472, 2365, 1456, 2165, 1457, 2456]

Reverse array : [2456, 1457, 2165, 1456, 2365, 1472, 1254, 2458, 1458, 2013, 1456, 1899, 2035, 1789]

**10. Write a function to find the duplicate values of an array**

import java.util.Arrays;

public class Exercise12 {

public static void main(String[] args)

{

int[] my\_array = {1, 2, 5, 5, 6, 6, 7, 2};

for (int i = 0; i < my\_array.length-1; i++)

{

for (int j = i+1; j < my\_array.length; j++)

{

if ((my\_array[i] == my\_array[j]) && (i != j))

{

System.out.println("Duplicate Element : "+my\_array[j]);

}

}

}

}

}

Output

Duplicate Element : 2

Duplicate Element : 5

Duplicate Element : 6

**11. Write a program to find the common values between two arrays**

Given two arrays and our task is to find their common elements.

Examples:

Input: Array1 = ["Article", "for", "Geeks", "for", "Geeks"],

Array2 = ["Article", "Geeks", "Geeks"]

Output: [Article,Geeks]

Input: Array1 = ["a", "b", "c", "d", "e", "f"],

Array2 = ["b", "d", "e", "h", "g", "c"]

Output: [b, c, d, e]

Using Iterative Methods

Approach:

Get the two java Arrays.

Iterate through each and every element of the arrays one by one and check whether they are common in both.

Add each common element in the set for unique entries.

Java

// Java Program to find common elements

// in two Arrays

// Using iterative method

import java.io.\*;

import java.util.\*;

class GFG {

private static void FindCommonElemet(String[] arr1,

String[] arr2)

{

Set<String> set = new HashSet<>();

for (int i = 0; i < arr1.length; i++) {

for (int j = 0; j < arr2.length; j++) {

if (arr1[i] == arr2[j]) {

// add common elements

set.add(arr1[i]);

break;

}

}

}

for (String i : set) {

System.out.print(i + " ");

}

}

// main method

public static void main(String[] args)

{

// create Array 1

String[] arr1

= { "Article", "in", "Geeks", "for", "Geeks" };

// create Array 2

String[] arr2 = { "Geeks", "for", "Geeks" };

// print Array 1

System.out.println("Array 1: "

+ Arrays.toString(arr1));

// print Array 2

System.out.println("Array 2: "

+ Arrays.toString(arr2));

System.out.print("Common Elements: ");

// Find the common elements

FindCommonElemet(arr1, arr2);

}

}

Output

Array 1: [Article, in, Geeks, for, Geeks]

Array 2: [Geeks, for, Geeks]

Common Elements: Geeks for

**12. Write a method to remove duplicate elements from an array**

There are two ways to remove duplicates from array java: first using temporary array and second using seperate index.

To remove duplicates from array in java, the array should be in sorted order.

remove duplicate element in an array in java

If array is not sorted then sort given array using Arrays.sort() method.

public class RemoveDuplicateInArrayExample{

public static int removeDuplicateElements(int arr[], int n){

if (n==0 || n==1){

return n;

}

int[] temp = new int[n];

int j = 0;

for (int i=0; i<n-1; i++){

if (arr[i] != arr[i+1]){

temp[j++] = arr[i];

}

}

temp[j++] = arr[n-1];

// Changing original array

for (int i=0; i<j; i++){

arr[i] = temp[i];

}

return j;

}

public static void main (String[] args) {

int arr[] = {10,20,20,30,30,40,50,50};

int length = arr.length;

length = removeDuplicateElements(arr, length);

//printing array elements

for (int i=0; i<length; i++)

System.out.print(arr[i]+" ");

}

}

Output:

10 20 30 40 50

**13. Write a method to find the second largest number in an array**

public class SecondLargestInArrayExample{

public static int getSecondLargest(int[] a, int total){

int temp;

for (int i = 0; i < total; i++)

{

for (int j = i + 1; j < total; j++)

{

if (a[i] > a[j])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

return a[total-2];

}

public static void main(String args[]){

int a[]={1,2,5,6,3,2};

int b[]={44,66,99,77,33,22,55};

System.out.println("Second Largest: "+getSecondLargest(a,6));

System.out.println("Second Largest: "+getSecondLargest(b,7));

}}

Output:

Second Largest: 5

Second Largest: 77

**15. Write a method to find number of even number and odd numbers in an array**We can print odd and even numbers from an array in java by getting remainder of each element and checking if it is divided by 2 or not. If it is divided by 2, it is even number otherwise it is odd number.

public class OddEvenInArrayExample{

public static void main(String args[]){

int a[]={1,2,5,6,3,2};

System.out.println("Odd Numbers:");

for(int i=0;i<a.length;i++){

if(a[i]%2!=0){

System.out.println(a[i]);

}

}

System.out.println("Even Numbers:");

for(int i=0;i<a.length;i++){

if(a[i]%2==0){

System.out.println(a[i]);

}

}

}}

Output

$javac OddEvenInArrayExample.java

$java -Xmx128M -Xms16M OddEvenInArrayExample

Odd Numbers:

1

5

3

Even Numbers:

2

6

2

**16. Write a function to get the difference of largest and smallest value**

The length of the array must be 1 and above.

import java.util.Arrays;

public class Exercise28 {

public static void main(String[] args)

{

int[] array\_nums = {5, 7, 2, 4, 9};

System.out.println("Original Array: "+Arrays.toString(array\_nums));

int max\_val = array\_nums[0];

int min = array\_nums[0];

for(int i = 1; i < array\_nums.length; i++)

{

if(array\_nums[i] > max\_val)

max\_val = array\_nums[i];

else if(array\_nums[i] < min)

min = array\_nums[i];

}

System.out.println("Difference between the largest and smallest values of the said array: "+(max\_val-min));

}

}

Output

Original Array: [5, 7, 2, 4, 9]

Difference between the largest and smallest values of the said array: 7

**17. Write a method to verify if the array contains two specified elements(12,23)**

import java.util.\*;

import java.io.\*;

public class Exercise32 {

public static void main(String[] args)

{

int[] array\_nums = {77, 77, 65, 65, 65, 77};

System.out.println("Original Array: "+Arrays.toString(array\_nums));

int num1 = 77;

int num2 = 65;

System.out.println("Result: "+result(array\_nums, num1, num2));

}

public static boolean result(int[] array\_nums, int num1, int num2) {

for (int number : array\_nums) {

boolean r = number != num1 && number != num2;

if (r) {

return false;

}

}

return true;

}

}

Output

Original Array: [77, 77, 65, 65, 65, 77]

Result: true

**19. Write a function to find the missing number of sorted array of 1 to 100**

Given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

Examples:

Input : arr[] = [1, 2, 3, 4, 6, 7, 8]

Output : 5

Input : arr[] = [1, 2, 3, 4, 5, 6, 8, 9]

Output : 7

**5. Static**

**71. Write a class with 2 static variables, 2 Instance variables, 2 static methods, 2 instance**

**methods and a main method.**

**Java Static Variables**

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

Few Important Points:

Static variables are also known as Class Variables.

Unlike non-static variables, such variables can be accessed directly in static and non-static methods.

**What is instance variable in Java?**

Instance variables in Java are non-static variables which are defined in a class outside any method, constructor or a block. Each instantiated object of the class has a separate copy or instance of that variable. An instance variable belongs to a class.

You must be wondering about what exactly is an Instance? Let me help you by simplifying it.

When you create a new object of the class you create an instance. Consider, if you have a STUDENT class, then

class Student

{

String studentName;

int studentScore;

}

And if you create two STUDENT objects like,

Student student1 = new Student();

Student student2 = new Student();

Then two instances of the class Student will be created.

Features of an instance variable?

The life of an instance variable depends on the life of an Object, i.e., when the object is created, an instance variable also gets created and the same happens when an object is destroyed.

Instance Variable can be used only by creating objects

Every object will have its own copy of Instance variables

Initialization of instance variable is not compulsory. The default value is zero

The declaration is done in a class outside any method, constructor or block

Instance variables are used when the variable has to be known to different methods in a class

Access modifiers can be assigned to instance variables

**Java Static Methods**

Static Methods can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects.

Static methods can be accessed directly in static and non-static methods.

Syntax:

Static keyword followed by return type, followed by method name.

**Instance methods**

Non-static methods, which we will refer to as instance methods or object methods are called using an object and therefore have access to an object’s instance variables. Note the method header will not include the keyword static.

There are three steps to creating and calling an instance method:

Object of the Class: Declare an object of your class in the main method or from outside the class.

// Step 1: declare an object in main or from outside the class

Classname objectName = new Classname();

Method Definition: write the method’s header and body code like below:

// Step 3: Define the method in the class

// method header

public void methodName()

{

// method body for the code

}

Method Call: whenever you want to use the method, call objectName.methodName();

// Step 2: call the object's method

objectName.methodName(); //Step 2

How is this different than calling a static method? Notice the object reference and dot notation before the method name. Since the method may set and get instance variables, you must call the method on an actual instance of the class.

**main method**

Different ways of writing main() method are:

static public void main(String []x)

static public void main(String... args)

A Java application is a public Java class with a main() method. The main() method is the entry point into the application. The signature of the method is always: public static void main(String[] args) Command-line arguments are passed through the args parameter, which is an array of String s.

**Public- it is access specifier from anywhere we can access it Static- it is access modifier we can call the methods directly by class name without creating its objects Void- it is the return type Main- it is a method name String[]args- in java we accept only the string type of argument and store it.**

**Print instance variables in static methods**

We cannot directly access the instance variables within a static method because a static method can only access static variables or static methods.

An instance variable, as the name suggests is tied to an instance of a class. Therefore, accessing it directly from a static method, which is not tied to any specific instance doesn't make sense. Therefore, to access an instance variable, we must have an instance of the class from which we access the instance variable.

Example:

public class Test {

public int instanceVariable = 10;

public static void main(String args[]) {

Test test = new Test();

System.out.println(test.instanceVariable);

}

}

Output:

10

**Print static variables in Instance methods**

Instance method can access the instance methods and instance variables directly.

Instance method can access static variables and static methods directly.

Static methods can access the static variables and static methods directly.

Static methods can’t access instance methods and instance variables directly. They must use reference to object. And static method can’t use this keyword as there is no instance for ‘this’ to refer to.

**Call instance methods in static methods**

Instance method

Instance method requires an object of its class to be created before it can be called. To invoke an instance method, we have to create an Object of the class within which it defined.

Syntax of Instance method

public void geek(String name)

{

// code to be executed...

}

// Return type can be int, float String or user-defined data type.

Static method

Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the object of that class.

**Call static methods in instance methods**

Static methods can be called freely, but instance methods can only be called if you have an instance of the class. The static method needs to either get an instance from somewhere, or create one itself.

**Print all the static, instance variables in main method**

**Call static methods and instance methods in main method**

**6. Strings**

**Different ways creating a string**

You can create a String by −

Step 1 − Assigning a string value wrapped in " " to a String type variable.

String message = "Hello Welcome to Tutorialspoint";

Step 2 − Creating an object of the String class using the new keyword by passing the string value as a parameter of its constructor.

String message = new String ("Hello Welcome to Tutorialspoint");

Step 3 − Passing a character array to the String constructor.

char arr[] = {'H','e','l','l','o'};

String message = new String(arr);

**Concatenating two strings using + operator**

class TestStringConcatenation1{

public static void main(String args[]){

String s="Sachin"+" Tendulkar";

System.out.println(s);//Sachin Tendulkar

}

}

Output:Sachin Tendulkar

**Finding the length of the string**

The length of a string is referred to as the total number of characters it contains.

The length() method

To calculate the length of a string in Java, you can use an inbuilt length() method of the Java string class.

In Java, strings are objects created using the string class and the length() method is a public member method of this class. So, any variable of type string can access this method using the . (dot) operator.

The length() method counts the total number of characters in a String.

Method signature

The signature of the length() method is as follows:

1 public int length()

The return type of the length() method is int.

Example

Let’s calculate & printout the length of a string using the length() method.

class CalcLength {

public static void main( String args[] ) {

String name = "educative"; //Initilizing a String Object name

int length = name.length(); //Calling the inbuilt lenght() method

System.out.println("The length of the String \""+name+"\" is: " +length); }

}

Run

The length of the string name is 9:

educative

**Extract a string using Substring**

You can extract a substring from a String using the substring() method of the String class to this method you need to pass the start and end indexes of the required substring.

Example

Live Demo

public class Substring {

public static void main(String args[]) {

String str = "Welcome to Tutorialspoint";

String sub = str.substring(10, 25);

System.out.println(sub);

}

}

Output

Tutorialspoint

**Searching in strings using indexOf()**

indexOf()- This method searches forward from the beginning of the string and returns the index within this string of the first occurrence of the specified character/substring. If a character or substring is not found indexOf() returns -1.

Search a string for the first occurrence of "planet":

String myStr = "Hello planet earth, you are a great planet.";

System.out.println(myStr.indexOf("planet"));

The indexOf() method returns the position of the first occurrence of specified character(s) in a string.

Tip: Use the lastIndexOf method to return the position of the last occurrence of specified character(s) in a string.

Syntax

There are 4 indexOf() methods:

public int indexOf(String str)

public int indexOf(String str, int fromIndex)

public int indexOf(int char)

public int indexOf(int char, int fromIndex)

Parameter Values

Parameter Description

Str A String value, representing the string to search for

fromIndex An int value, representing the index position to start the search from

char An int value, representing a single character, e.g 'A', or a Unicode value

Technical Details

Returns: An int value, representing the index of the first occurrence of the character in the string, or -1 if it never occurs

Example

Find the first occurrence of the letter "e" in a string, starting the search at position 5:

public class Main {

public static void main(String[] args) {

String myStr = "Hello planet earth, you are a great planet.";

System.out.println(myStr.indexOf("e", 5));

}

}

**Matching a String Against a Regular Expression With matches()**

There are three variants of matches() method.

1. String matches() : This method tells whether or not this string matches the given regular expression. An invocation of this method of the form str.matches(regex) yields exactly the same result as the expression Pattern.matches(regex, str).

Syntax:

public boolean matches(String regex)

Parameters

regex : the regular expression to which this string is to be matched.

Return Value

This method returns true if, and only if, this string matches the given regular expression.

2. String regionMatches() (with ignoreCase) :This method has two variants which can be used to test if two string regions are equal.

Syntax

public boolean regionMatches(boolean ignoreCase,

int str\_strt,

String other,

int other\_strt,

int len)

3. String regionMatches() :This method has two variants which can be used to test if two string regions are equal.

Syntax

public boolean regionMatches(int str\_strt,

String other,

int other\_strt,

int len)

**Comparing strings using the methods equals()**

Compare strings to find out if they are equal:

String myStr1 = "Hello";

String myStr2 = "Hello";

String myStr3 = "Another String";

System.out.println(myStr1.equals(myStr2)); // Returns true because they are equal

System.out.println(myStr1.equals(myStr3)); // false

**Definition and Usage**

The equals() method compares two strings, and returns true if the strings are equal, and false if not.

Tip: Use the compareTo() method to compare two strings lexicographically.

**Syntax**

public boolean equals(Object anotherObject)

**Parameter Values**

**Parameter Description**

AnotherObject An Object, representing the other string to be compared

**Technical Details**

**Returns:** A boolean value:

true - if the strings are equal

false - if the strings are not equal

**Overrides:** equals in class Object

**equalsIgnoreCase(), startsWith(), endsWith() and compareTo()**

**String comparison using equals() and equalsIgnoreCase() methods**

**boolean equals(Object anObject)-** This method is used to compare the content of two strings. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

**boolean equalsIgnoreCase(String anotherString)-** Comparison of strings using equals() method is case sensitive, if you want case considerations to be ignored then use equalsIgnoreCase method. When two strings are compared using this method they are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

String comparison using compareTo() and compareToIgnoreCase() methods

int compareTo(String anotherString)- Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.

int compareToIgnoreCase(String str)- Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.

In lexicographical comparison if two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both.

Compare String portions using startsWith() and endsWith() methods

boolean startsWith(String str)– Tests if this string starts with the passed argument. Returns true if substring matches at the start, false otherwise.

boolean startsWith(String str, int toffset)– Tests if the substring of this string beginning at the specified index starts with the passed argument. Returns true if substring matches at the start, false otherwise.

boolean endsWith(String str)– Tests if this string ends with the passed argument. Returns true if substring matches at the end, false otherwise.

**Trimming strings with trim()**

The java string trim() method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

The string trim() method doesn't omits middle spaces.

Internal implementation

public String trim() {

int len = value.length;

int st = 0;

char[] val = value; /\* avoid getfield opcode \*/

while ((st < len) && (val[st] <= ' ')) {

st++;

}

while ((st < len) && (val[len - 1] <= ' ')) {

len--;

}

return ((st > 0) || (len < value.length)) ? substring(st, len) : this;

}

**Signature**

The signature or syntax of string trim method is given below:

public String trim()

**Returns**

string with omitted leading and trailing spaces

**Java String trim() method example**

public class StringTrimExample{

public static void main(String args[]){

String s1=" hello string ";

System.out.println(s1+"javatpoint");//without trim()

System.out.println(s1.trim()+"javatpoint");//with trim()

}}

Test it Now

hello string javatpoint

hello stringjavatpoint

**Replacing characters in strings with replace()**

The java string replace() method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Since JDK 1.5, a new replace() method is introduced, allowing you to replace a sequence of char values.

**Signature**

There are two type of replace methods in java string.

public String replace(char oldChar, char newChar)

and

public String replace(CharSequence target, CharSequence replacement)

**Parameters**

oldChar : old character

newChar : new character

target : target sequence of characters

replacement : replacement sequence of characters

**Returns**

replaced string

**Java String replace(char old, char new) method example**

public class ReplaceExample1{

public static void main(String args[]){

String s1="javatpoint is a very good website";

String replaceString=s1.replace('a','e');//replaces all occurrences of 'a' to 'e'

System.out.println(replaceString);

}}

Test it Now

jevetpoint is e very good website

**Java String replace(CharSequence target, CharSequence replacement) method example**

public class ReplaceExample2{

public static void main(String args[]){

String s1="my name is khan my name is java";

String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"

System.out.println(replaceString);

}}

Test it Now

my name was khan my name was java

**Splitting strings with split()**

The java string split() method splits this string against given regular expression and returns a char array.

**Signature**

There are two signature for split() method in java string.

public String split(String regex)

and,

public String split(String regex, int limit)

Parameter

regex : regular expression to be applied on string.

limit : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

**Returns**

array of strings

**Throws**

PatternSyntaxException if pattern for regular expression is invalid

Since

1.4

**Java String split() method example**

The given example returns total number of words in a string excluding space only. It also includes special characters.

public class SplitExample{

public static void main(String args[]){

String s1="java string split method by javatpoint";

String[] words=s1.split("\\s");//splits the string based on whitespace

//using java foreach loop to print elements of string array

for(String w:words){

System.out.println(w);

}

}}

Test it Now

java

string

split

method

by

javatpoint

**Converting Numbers to Strings with valueOf()**

String.valueOf()

Pass your integer (as an int or Integer) to this method and it will return a string:

String.valueOf(Integer(123));

public class main{

public static void main(String[] args){

Integer i = new Integer(123);

System.out.println("Before conversion: " + i.getClass().getName());

System.out.println("After conversion: " + String.valueOf(i).getClass().getName());

}

}

**Converting integer objects to Strings**We can convert Object to String in java using toString() method of Object class or String.valueOf(object) method.

You can convert any object to String in java whether it is user-defined class, StringBuilder, StringBuffer or anything else.

Here, we are going to see two examples of converting Object into String. In the first example, we are going to convert Emp class object into String which is an user-defined class. In second example, we are going to convert StringBuilder to String.

class Emp{}

public class ObjectToStringExample{

public static void main(String args[]){

Emp e=new Emp();

String s=e.toString();

String s2=String.valueOf(e);

System.out.println(s);

System.out.println(s2);

}}

Test it Now

Output:

Emp@2a139a55

Emp@2a139a55

Java Convert Object to String

We can convert Object to String in java using toString() method of Object class or String.valueOf(object) method.

Java Convert Object to String

You can convert any object to String in java whether it is user-defined class, StringBuilder, StringBuffer or anything else.

Here, we are going to see two examples of converting Object into String. In the first example, we are going to convert Emp class object into String which is an user-defined class. In second example, we are going to convert StringBuilder to String.

Java Object to String Example: Converting User-defined class

Let's see the simple code to convert String to Object in java.

class Emp{}

public class ObjectToStringExample{

public static void main(String args[]){

Emp e=new Emp();

String s=e.toString();

String s2=String.valueOf(e);

System.out.println(s);

System.out.println(s2);

}}

Test it Now

Output:

Emp@2a139a55

Emp@2a139a55

As you can see above, a reference id of Emp class is printed on the console.

Java Object to String Example: Converting StringBuilder

Let's see the simple code to convert StringBuilder object to String in java.

public class ObjectToStringExample2{

public static void main(String args[]){

String s="hello";

StringBuilder sb=new StringBuilder(s);

sb.reverse();

String rev=sb.toString();//converting StringBuilder to String

System.out.println("String is: "+s);

System.out.println("Reverse String is: "+rev);

}}

Test it Now

Output:

String is: hello

Reverse String is: olleh

Now you can write the code to check the palindrome string.

public class ObjectToStringExample3{

public static void main(String args[]){

String s="nitin";

StringBuilder sb=new StringBuilder(s);

sb.reverse();

String rev=sb.toString();//converting StringBuilder to String

if(s.equals(rev)){

System.out.println("Palindrome String");

}else{

System.out.println("Not Palindrome String");

}

}}

Test it Now

Output:

Palindrome String

So, you can convert any Object to String in java using toString() or String.valueOf(object) methods.

**Converting to uppercase and lowercase**

To convert or change uppercase string or character to lowercase string or character in Java Programming, use the ASCII values of character to convert any character from uppercase to lowercase as shown in the first program. And the second program uses the method toLowerCase() to convert string from uppercase to lowercase.

Convert a string to upper case and lower case letters:

String txt = "Hello World";

System.out.println(txt.toUpperCase());

System.out.println(txt.toLowerCase());

Definition and Usage

The toUpperCase() method converts a string to upper case letters.

Note: The toLowerCase() method converts a string to lower case letters.

Syntax

public String toUpperCase()

Parameters

None.

Technical Details

Returns: A String value, representing the new string converted to upper case

**7. Inheritance**

**Inheritance in Java**

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

**Why use inheritance in java**

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

**Terms used in Inheritance**

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**The syntax of Java Inheritance**

class Subclass-name extends Superclass-name

{

//methods and fields

}

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Next →← Prev

Inheritance in Java

Inheritance

Types of Inheritance

**Why multiple inheritance is not possible in Java in case of class?**

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

class Subclass-name extends Superclass-name

{

//methods and fields

}

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example

**Employee**

**Salary:float**

**Programmer**

**Bonus:int**

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

class Employee{

float salary=40000;

}

class Programmer extends Employee{

int bonus=10000;

public static void main(String args[]){

Programmer p=new Programmer();

System.out.println("Programmer salary is:"+p.salary);

System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

Test it Now

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

**Types of inheritance in java**

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

Inheritance in Java

Inheritance

Types of Inheritance

Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Terms used in Inheritance

**Class**: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

**Sub Class/Child Class**: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

**Super Class/Parent Class**: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

**Reusability**: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**The syntax of Java Inheritance**

class Subclass-name extends Superclass-name

{

//methods and fields

}

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example

Inheritance in Java

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

class Employee{

float salary=40000;

}

class Programmer extends Employee{

int bonus=10000;

public static void main(String args[]){

Programmer p=new Programmer();

System.out.println("Programmer salary is:"+p.salary);

System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

Test it Now

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

**Note: Multiple inheritance is not supported in Java through class.**

When one class inherits multiple classes, it is known as multiple inheritance. For Example:

Single Inheritance Example

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void bark(){System.out.println("barking...");}

}

class TestInheritance{

public static void main(String args[]){

Dog d=new Dog();

d.bark();

d.eat();

}}

Output:

barking...

eating...

**Multilevel Inheritance Example**

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void bark(){System.out.println("barking...");}

}

class BabyDog extends Dog{

void weep(){System.out.println("weeping...");}

}

class TestInheritance2{

public static void main(String args[]){

BabyDog d=new BabyDog();

d.weep();

d.bark();

d.eat();

}}

Output:

weeping...

barking...

eating...

**Hierarchical Inheritance Example**

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void bark(){System.out.println("barking...");}

}

class Cat extends Animal{

void meow(){System.out.println("meowing...");}

}

class TestInheritance3{

public static void main(String args[]){

Cat c=new Cat();

c.meow();

c.eat();

//c.bark();//C.T.Error

}}

Output:

meowing...

eating...

**Q)** **Why multiple inheritance is not supported in java?**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

class A{

void msg(){System.out.println("Hello");}

}

class B{

void msg(){System.out.println("Welcome");}

}

class C extends A,B{//suppose if it were

public static void main(String args[]){

C obj=new C();

obj.msg();//Now which msg() method would be invoked?

}

}

Test it Now

Compile Time Error

**2. A is a super class. B is a sub class of A. C is a sub class of B.**

**Create three methods in each class, 2 methods are specific to each class and third method (override method) should be in all three Classes A, B and C**

//Java Program to demonstrate the real scenario of Java Method Overriding

//where three classes are overriding the method of a parent class.

//Creating a parent class.

class Bank{

int getRateOfInterest(){return 0;}

}

//Creating child classes.

class SBI extends Bank{

int getRateOfInterest(){return 8;}

}

class ICICI extends Bank{

int getRateOfInterest(){return 7;}

}

class AXIS extends Bank{

int getRateOfInterest(){return 9;}

}

//Test class to create objects and call the methods

class Test2{

public static void main(String args[]){

SBI s=new SBI();

ICICI i=new ICICI();

AXIS a=new AXIS();

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}

}

Output

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

**Create a class with main method. Create an object for each class A, B and C in main method and call every method of each class using its own object/instance.**

**class Student{**

**int id;**

**String name;**

**}**

**class TestStudent3{**

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

**//Creating objects**

**Student s1=new Student();**

**Student s2=new Student();**

**//Initializing objects**

**s1.id=101;**

**s1.name="Sonoo";**

**s2.id=102;**

**s2.name="Amit";**

**//Printing data**

**System.out.println(s1.id+" "+s1.name);**

**System.out.println(s2.id+" "+s2.name);**

**}**

**}**

**Output:**

**101 Sonoo**

**102 Amit**

**What is an object in Java**

**object in Java**

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

An object is a real-world entity.

An object is a runtime entity.

The object is an entity which has state and behavior.

The object is an instance of a class.

Objects and Classes in Java

Object in Java

Class in Java

Instance Variable in Java

Method in Java

Example of Object and class that maintains the records of student

Anonymous Object

In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

ADVERTISEMENT

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java

object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Characteristics of Object in Java

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

An object is a real-world entity.

An object is a runtime entity.

The object is an entity which has state and behavior.

The object is an instance of a class.

**What is a class in Java**

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

Fields

Methods

Constructors

Blocks

Nested class and interface

**Instance variable in Java**

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

Code Reusability

Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

3 Ways to initialize object

There are 3 ways to initialize object in Java.

By reference variable

By method

By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

By new keyword

By newInstance() method

By clone() method

By deserialization

By factory method etc.

**Call an overridden method with super class reference to B and C class’s objects**

**Overriding in Java**

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

// A Simple Java program to demonstrate

// method overriding in java

// Base Class

class Parent {

void show()

{

System.out.println("Parent's show()");

}

}

// Inherited class

class Child extends Parent {

// This method overrides show() of Parent

@Override

void show()

{

System.out.println("Child's show()");

}

}

// Driver class

class Main {

public static void main(String[] args)

{

// If a Parent type reference refers

// to a Parent object, then Parent's

// show is called

Parent obj1 = new Parent();

obj1.show();

// If a Parent type reference refers

// to a Child object Child's show()

// is called. This is called RUN TIME

// POLYMORPHISM.

Parent obj2 = new Child();

obj2.show();

}

}

Output:

Parent's show()

Child's show()

**Rules for method overriding:**

Overriding and Access-Modifiers: The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

**Runtime Polymorphism with Data Members/Instance variables, Repeat the above**

**process only for data members.**

In Java, we can override methods only, not the variables (data members), so runtime polymorphism cannot be achieved by data members.

// Java program to illustrate the fact that

// runtime polymorphism cannot be achieved

// by data members

// class A

class A

{

int x = 10;

}

// class B

class B extends A

{

int x = 20;

}

// Driver class

public class Test

{

public static void main(String args[])

{

A a = new B(); // object of type B

// Data member of class A will be accessed

System.out.println(a.x);

}

}

Output:

10

Explanatio : In above program, both the class A(super class) and B(sub class) have a common variable ‘x’. Now we make object of class B, referred by ‘a’ which is of type of class A. Since variables are not overridden, so the statement “a.x” will always refer to data member of super class.

**8. Access Modifiers**

Access modifiers in Java are used to define the accessibility and scope of a class or data variable. An access modifier must be specified whenever a variable or method is being defined.

Main access modifiers

There are 4 access modifiers in Java:

Private: This access modifier ensures that a member can only be accessed from within the class. Java does not offer the Private word with a class.

Public: This access modifier allows the data variable or method to be accessed anywhere. A variable can be accessed within the class or outside the class.

Protected: In a child class, the protected modified variable can be accessed within the package and outside the package. The access level of a protected modifier is only inside and outside the package through the child class.

Default: The default access modified variable can only be accessed within the same package. The variable cannot be accessed outside the package. If no access modifier is mentioned, then the Default modifier is automatically added.

**Create a class with PRIVATE fields, private method and a main method. Print the fields in main method. Call the private method in main method.**

**Create a sub class and try to access the private fields and methods from sub class**

Java Reflection - Private Fields and Methods

**Accessing Private Fields**

To access a private field you will need to call the Class.getDeclaredField(String name) or Class.getDeclaredFields() method. The methods Class.getField(String name) and Class.getFields() methods only return public fields, so they won't work. Here is a simple example of a class with a private field, and below that the code to access that field via Java Reflection:

public class PrivateObject {

private String privateString = null;

public PrivateObject(String privateString) {

this.privateString = privateString;

}

}

PrivateObject privateObject = new PrivateObject("The Private Value");

Field privateStringField = PrivateObject.class.

getDeclaredField("privateString");

privateStringField.setAccessible(true);

String fieldValue = (String) privateStringField.get(privateObject);

System.out.println("fieldValue = " + fieldValue);

This code example will print out the text "fieldValue = The Private Value", which is the value of the private field privateString of the PrivateObject instance created at the beginning of the code sample.

Notice the use of the method PrivateObject.class.getDeclaredField("privateString"). It is this method call that returns the private field. This method only returns fields declared in that particular class, not fields declared in any superclasses.

Notice the line in bold too. By calling Field.setAcessible(true) you turn off the access checks for this particular Field instance, for reflection only. Now you can access it even if it is private, protected or package scope, even if the caller is not part of those scopes. You still can't access the field using normal code. The compiler won't allow it.

**Accessing Private Methods**

To access a private method you will need to call the Class.getDeclaredMethod(String name, Class[] parameterTypes) or Class.getDeclaredMethods() method. The methods Class.getMethod(String name, Class[] parameterTypes) and Class.getMethods() methods only return public methods, so they won't work. Here is a simple example of a class with a private method, and below that the code to access that method via Java Reflection:

public class PrivateObject {

private String privateString = null;

public PrivateObject(String privateString) {

this.privateString = privateString;

}

private String getPrivateString(){

return this.privateString;

}

}

PrivateObject privateObject = new PrivateObject("The Private Value");

Method privateStringMethod = PrivateObject.class.

getDeclaredMethod("getPrivateString", null);

privateStringMethod.setAccessible(true);

String returnValue = (String)

privateStringMethod.invoke(privateObject, null);

System.out.println("returnValue = " + returnValue);

This code example will print out the text "returnValue = The Private Value", which is the value returned by the method getPrivateString() when invoked on the PrivateObject instance created at the beginning of the code sample.

Notice the use of the method PrivateObject.class.getDeclaredMethod("privateString"). It is this method call that returns the private method. This method only returns methods declared in that particular class, not methods declared in any superclasses.

Notice the line in bold too. By calling Method.setAcessible(true) you turn off the access checks for this particular Method instance, for reflection only. Now you can access it even if it is private, protected or package scope, even if the caller is not part of those scopes. You still can't access the method using normal code. The compiler won't allow it.

**Create a class with DEFAULT fields and methods. Access these fields and methods**

**from any other class in the same package**

**. Default access modifier**

When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package. Lets see an example to understand this:

**Default Access Modifier Example in Java**

To understand this example, you must have the knowledge of packages in java.

In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.

**Addition.java**

package abcpackage;

public class Addition {

/\* Since we didn't mention any access modifier here, it would

\* be considered as default.

\*/

int addTwoNumbers(int a, int b){

return a+b;

}

}

Test.java

package xyzpackage;

/\* We are importing the abcpackage

\* but still we will get error because the

\* class we are trying to use has default access

\* modifier.

\*/

import abcpackage.\*;

public class Test {

public static void main(String args[]){

Addition obj = new Addition();

/\* It will throw error because we are trying to access

\* the default method in another package

\*/

obj.addTwoNumbers(10, 21);

}

}

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The method addTwoNumbers(int, int) from the type Addition is not visible

at xyzpackage.Test.main(Test.java:12)

**Create a class with PROTECTED fields and methods. Access these fields and methods**

**from any other class in the same package.**

**Also, Access the PROTECTED fields and methods from child class located in a different**

**package**

**Access the PROTECTED fields and methods from any class in different package**

A Java protected keyword is an access modifier. It can be assigned to variables, methods, constructors and inner classes.

The protected Keyword

While elements declared as private can be accessed only by the class in which they're declared, the protected keyword allows access from sub-classes and members of the same package.

By using the protected keyword, we make decisions about which methods and fields should be considered internals of a package or class hierarchy, and which are exposed to outside code.

3. Declaring protected Fields, Methods, and Constructors

First, let's create a class named FirstClass containing a protected field, method, and constructor:

public class FirstClass {

protected String name;

protected FirstClass(String name) {

this.name = name;

}

protected String getName() {

return name;

}

}

With this example, by using the protected keyword, we've granted access to these fields to classes in the same package as FirstClass and to sub-classes of FirstClass.

4. Accessing protected Fields, Methods, and Constructors

4.1 From the Same Package

Now, let's see how we can access protected fields by creating a new GenericClass declared in the same package as FirstClass:

public class GenericClass {

public static void main(String[] args) {

FirstClass first = new FirstClass("random name");

System.out.println("FirstClass name is " + first.getName());

first.name = "new name";

}

}

As this calling class is in the same package as FirstClass, it's allowed to see and interact with all the protected fields, methods, and constructors.

4.2. From a Different Package

Let's now try to interact with these fields from a class declared in a different package from FirstClass:

public class SecondGenericClass {

public static void main(String[] args) {

FirstClass first = new FirstClass("random name");

System.out.println("FirstClass name is "+ first.getName());

first.name = "new name";

}

}

As we can see, we get compilation errors:

freestar

The constructor FirstClass(String) is not visible

The method getName() from the type FirstClass is not visible

The field FirstClass.name is not visible

That's exactly what we were expecting by using the protected keyword. This is because SecondGenericClass is not in the same package as FirstClass and does not subclass it.

4.3 From a Sub-Class

Let's now see what happens when we declare a class extending FirstClass but declared in a different package:

public class SecondClass extends FirstClass {

public SecondClass(String name) {

super(name);

System.out.println("SecondClass name is " + this.getName());

this.name = "new name";

}

}

As expected, we can access all the protected fields, methods, and constructors. This is because SecondClass is a sub-class of FirstClass.

5. protected Inner Class

In the previous examples, we saw protected fields, methods, and constructors in action. There is one more particular case — a protected inner class.

Let's create this empty inner class inside our FirstClass:

freestar

package com.baeldung.core.modifiers;

public class FirstClass {

// ...

protected static class InnerClass {

}

}

As we can see, this is a static inner class, and so can be constructed from outside of an instance of FirstClass. However, as it is protected, we can only instantiate it from code in the same package as FirstClass.

5.1 From the Same Package

To test this, let's edit our GenericClass:

public class GenericClass {

public static void main(String[] args) {

// ...

FirstClass.InnerClass innerClass = new FirstClass.InnerClass();

}

}

As we can see, we can instantiate the InnerClass without any problem because GenericClass is in the same package as FirstClass.

5.2. From a Different Package

Let's try to instantiate an InnerClass from our SecondGenericClass which, as we remember, is outside FirstClass' package:

public class SecondGenericClass {

public static void main(String[] args) {

// ...

FirstClass.InnerClass innerClass = new FirstClass.InnerClass();

}

}

As expected, we get a compilation error:

The type FirstClass.InnerClass is not visible

5.3. From a Sub-Class

Let's try to do the same from our SecondClass:

public class SecondClass extends FirstClass {

public SecondClass(String name) {

// ...

FirstClass.InnerClass innerClass = new FirstClass.InnerClass();

}

}

We were expecting to instantiate our InnerClass with ease. However, we are getting a compilation error here too:

The constructor FirstClass.InnerClass() is not visible

Let's take a look at our InnerClass declaration:

freestar

protected static class InnerClass {

}

The main reason we are getting this error is that the default constructor of a protected class is implicitly protected. In addition, SecondClass is a sub-class of FirstClass but is not a sub-class of InnerClass. Finally, we also declared SecondClass outside FirstClass' package.

For all these reasons, SecondClass can't access the protected InnerClass constructor.

If we wanted to solve this issue and allow our SecondClass to instantiate an InnerClass object, we could explicitly declare a public constructor:

protected static class InnerClass {

public InnerClass() {

}

}

By doing this, we no longer get a compilation error, and we can now instantiate an InnerClass from SecondClass.

6. Conclusion

In this quick tutorial, we discussed the protected access modifier in Java. With it, we can ensure exposing only the required data and methods to sub-classes and classes in the same package.

As always, the example code is available over on GitHub.

**Create a class with PUBLIC fields and methods.**

**Access the public methods and fields from any class in the same package or different**

**package**

public Access Modifier

The Java access modifier public means that all code can access the class, field, constructor or method, regardless of where the accessing code is located. The accessing code can be in a different class and different package.

Here is a public access modifier example:

public class Clock {

public long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

The time field in the Clock class is marked with the public Java access modifier. Therefore, the ClockReader class can access the time field in the Clock no matter what package the ClockReader is located in.

Polymorphism in Java is a concept by which we can perform a single action in different ways. ... So polymorphism means many forms. There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

1. **How to create a class, object, method and its signature**

**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 MyClass, so now we can use this to create objects.

To create an object of MyClass, 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);

}

}

**Java Methods**

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.

void means that this method does not have a return value.

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 Signatures**

About:

A signature is a list that specifies a class constructor, an instance method, or a static method, thereby distinguishing it from other constructors, instance methods, or static methods.

Two forms of signatures are accepted: simple and full. A simple signature is a single element list containing the name of the method or constructor. In most cases a simple signature is all that is needed as the Java method resolver is able to disambiguate overloaded Java methods based on the types of Java object arguments. There are some cases where the Java method resolver is unable to determine which Java method you intended to invoke so you will need to use the full signature for the method or constructor. The full signature is used to distinguish between two or more methods or constructors that have the same number of arguments. The full signature of a method is a Tcl list containing the method name followed by the name of the Java object type for each parameter of the method.

**3. Write a program for a Single line comment, multi-line and documentation comments**

**Java Comments**

The Java comments are the statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code.

**Types of Java Comments**

There are three types of comments in Kava.

Single Line Comment

Multi Line Comment

Documentation Comment

**1) Java Single Line Comment**

The single line comment is used to comment only one line.

Syntax:

//This is single line comment

**2) Java Multi Line Comment**

The multi line comment is used to comment multiple lines of code.

Syntax:

/\* This is multi line comment \*/

**3) Java Documentation Comment**

The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

Syntax:

/\*\* This is documentation comment \*/

This kind of Java comments is utilized by large code for a programming bundle since it produces a documentation page for reference, which can be utilized for getting data about strategies, its parameters, and so forth.

This type of comments are used generally when writing code for a project/software package, since it helps to generate a documentation page for reference, which can be used for getting information about methods present, its parameters, etc.

Syntax

/\*\*Comment start

\*

\*tags are used in order to specify a parameter

\*or method or heading

\*HTML tags can also be used

\*such as <h1>

\*

\*comment ends\*/

**4. Define variables for different Data Types int, Boolean, char, float, double and print on the Console?**

**Data Types in Java**

**int**

The int data type is used to store integers. Integers are numbers which don’t have decimal. For example, -5, 0, 6, etc.

class Test {

public static void main(String[] args) {

int num;

num = 10;

System.out.println(num);

}

}

Output

10

The variable num is declared of type int and is assigned an integer value 10.

**double**

The double data type is used to store double-precision 64-bit floating point numbers. Floating point numbers are numbers which have decimal. In other words, the double data type is used to store numbers having decimal. For example, -5.64, 10.228, etc.

class Test {

public static void main(String[] args) {

double num;

num = 10.5;

System.out.println(num);

}

}

Output

10.5

The variable num is declared of type double and is assigned a floating point value 10.5.

Note that 8 is an int but 8.0 is a double.

**float**

The float data type is used to store single-precision 32-bit floating point numbers. A float value should always end with f or F. For example, -5.64f, 10.228F, etc.

class Test {

public static void main(String[] args) {

float num;

num = 10.5f;

System.out.println(num);

}

}

Output

10.5

In this example, we assigned the value 10.5f and not 10.5 to the variable num because 10.5 is a double. To tell the compiler to consider 10.5 as a float, we assigned 10.5f to the variable.

**char**

The char data type is used to store a character. A character value must be written within single quotes ' '. For example, ‘a’, ‘B’, ‘@’, etc.

class Test {

public static void main(String[] args) {

char ch;

ch = 'e';

System.out.println(ch);

}

}

Output

e

Here, ch is the name of a variable of type char which is assigned a character value 'e'.

**String**

The String data type is used to store a string. A string is a sequence of characters. For example, “Hello” is a string having characters ‘H’, ‘e’, ‘l’, ‘l’ and ‘o’.

A string value must be enclosed within double quotes " ". In fact, any value enclosed within double quotes " " is a string. Some examples of string are “Hello World”, “Hello123”, "123" and “Name: John”.

class Test {

public static void main(String[] args) {

String msg;

msg = "Let's learn Java";

System.out.println(msg);

}

}

Output

Let's learn Java

In this example, the variable msg is declared of type String and is assigned the string "Let's learn Java".

As we know that any value enclosed within double quotes “ “ is a string, so values like “10” and “10+2” are also strings.

Note that 10 is an int but “10” is a String.

Look at the following example.

class Test {

public static void main(String[] args) {

int a = 10 + 2;

String b = "10 + 2";

System.out.println(a);

System.out.println(b);

}

}

Output

12

10 + 2

In the above example, the integer variable a is assigned the expression 10 + 2. This expression first gets evaluated to 12 and then 12 is assigned to a. On the other hand, the string variable b is assigned the string "10 + 2".

We will learn more about String later in the topic Strings.

**boolean**

The boolean data type consists of two values - true and false.

class Test {

public static void main(String[] args) {

boolean a, b;

a = true;

b = false;

System.out.println(a);

System.out.println(b);

}

}

Output

true

false

The variables a and b are declared of type boolean and are assigned the values true and false respectively.

So these were all the basic data types. Now, let’s look at the range of values that different data types can take.

The following table states different data types along with the maximum and minimum value they can take.

Data Type Maximum Value Minimum Value

int 2,147,483,647

- 2,147,483,648

float 3.4028235E38

1.4E-45

double 1.7976931348623157E308

4.9E-324

char 65,535

0

short 32767

-32767

long 9223372036854775807

-9223372036854775808

10E5 means 105 i.e. 100000.

Let's see an example of double, char and boolean values.

class Test {

public static void main(String[] args) {

double b = 123.43555;

char c = 'e';

boolean d = true;

System.out.println("Double: " + b);

System.out.println("Character: " + c);

System.out.println("Boolean: " + d);

}

}

Output

Double: 123.43555

Character: e

Boolean: true

You must have understood the code. While printing, + joined a string and the value of a variable in each System.out.println() method.

In the first System.out.println() method, the string "Double: " and the value of b (because b is not inside " ") are combined and printed.

Look at another example.

class Test {

public static void main(String[] args) {

int x = 1, y = 5;

System.out.println("x");

System.out.println("y");

System.out.println("x" + "y");

System.out.println(x);

System.out.println(y);

System.out.println(x + y);

}

}

Output

x

y

xy

1

5

6

Here, whatever is written within " " got printed as it is, without getting evaluated. Whatever is not inside " " got evaluated first and then their values got printed. For example, "x"+"y" got printed as xy (without evaluation) but x+y got evaluated first as 1+5 i.e., 6 and then 6 got printed.

**Primitive and Non-Primitive Data types**

Using data types is so easy, isn't it?

All the data types are broadly classified into primitive and non-primitive.

**Primitive Data type**

Primitive data types are predefined (already defined) data types in Java.

There are eight predefined data types in Java, which are

int, float, double, short, long, char, boolean, byte

**Non-Primitive Data type**

Non-primitive data types are defined by the programmer. Some examples of non-primitive data types are Array, Class and Interface.

At present, there is no need to go into the details of non-primitive data types as we will learn about them later.

**Type Casting**

Suppose we are writing a program and we have an integer variable having a value 10 (an integer) and at some point of time we want it to be a string i.e., “10”. Or a more practical case would be to convert a double (10.2) to an integer (10). We can easily do so in Java using type casting.

Type Casting is the conversion of a value from one data type to another data type. For example, we can convert a double value to an int value or a char value to an int value.

Type Conversions are of two types - implicit and explicit.

**Implicit Conversion**

Suppose we are adding two numbers. The first number is of type int and the second number is of type double. We cannot add an int and a double because both the numbers have to be of the same data type i.e. either both are int or both are double. Since double is a larger data type than int, therefore while adding, the int variable automatically gets converted into double and then both the double variables add up.

Order of size of data types:

double > float > long > int > char > short

From the above order, we can see that double is the largest data type and short is the smallest data type. Any smaller data type gets implicitly converted into a larger data type when performing arithmetic operations or in any such other expression.

For example, when adding a value of type int and a value of type long, the value of type int gets automatically converted to long and then both the values get added.

Similarly, a char variable gets converted into an int while performing some arithmetic operation.

class Test {

public static void main(String[] args) {

int n = 10;

char ch = 'h';

int sum = n + ch;

System.out.println(sum);

}

}

Output

114

In the above program, when the variables n and ch are added, the integer value (ASCII value) of ch i.e. 104 is added to the value of n to produce a sum of 114. Note that every character has an ASCII value. You can get the ASCII chart from here.

**Explicit Conversion**

We know that a smaller data type can be implicitly converted to a larger data type. But what if we want to convert a larger data type to a smaller data type?

We can also convert values from one data type to another as shown below:

( data-type ) expression;

For example, a double value 10.5 can be converted to int as shown below.

(int)10.5;

Consider an example.

class D3 {

public static void main(String[] args) {

int sum = 23;

int n = 7;

double avg;

avg = (double) sum / n;

System.out.println("Average = " + avg);

}

}

Output

Average = 3.2857142857142856

In this example, since the variable avg is declared of type double, we are converting sum/n to type double by writing (double)sum/n (since int/int gives int in Java).

**Console.** Java programs often write to the console window. We use System.out—we read input, and write output, with this stream. And this fills many program requirements.

**Console usage**. We can use println with no argument for a blank line. Different argument types, like arrays, strings, and ints are supported. The methods are versatile.

**5. Define the local and Global variables with the same name and print both variables and understand the scope of the variables?**

**Difference between Local variable and Global Variable**

Variables in any programming language have a crucial role. Variables are classified into Global variables and Local variables based on their scope. The main difference between Global and local variables is that global variables can be accessed globally in the entire program, whereas local variables can be accessed only within the function or block in which they are defined. In this topic, we will first understand what are the variables and scope, along with local variables, global variables, and then differences between both the variables.

**What is a Variable?**

A variable is a name given to a memory location to store values in a computer program. It is used to store information that can be referenced and manipulated in a program.

We can choose any name for the variable, but it must follow the programming semantics. Such as it can be, a, b, x, y, z, sub, div, total, avg, etc.

Let's say there are two values, 10 and 20, that we want to store and use in our program. For this, we need to use a variable, and we will do the below steps:

First, we will create or declare a variable with a suitable name.

Assign those values to the variables to store them.

Once these values are stores, we can use these variables with their name in our program.

Local variable vs Global Variable

As we can see in the above image, there are two memory slots, 001 and 002, and we have given names to these locations as A and B. A is containing 10, and B is containing 20.

Different programming languages have different ways to declare the variable. For example, in C language, we can declare the variable in the following manner:

Syntax: (Variable declaration syntax in C language)

datatype v1, v2, v3,....;

Example:

#include <stdio.h>

void main(){

int a;

int b;

int sum;

}

**Scope of Variable**

Each variable is defined and can be used within its scope and determines that wherein the program this variable is available to use. The scope means the lifetime of that variable. It means the variable can only be accessed or visible within its scope.

The scope of variables can be defined with their declaration, and variables are declared mainly in two ways:

Global Variable: Outside of all the functions

Local Variable: Within a function block:

**What is a Global Variable?**

Global variables are those variables which are declared outside of all the functions or block and can be accessed globally in a program.

It can be accessed by any function present in the program.

Once we declare a global variable, its value can be varied as used with different functions.

The lifetime of the global variable exists till the program executes. These variables are stored in fixed memory locations given by the compiler and do not automatically clean up.

Global variables are mostly used in programming and useful for cases where all the functions need to access the same data.

Example:

#include<stdio.h>

int a=50, b=40;

void main()

{

printf("a = %d and b=%d",a,b);

}

In the above example, a and b are the global variables.

**Advantages of Global Variable**

Global variables can be accessed by all the functions present in the program.

Only a single declaration is required.

Very useful if all the functions are accessing the same data.

**Disadvantages of Global Variable**

The value of a global variable can be changed accidently as it can be used by any function in the program.

If we use a large number of global variables, then there is a high chance of error generation in the program.

**What is a Local Variable?**

Variables that are declared within or inside a function block are known as Local variables.

These variables can only be accessed within the function in which they are declared.

The lifetime of the local variable is within its function only, which means the variable exists till the function executes. Once function execution is completed, local variables are destroyed and no longer exist outside the function.

The reason for the limited scope of local variables is that local variables are stored in the stack, which is dynamic in nature and automatically cleans up the data stored within it.

But by making the variable static with "static" keyword, we can retain the value of local variable.

Example:

#include<stdio.h>

void main()

{

int x=50, y=40;

printf("x = %d and y=%d",x, y);

}

In the above example, we have declared x and y two variables inside the main function. Hence these are local variables.

Advantages of Local Variable

The same name of a local variable can be used in different functions as it is only recognized by the function in which it is declared.

Local variables use memory only for the limited time when the function is executed; after that same memory location can be reused.

Disadvantages of Local Variables

The scope of the local variable is limited to its function only and cannot be used by other functions.

Data sharing by the local variable is not allowed.

Comparison Chart between Global Variable and Local Variable

Global Variable Local Variable

Global variables are declared outside all the function blocks. Local Variables are declared within a function block.

The scope remains throughout the program. The scope is limited and remains within the function only in which they are declared.

Any change in global variable affects the whole program, wherever it is being used. Any change in the local variable does not affect other functions of the program.

A global variable exists in the program for the entire time the program is executed. A local variable is created when the function is executed, and once the execution is finished, the variable is destroyed.

It can be accessed throughout the program by all the functions present in the program. It can only be accessed by the function statements in which it is declared and not by the other functions.

If the global variable is not initialized, it takes zero by default. If the local variable is not initialized, it takes the garbage value by default.

Global variables are stored in the data segment of memory. Local variables are stored in a stack in memory.

We cannot declare many variables with the same name. We can declare various variables with the same name but in other functions.

Examples to understand differences between Local and Global Variable

Now let's understand examples in different programming languages to better understand the difference between local and global variables.

**6. Write a function to print your name and call the function from main method?**

**Java Methods**

A method is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

A method to draw the circle

A method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

**In Java, there are two types of methods:**

**User-defined Methods:** We can create our own method based on our requirements.

**Standard Library Methods:** These are built-in methods in Java that are available to use.

**Let's first learn about user-defined methods.**

**Declaring a Java Method**

**The syntax to declare a method is:**

returnType methodName() {

// method body

}

Here,

**returnType -** It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value.

If the method does not return a value, its return type is void.

**methodName** - It is an identifier that is used to refer to the particular method in a program.

**method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }.

**For example,**

int addNumbers() {

// code

}

In the above example, the name of the method is adddNumbers(). And, the return type is int. We will learn more about return types later in this tutorial.

This is the simple syntax of declaring a method. However, the complete syntax of declaring a method is

modifier static returnType nameOfMethod (parameter1, parameter2, ...) {

// method body

}

Here,

**modifier -** It defines access types whether the method is public, private, and so on. To learn more, visit Java Access Specifier.

**static -** If we use the static keyword, it can be accessed without creating objects.

**For example,** the sqrt() method of standard Math class is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.

parameter1/parameter2 - These are values passed to a method. We can pass any number of arguments to a method.

**Calling a Method in Java**

In the above example, we have declared a method named addNumbers(). Now, to use the method, we need to call it.

Here's is how we can call the addNumbers() method.

// calls the method

addNumbers();

Call a method in Java using the name the method followed by a parenthesis

Working of Java Method Call

Example 1: Java Methods

class Main {

**// create a method**

public int addNumbers(int a, int b) {

int sum = a + b;

// return value

return sum;

}

public static void main(String[] args) {

int num1 = 25;

int num2 = 15;

**// create an object of Main**

Main obj = new Main();

// calling method

int result = obj.addNumbers(num1, num2);

System.out.println("Sum is: " + result);

}

}

**Output**

Sum is: 40

In the above example, we have created a method named addNumbers(). The method takes two parameters a and b. Notice the line,

int result = obj.addNumbers(num1, num2);

Here, we have called the method by passing two arguments num1 and num2. Since the method is returning some value, we have stored the value in the result variable.

Note: The method is not static. Hence, we are calling the method using the object of the class**.**

**Java Method Return Type**

A Java method may or may not return a value to the function call. We use the return statement to return any value. For example,

int addNumbers() {

...

return sum;

}

Here, we are returning the variable sum. Since the return type of the function is int. The sum variable should be of int type. Otherwise, it will generate an error.

**Example 2: Method Return Type**

class Main {

// create a method

public static int square(int num) {

// return statement

return num \* num;

}

public static void main(String[] args) {

int result;

// call the method

// store returned value to result

result = square(10);

System.out.println("Squared value of 10 is: " + result);

}

}

Output:

Squared value of 10 is: 100

In the above program, we have created a method named square(). The method takes a number as its parameter and returns the square of the number.

Here, we have mentioned the return type of the method as int. Hence, the method should always return an integer value.

Java method returns a value to the method call

Representation of the Java method returning a value

Note: If the method does not return any value, we use the void keyword as the return type of the method. For example,

public void square(int a) {

int square = a \* a;

System.out.println("Square is: " + a);

}

**Method Parameters in Java**

A method parameter is a value accepted by the method. As mentioned earlier, a method can also have any number of parameters. For example,

// method with two parameters

int addNumbers(int a, int b) {

// code

}

// method with no parameter

int addNumbers(){

// code

}

If a method is created with parameters, we need to pass the corresponding values while calling the method. For example,

// calling the method with two parameters

addNumbers(25, 15);

// calling the method with no parameters

addNumbers()

**Example 3: Method Parameters**

class Main {

// method with no parameter

public void display1() {

System.out.println("Method without parameter");

}

// method with single parameter

public void display2(int a) {

System.out.println("Method with a single parameter: " + a);

}

public static void main(String[] args) {

// create an object of Main

Main obj = new Main();

// calling method with no parameter

obj.display1();

// calling method with the single parameter

obj.display2(24);

}

}

**Output**

Method without parameter

Method with a single parameter: 24

Here, the parameter of the method is int. Hence, if we pass any other data type instead of int, the compiler will throw an error. It is because Java is a strongly typed language.

Note: The argument 24 passed to the display2() method during the method call is called the actual argument.

The parameter num accepted by the method definition is known as a formal argument. We need to specify the type of formal arguments. And, the type of actual arguments and formal arguments should always match.

**Standard Library Methods**

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (\*.jar) file with JVM and JRE.

For example,

print() is a method of java.io.PrintSteam. The print("...") method prints the string inside quotation marks.

sqrt() is a method of Math class. It returns the square root of a number.

Here's a working example:

**Example 4: Java Standard Library Method**

public class Main {

public static void main(String[] args) {

// using the sqrt() method

System.out.print("Square root of 4 is: " + Math.sqrt(4));

}

}

Output:

Square root of 4 is: 2.0

To learn more about standard library methods, visit Java Library Methods.

**What are the advantages of using methods?**

1. The main advantage is code reusability. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".

Example 5: Java Method for Code Reusability

public class Main {

// method defined

private static int getSquare(int x){

return x \* x;

}

public static void main(String[] args) {

for (int i = 1; i <= 5; i++) {

// method call

int result = getSquare(i);

System.out.println("Square of " + i + " is: " + result);

}

}

}

Output:

Square of 1 is: 1

Square of 2 is: 4

Square of 3 is: 9

Square of 4 is: 16

Square of 5 is: 25

In the above program, we have created the method named getSquare() to calculate the square of a number. Here, the method is used to calculate the square of numbers less than 6.

Hence, the same method is used again and again.

2. Methods make code more readable and easier to debug. Here, the getSquare() method keeps the code to compute the square in a block. Hence, makes it more readable.

1. **Operators**
2. **Write a function for arithmetic operators(+,-,\*,/)**

Java too provides many types of operators which can be used according to the need to perform various calculation and functions be it logical, arithmetic, relational etc. They are classified based on the functionality they provide. Here are a few types:

Arithmetic Operators

Unary Operators

Assignment Operator

Relational Operators

Logical Operators

Ternary Operator

Bitwise Operators

Shift Operators

**Arithmetic Operators**

These operators involve the mathematical operators that can be used to perform various simple or advance 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 respectively. Let’s look at the various operators that Java has to provide under the arithmetic operators.

Addition(+): This operator is a binary operator and is used to add two operands.

Syntax:

num1 + num2

Subtraction(-): This operator is a binary operator and is used to subtract two operands.

Syntax:

num1 - num2

Multiplication(\*): This operator is a binary operator and is used to multiply two operands.

Syntax:

num1 \* num2

Division(/): This is a binary operator that is used to divide the first operand(dividend) by the second operand(divisor) and give the quotient as result.

Syntax:

num1 / num2

Modulus(%): This is a binary operator that is used to return the remainder when the first operand(dividend) is divided by the second operand(divisor).

Syntax:

num1 % num2

Increment(++): This is a unary operator that acts on one operand, unlike the previous operations. It is used to increment the value of an integer. It can be used in two ways:

Post-increment operator: When placed after the variable name, the value of the operand is incremented but the previous value is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

Syntax:

num++

Pre-increment operator: When placed before the variable name, the operand’s value is incremented instantly.

Syntax:

++num

Decrement(–): This is also a unary operator that acts on one operand. It is used to decrement the value of an integer. It can be used in two ways:

Post-decrement operator: When placed after the variable name, the value of the operand is decremented but the previous values is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

Syntax:

num--

Pre-decrement operator: When placed before the variable name, the operand’s value is decremented instantly.

Syntax:

--num

**Unary Operators**

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by 1. That is, ++5 will return 6.

Different types of unary operators are:

Operator Meaning

+ Unary plus: not necessary to use since numbers are positive without using it

- Unary minus: inverts the sign of an expression

++ Increment operator: increments value by 1

-- Decrement operator: decrements value by 1

! Logical complement operator: inverts the value of a Boolean

**Assignment Operator**

Assignment operators are used in Java to assign values to variables. For example,

int age;

age = 5;

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, 5 is assigned to the variable age.

Let's see some more assignment operators available in Java.

Operator Example Equivalent to

= a = b; a = b;

+= a += b; a = a + b;

-= a -= b; a = a - b;

\*= a \*= b; a = a \* b;

/= a /= b; a = a / b;

%= a %= b; a = a % b;

**Relational Operators**

Relational operators are used to check the relationship between two operands. For example,

// check is a is less than b

a < b;

Here, > operator is the relational operator. It checks if a is less than b or not.

It returns either true or false.

**Logical Operators**

Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator Example Meaning

&& (Logical AND) expression1 && expression2 true only if both expression1 and expression2 are true

|| (Logical OR) expression1 || expression2 true if either expression1 or expression2 is true

! (Logical NOT) !expression true if expression is false and vice versa

**Ternary Operator**

**Ternary Operator in Java**

**A ternary operator evaluates the test condition and executes a block of code based on the result of the condition. if condition is true , expression1 is executed. And, if condition is false , expression2 is executed.**

The ternary operator is a part of Java’s conditional statements. As the name ternary suggests, it is the only operator in Java consisting of three operands.

The ternary operator can be thought of as a simplified version of the if-else statement with a value to be returned.

**The ternary operator take three arguments**:

The first is a comparison argument.

The second is the result upon a true comparison.

The third is the result upon a false comparison.

**Syntax**

The three operands in a ternary operator include:

A boolean expression that evaluates to either true or false.

A value to be assigned if the expression is evaluated to true.

A value to be assigned if the expression is evaluated to false.

1 variable var = (booleanExpression) ? value1 if true : value2 if false

The variable var on the left-hand side of the = (assignment) operator will be assigned:

value1 if the booleanExpression evaluates to true

value2 if the booleanExpression evaluates to false

**Bitwise Operators**

In Java, an operator is a symbol that performs the specified operations. In this section, we will discuss only the bitwise operator and its types with proper examples.

Types of Bitwise Operator

There are six types of the bitwise operator in Java:

Bitwise AND

Bitwise exclusive OR

Bitwise inclusive OR

Bitwise Compliment

Bit Shift Operators

Operators Symbol Uses

Bitwise AND & op1 & op2

Bitwise exclusive OR ^ op1 ^ op2

Bitwise inclusive OR | op1 | op2

Bitwise Compliment ~ ~ op

Bitwise left shift << op1 << op2

Bitwise right shift >> op1 >> op2

Unsigned Right Shift Operator >>> op >>> number of places to shift

Bitwise Operator in Java

In Java, an operator is a symbol that performs the specified operations. In this section, we will discuss only the bitwise operator and its types with proper examples.

Types of Bitwise Operator

There are six types of the bitwise operator in Java:

Bitwise AND

Bitwise exclusive OR

Bitwise inclusive OR

Bitwise Compliment

Bit Shift Operators

Operators Symbol Uses

Bitwise AND & op1 & op2

Bitwise exclusive OR ^ op1 ^ op2

Bitwise inclusive OR | op1 | op2

Bitwise Compliment ~ ~ op

Bitwise left shift << op1 << op2

Bitwise right shift >> op1 >> op2

Unsigned Right Shift Operator >>> op >>> number of places to shift

**Bitwise AND (&)**

It is a binary operator denoted by the symbol &. It returns 1 if and only if both bits are 1, else returns 0.

**Bitwise exclusive OR (^)**

It is a binary operator denoted by the symbol ^ (pronounced as caret). It returns 0 if both bits are the same, else returns 1.

**Bitwise inclusive OR (|)**

It is a binary operator denoted by the symbol | (pronounced as a pipe). It returns 1 if either of the bit is 1, else returns 0.

**Bitwise Complement (~)**

It is a unary operator denoted by the symbol ~ (pronounced as the tilde). It returns the inverse or complement of the bit. It makes every 0 a 1 and every 1 a 0.

**Bit Shift Operators**

Shift operator is used in shifting the bits either right or left. We can use shift operators if we divide or multiply any number by 2. The general format to shift the bit is as follows:

variable << or >> number of places to shift;

**Signed Right Shift Operator (>>)**

The signed right shift operator shifts a bit pattern of a number towards the right with a specified number of positions and fills 0. The operator is denoted by the symbol >>. It also preserves the leftmost bit (sign bit). If 0 is presented at the leftmost bit, it means the number is positive. If 1 is presented at the leftmost bit, it means the number is negative.

**Signed Left Shift Operator (<<)**

The signed left shift operator (<<) shifts a bit pattern to the left. It is represented by the symbol <<. It also preserves the leftmost bit (sign bit). It does not preserve the sign bit.

**Unsigned Right Shift Operator (>>>)**

It shifts a zero at the leftmost position and fills 0. It is denoted by the symbol >>>. Note that the leftmost position after >> depends on the sign bit. It does not preserve the sign bit.

**2. Write a method for increment and decrement operators(++, --)** Incrementing and decrementing are such common operations that Java provides special operators for them. The ++ operator adds one to the current value of an int or char. -- subtracts one. Neither operator works on doubles, booleans or Strings.

Increment and decrement operators each have two forms: pre and post. In above example we have used the post form of increment and decrement operator

Both the pre- and post-increment operators increment the value of the variable by 1. Similarly, the pre- and post-decrement operators decrement the value of the variable by 1. The difference becomes apparent when the variable using these operators is employed in an expression.

**Operators**

‘Logical AND’ Operator(&&): This operator returns true when both the conditions under consideration are satisfied or are true. If even one of the two yields false, the operator results false. For example, cond1 && cond2 returns true when both cond1 and cond2 are true (i.e. non-zero).

Syntax:

condition1 && condition2

'Logical OR' Operator(||): This operator returns true when one of the two conditions under consideration are satisfied or are true. If even one of the two yields true, the operator results true. To make the result false, both the constraints need to return false.

Syntax:

condition1 || condition2

'Logical NOT' Operator(!): Unlike the previous two, this is a unary operator and returns true when the condition under consideration is not satisfied or is a false condition. Basically, if the condition is false, the operation returns true and when the condition is true, the operation returns false.

Syntax:

!(condition)

**Armstrong Number in Java**

Let's write a java program to check whether the given number is armstrong number or not.

Armstrong Number in Java: A positive number is called armstrong number if it is equal to the sum of cubes of its digits for example 0, 1, 153, 370, 371, 407 etc.

Let's try to understand why 153 is an Armstrong number.

153 = (1\*1\*1)+(5\*5\*5)+(3\*3\*3)

where:

(1\*1\*1)=1

(5\*5\*5)=125

(3\*3\*3)=27

So:

1+125+27=153

Let's try to understand why 371 is an Armstrong number.

371 = (3\*3\*3)+(7\*7\*7)+(1\*1\*1)

where:

(3\*3\*3)=27

(7\*7\*7)=343

(1\*1\*1)=1

So:

27+343+1=371

**Loops in Java**

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

for loop

while loop

do-while loop

Java For Loop vs While Loop vs Do While Loop

Comparison for loop while loop do while loop

Introduction The Java for loop is a control flow statement that iterates a part of the programs multiple times.

The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.

The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.

When to use

If the number of iteration is fixed, it is recommended to use for loop.

If the number of iteration is not fixed, it is recommended to use while loop.

If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

Syntax

for(init;condition;incr/decr){

// code to be executed

}

while(condition){

//code to be executed

}

do{

//code to be executed

}while(condition);

Example

**//for loop**

for(int i=1;i<=10;i++){

System.out.println(i);

}

**//while loop**

int i=1;

while(i<=10){

System.out.println(i);

i++;

}

**//do-while loop**

int i=1;

do{

System.out.println(i);

i++;

}while(i<=10);

Syntax for infinitive loop

for(;;){

//code to be executed

}

while(true){

//code to be executed

}

do{

//code to be executed

}while(true);

**4. Arrays**

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

advantages

Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.

Random access: We can get any data located at an index position.

Disadvantages

Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

ADVERTISEMENT

Single Dimensional Array

Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

Instantiation of an Array in Java

arrayRefVar=new datatype[size];

Example of Java Array

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

int a[]={33,3,4,5};//declaration, instantiation and initialization

For-each Loop for Java Array

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

for(data\_type variable:array){

//body of the loop

}

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

//Java Program to demonstrate the case of

//ArrayIndexOutOfBoundsException in a Java Array.

public class TestArrayException{

public static void main(String args[]){

int arr[]={50,60,70,80};

for(int i=0;i<=arr.length;i++){

System.out.println(arr[i]);

}

}}

Test it Now

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

int[][] arr=new int[3][3];//3 row and 3 column

Example to initialize Multidimensional Array in Java

arr[0][0]=1;

arr[0][1]=2;

arr[0][2]=3;

arr[1][0]=4;

arr[1][1]=5;

arr[1][2]=6;

arr[2][0]=7;

arr[2][1]=8;

arr[2][2]=9;

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.

1. **Write a function to add integer values of an array**

// Function to add x in arr

public static int[] addX(int n, int arr[], int x)

{

int i;

1. **Write a function to calculate the average value of an array of integers**

import java.util.Scanner;

public class Sum\_Average

{

public static void main(String[] args)

{

int n, sum = 0;

float average;

Scanner s = new Scanner(System.in);

System.out.print("Enter no. of elements you want in array:");

n = s.nextInt();

int a[] = new int[n];

System.out.println("Enter all the elements:");

for(int i = 0; i < n ; i++)

{

a[i] = s.nextInt();

sum = sum + a[i];

}

System.out.println("Sum:"+sum);

average = (float)sum / n;

System.out.println("Average:"+average);

}

}

**Output:**

$ javac Sum\_Average.java

$ java Sum\_Average

Enter no. of elements you want in array:5

Enter all the elements:

4

7

6

9

3

Sum:29

Average:5.8

**4.Write a function to test if array contains a specific value**

public static boolean contains(int[] arr, int item) {

for (int n : arr) {

if (item == n) {

return true;

}

}

return false;

}

public static void main(String[] args) {

int[] my\_array1 = {

1789, 2035, 1899, 1456, 2013,

1458, 2458, 1254, 1472, 2365,

1456, 2265, 1457, 2456};

System.out.println(contains(my\_array1, 2013));

System.out.println(contains(my\_array1, 2015));

}

}

Output

$javac Exercise5.java

$java -Xmx128M -Xms16M Exercise5

true

false

**5. Write a function to remove a specific element from an array**

import java.util.Arrays;

public class Exercise7 {

public static void main(String[] args) {

int[] my\_array = {25, 14, 56, 15, 36, 56, 77, 18, 29, 49};

System.out.println("Original Array : "+Arrays.toString(my\_array));

// Remove the second element (index->1, value->14) of the array

int removeIndex = 1;

for(int i = removeIndex; i < my\_array.length -1; i++){

my\_array[i] = my\_array[i + 1];

}

// We cannot alter the size of an array , after the removal, the last and second last element in the array will exist twice

System.out.println("After removing the second element: "+Arrays.toString(my\_array));

}

}

Output

Original Array : [25, 14, 56, 15, 36, 56, 77, 18, 29, 49]

After removing the second element: [25, 56, 15, 36, 56, 77, 18, 29, 49, 49]

**6. Write a function to copy an array to another array**

public class CopyArray {

public static void main(String[] args) {

//Initialize array

int [] arr1 = new int [] {1, 2, 3, 4, 5};

//Create another array arr2 with size of arr1

int arr2[] = new int[arr1.length];

//Copying all elements of one array into another

for (int i = 0; i < arr1.length; i++) {

arr2[i] = arr1[i];

}

//Displaying elements of array arr1

System.out.println("Elements of original array: ");

for (int i = 0; i < arr1.length; i++) {

System.out.print(arr1[i] + " ");

}

System.out.println();

//Displaying elements of array arr2

System.out.println("Elements of new array: ");

for (int i = 0; i < arr2.length; i++) {

System.out.print(arr2[i] + " ");

}

}

}

Output

Elements of original array

1 2 3 4 5

Elements of new array:

1 2 3 4 5

**7. Write a function to insert an element at a specific position in the array**

import java.util.Arrays;

public class Exercise9 {

public static void main(String[] args) {

int[] my\_array = {25, 14, 56, 15, 36, 56, 77, 18, 29, 49};

// Insert an element in 3rd position of the array (index->2, value->5)

int Index\_position = 2;

int newValue = 5;

System.out.println("Original Array : "+Arrays.toString(my\_array));

for(int i=my\_array.length-1; i > Index\_position; i--){

my\_array[i] = my\_array[i-1];

}

my\_array[Index\_position] = newValue;

System.out.println("New Array: "+Arrays.toString(my\_array));

}

}

Output

Original Array : [25, 14, 56, 15, 36, 56, 77, 18, 29, 49]

New Array: [25, 14, 5, 56, 15, 36, 56, 77, 18, 29]

**8. Write a function to find the minimum and maximum value of an array**

public class MinAndMax {

public int max(int [] array) {

int max = 0;

for(int i=0; i<array.length; i++ ) {

if(array[i]>max) {

max = array[i];

}

}

return max;

}

public int min(int [] array) {

int min = array[0];

for(int i=0; i<array.length; i++ ) {

if(array[i]<min) {

min = array[i];

}

}

return min;

}

public static void main(String args[]) {

int[] myArray = {23, 92, 56, 39, 93};

MinAndMax m = new MinAndMax();

System.out.println("Maximum value in the array is::"+m.max(myArray));

System.out.println("Minimum value in the array is::"+m.min(myArray));

}

}

Output

Maximum value in the array is::93

Minimum value in the array is::23

**9. Write a function to reverse an array of integer values**

import java.util.Arrays;

public class Exercise11 {

public static void main(String[] args){

int[] my\_array1 = {

1789, 2035, 1899, 1456, 2013,

1458, 2458, 1254, 1472, 2365,

1456, 2165, 1457, 2456};

System.out.println("Original array : "+Arrays.toString(my\_array1));

for(int i = 0; i < my\_array1.length / 2; i++)

{

int temp = my\_array1[i];

my\_array1[i] = my\_array1[my\_array1.length - i - 1];

my\_array1[my\_array1.length - i - 1] = temp;

}

System.out.println("Reverse array : "+Arrays.toString(my\_array1));

}

}

Output

Original array : [1789, 2035, 1899, 1456, 2013, 1458, 2458, 1254, 1472, 2365, 1456, 2165, 1457, 2456]

Reverse array : [2456, 1457, 2165, 1456, 2365, 1472, 1254, 2458, 1458, 2013, 1456, 1899, 2035, 1789]

**10. Write a function to find the duplicate values of an array**

import java.util.Arrays;

public class Exercise12 {

public static void main(String[] args)

{

int[] my\_array = {1, 2, 5, 5, 6, 6, 7, 2};

for (int i = 0; i < my\_array.length-1; i++)

{

for (int j = i+1; j < my\_array.length; j++)

{

if ((my\_array[i] == my\_array[j]) && (i != j))

{

System.out.println("Duplicate Element : "+my\_array[j]);

}

}

}

}

}

Output

Duplicate Element : 2

Duplicate Element : 5

Duplicate Element : 6

**11. Write a program to find the common values between two arrays**

Given two arrays and our task is to find their common elements.

Examples:

Input: Array1 = ["Article", "for", "Geeks", "for", "Geeks"],

Array2 = ["Article", "Geeks", "Geeks"]

Output: [Article,Geeks]

Input: Array1 = ["a", "b", "c", "d", "e", "f"],

Array2 = ["b", "d", "e", "h", "g", "c"]

Output: [b, c, d, e]

Using Iterative Methods

Approach:

Get the two java Arrays.

Iterate through each and every element of the arrays one by one and check whether they are common in both.

Add each common element in the set for unique entries.

Java

// Java Program to find common elements

// in two Arrays

// Using iterative method

import java.io.\*;

import java.util.\*;

class GFG {

private static void FindCommonElemet(String[] arr1,

String[] arr2)

{

Set<String> set = new HashSet<>();

for (int i = 0; i < arr1.length; i++) {

for (int j = 0; j < arr2.length; j++) {

if (arr1[i] == arr2[j]) {

// add common elements

set.add(arr1[i]);

break;

}

}

}

for (String i : set) {

System.out.print(i + " ");

}

}

// main method

public static void main(String[] args)

{

// create Array 1

String[] arr1

= { "Article", "in", "Geeks", "for", "Geeks" };

// create Array 2

String[] arr2 = { "Geeks", "for", "Geeks" };

// print Array 1

System.out.println("Array 1: "

+ Arrays.toString(arr1));

// print Array 2

System.out.println("Array 2: "

+ Arrays.toString(arr2));

System.out.print("Common Elements: ");

// Find the common elements

FindCommonElemet(arr1, arr2);

}

}

Output

Array 1: [Article, in, Geeks, for, Geeks]

Array 2: [Geeks, for, Geeks]

Common Elements: Geeks for

**12. Write a method to remove duplicate elements from an array**

There are two ways to remove duplicates from array java: first using temporary array and second using seperate index.

To remove duplicates from array in java, the array should be in sorted order.

remove duplicate element in an array in java

If array is not sorted then sort given array using Arrays.sort() method.

public class RemoveDuplicateInArrayExample{

public static int removeDuplicateElements(int arr[], int n){

if (n==0 || n==1){

return n;

}

int[] temp = new int[n];

int j = 0;

for (int i=0; i<n-1; i++){

if (arr[i] != arr[i+1]){

temp[j++] = arr[i];

}

}

temp[j++] = arr[n-1];

// Changing original array

for (int i=0; i<j; i++){

arr[i] = temp[i];

}

return j;

}

public static void main (String[] args) {

int arr[] = {10,20,20,30,30,40,50,50};

int length = arr.length;

length = removeDuplicateElements(arr, length);

//printing array elements

for (int i=0; i<length; i++)

System.out.print(arr[i]+" ");

}

}

Output:

10 20 30 40 50

**13. Write a method to find the second largest number in an array**

public class SecondLargestInArrayExample{

public static int getSecondLargest(int[] a, int total){

int temp;

for (int i = 0; i < total; i++)

{

for (int j = i + 1; j < total; j++)

{

if (a[i] > a[j])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

return a[total-2];

}

public static void main(String args[]){

int a[]={1,2,5,6,3,2};

int b[]={44,66,99,77,33,22,55};

System.out.println("Second Largest: "+getSecondLargest(a,6));

System.out.println("Second Largest: "+getSecondLargest(b,7));

}}

Output:

Second Largest: 5

Second Largest: 77

**15. Write a method to find number of even number and odd numbers in an array**We can print odd and even numbers from an array in java by getting remainder of each element and checking if it is divided by 2 or not. If it is divided by 2, it is even number otherwise it is odd number.

public class OddEvenInArrayExample{

public static void main(String args[]){

int a[]={1,2,5,6,3,2};

System.out.println("Odd Numbers:");

for(int i=0;i<a.length;i++){

if(a[i]%2!=0){

System.out.println(a[i]);

}

}

System.out.println("Even Numbers:");

for(int i=0;i<a.length;i++){

if(a[i]%2==0){

System.out.println(a[i]);

}

}

}}

Output

$javac OddEvenInArrayExample.java

$java -Xmx128M -Xms16M OddEvenInArrayExample

Odd Numbers:

1

5

3

Even Numbers:

2

6

2

**16. Write a function to get the difference of largest and smallest value**

The length of the array must be 1 and above.

import java.util.Arrays;

public class Exercise28 {

public static void main(String[] args)

{

int[] array\_nums = {5, 7, 2, 4, 9};

System.out.println("Original Array: "+Arrays.toString(array\_nums));

int max\_val = array\_nums[0];

int min = array\_nums[0];

for(int i = 1; i < array\_nums.length; i++)

{

if(array\_nums[i] > max\_val)

max\_val = array\_nums[i];

else if(array\_nums[i] < min)

min = array\_nums[i];

}

System.out.println("Difference between the largest and smallest values of the said array: "+(max\_val-min));

}

}

Output

Original Array: [5, 7, 2, 4, 9]

Difference between the largest and smallest values of the said array: 7

**17. Write a method to verify if the array contains two specified elements(12,23)**

import java.util.\*;

import java.io.\*;

public class Exercise32 {

public static void main(String[] args)

{

int[] array\_nums = {77, 77, 65, 65, 65, 77};

System.out.println("Original Array: "+Arrays.toString(array\_nums));

int num1 = 77;

int num2 = 65;

System.out.println("Result: "+result(array\_nums, num1, num2));

}

public static boolean result(int[] array\_nums, int num1, int num2) {

for (int number : array\_nums) {

boolean r = number != num1 && number != num2;

if (r) {

return false;

}

}

return true;

}

}

Output

Original Array: [77, 77, 65, 65, 65, 77]

Result: true

**19. Write a function to find the missing number of sorted array of 1 to 100**

Given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

Examples:

Input : arr[] = [1, 2, 3, 4, 6, 7, 8]

Output : 5

Input : arr[] = [1, 2, 3, 4, 5, 6, 8, 9]

Output : 7

**5. Static**

**71. Write a class with 2 static variables, 2 Instance variables, 2 static methods, 2 instance**

**methods and a main method.**

**Java Static Variables**

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

Few Important Points:

Static variables are also known as Class Variables.

Unlike non-static variables, such variables can be accessed directly in static and non-static methods.

**What is instance variable in Java?**

Instance variables in Java are non-static variables which are defined in a class outside any method, constructor or a block. Each instantiated object of the class has a separate copy or instance of that variable. An instance variable belongs to a class.

You must be wondering about what exactly is an Instance? Let me help you by simplifying it.

When you create a new object of the class you create an instance. Consider, if you have a STUDENT class, then

class Student

{

String studentName;

int studentScore;

}

And if you create two STUDENT objects like,

Student student1 = new Student();

Student student2 = new Student();

Then two instances of the class Student will be created.

Features of an instance variable?

The life of an instance variable depends on the life of an Object, i.e., when the object is created, an instance variable also gets created and the same happens when an object is destroyed.

Instance Variable can be used only by creating objects

Every object will have its own copy of Instance variables

Initialization of instance variable is not compulsory. The default value is zero

The declaration is done in a class outside any method, constructor or block

Instance variables are used when the variable has to be known to different methods in a class

Access modifiers can be assigned to instance variables

**Java Static Methods**

Static Methods can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects.

Static methods can be accessed directly in static and non-static methods.

Syntax:

Static keyword followed by return type, followed by method name.

**Instance methods**

Non-static methods, which we will refer to as instance methods or object methods are called using an object and therefore have access to an object’s instance variables. Note the method header will not include the keyword static.

There are three steps to creating and calling an instance method:

Object of the Class: Declare an object of your class in the main method or from outside the class.

// Step 1: declare an object in main or from outside the class

Classname objectName = new Classname();

Method Definition: write the method’s header and body code like below:

// Step 3: Define the method in the class

// method header

public void methodName()

{

// method body for the code

}

Method Call: whenever you want to use the method, call objectName.methodName();

// Step 2: call the object's method

objectName.methodName(); //Step 2

How is this different than calling a static method? Notice the object reference and dot notation before the method name. Since the method may set and get instance variables, you must call the method on an actual instance of the class.

**main method**

Different ways of writing main() method are:

static public void main(String []x)

static public void main(String... args)

A Java application is a public Java class with a main() method. The main() method is the entry point into the application. The signature of the method is always: public static void main(String[] args) Command-line arguments are passed through the args parameter, which is an array of String s.

**Public- it is access specifier from anywhere we can access it Static- it is access modifier we can call the methods directly by class name without creating its objects Void- it is the return type Main- it is a method name String[]args- in java we accept only the string type of argument and store it.**

**Print instance variables in static methods**

We cannot directly access the instance variables within a static method because a static method can only access static variables or static methods.

An instance variable, as the name suggests is tied to an instance of a class. Therefore, accessing it directly from a static method, which is not tied to any specific instance doesn't make sense. Therefore, to access an instance variable, we must have an instance of the class from which we access the instance variable.

Example:

public class Test {

public int instanceVariable = 10;

public static void main(String args[]) {

Test test = new Test();

System.out.println(test.instanceVariable);

}

}

Output:

10

**Print static variables in Instance methods**

Instance method can access the instance methods and instance variables directly.

Instance method can access static variables and static methods directly.

Static methods can access the static variables and static methods directly.

Static methods can’t access instance methods and instance variables directly. They must use reference to object. And static method can’t use this keyword as there is no instance for ‘this’ to refer to.

**Call instance methods in static methods**

Instance method

Instance method requires an object of its class to be created before it can be called. To invoke an instance method, we have to create an Object of the class within which it defined.

Syntax of Instance method

public void geek(String name)

{

// code to be executed...

}

// Return type can be int, float String or user-defined data type.

Static method

Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the object of that class.

**Call static methods in instance methods**

Static methods can be called freely, but instance methods can only be called if you have an instance of the class. The static method needs to either get an instance from somewhere, or create one itself.

**Print all the static, instance variables in main method**

**Call static methods and instance methods in main method**

**6. Strings**

**Different ways creating a string**

You can create a String by −

Step 1 − Assigning a string value wrapped in " " to a String type variable.

String message = "Hello Welcome to Tutorialspoint";

Step 2 − Creating an object of the String class using the new keyword by passing the string value as a parameter of its constructor.

String message = new String ("Hello Welcome to Tutorialspoint");

Step 3 − Passing a character array to the String constructor.

char arr[] = {'H','e','l','l','o'};

String message = new String(arr);

**Concatenating two strings using + operator**

class TestStringConcatenation1{

public static void main(String args[]){

String s="Sachin"+" Tendulkar";

System.out.println(s);//Sachin Tendulkar

}

}

Output:Sachin Tendulkar

**Finding the length of the string**

The length of a string is referred to as the total number of characters it contains.

The length() method

To calculate the length of a string in Java, you can use an inbuilt length() method of the Java string class.

In Java, strings are objects created using the string class and the length() method is a public member method of this class. So, any variable of type string can access this method using the . (dot) operator.

The length() method counts the total number of characters in a String.

Method signature

The signature of the length() method is as follows:

1 public int length()

The return type of the length() method is int.

Example

Let’s calculate & printout the length of a string using the length() method.

class CalcLength {

public static void main( String args[] ) {

String name = "educative"; //Initilizing a String Object name

int length = name.length(); //Calling the inbuilt lenght() method

System.out.println("The length of the String \""+name+"\" is: " +length); }

}

Run

The length of the string name is 9:

educative

**Extract a string using Substring**

You can extract a substring from a String using the substring() method of the String class to this method you need to pass the start and end indexes of the required substring.

Example

Live Demo

public class Substring {

public static void main(String args[]) {

String str = "Welcome to Tutorialspoint";

String sub = str.substring(10, 25);

System.out.println(sub);

}

}

Output

Tutorialspoint

**Searching in strings using indexOf()**

indexOf()- This method searches forward from the beginning of the string and returns the index within this string of the first occurrence of the specified character/substring. If a character or substring is not found indexOf() returns -1.

Search a string for the first occurrence of "planet":

String myStr = "Hello planet earth, you are a great planet.";

System.out.println(myStr.indexOf("planet"));

The indexOf() method returns the position of the first occurrence of specified character(s) in a string.

Tip: Use the lastIndexOf method to return the position of the last occurrence of specified character(s) in a string.

Syntax

There are 4 indexOf() methods:

public int indexOf(String str)

public int indexOf(String str, int fromIndex)

public int indexOf(int char)

public int indexOf(int char, int fromIndex)

Parameter Values

Parameter Description

Str A String value, representing the string to search for

fromIndex An int value, representing the index position to start the search from

char An int value, representing a single character, e.g 'A', or a Unicode value

Technical Details

Returns: An int value, representing the index of the first occurrence of the character in the string, or -1 if it never occurs

Example

Find the first occurrence of the letter "e" in a string, starting the search at position 5:

public class Main {

public static void main(String[] args) {

String myStr = "Hello planet earth, you are a great planet.";

System.out.println(myStr.indexOf("e", 5));

}

}

**Matching a String Against a Regular Expression With matches()**

There are three variants of matches() method.

1. String matches() : This method tells whether or not this string matches the given regular expression. An invocation of this method of the form str.matches(regex) yields exactly the same result as the expression Pattern.matches(regex, str).

Syntax:

public boolean matches(String regex)

Parameters

regex : the regular expression to which this string is to be matched.

Return Value

This method returns true if, and only if, this string matches the given regular expression.

2. String regionMatches() (with ignoreCase) :This method has two variants which can be used to test if two string regions are equal.

Syntax

public boolean regionMatches(boolean ignoreCase,

int str\_strt,

String other,

int other\_strt,

int len)

3. String regionMatches() :This method has two variants which can be used to test if two string regions are equal.

Syntax

public boolean regionMatches(int str\_strt,

String other,

int other\_strt,

int len)

**Comparing strings using the methods equals()**

Compare strings to find out if they are equal:

String myStr1 = "Hello";

String myStr2 = "Hello";

String myStr3 = "Another String";

System.out.println(myStr1.equals(myStr2)); // Returns true because they are equal

System.out.println(myStr1.equals(myStr3)); // false

**Definition and Usage**

The equals() method compares two strings, and returns true if the strings are equal, and false if not.

Tip: Use the compareTo() method to compare two strings lexicographically.

**Syntax**

public boolean equals(Object anotherObject)

**Parameter Values**

**Parameter Description**

AnotherObject An Object, representing the other string to be compared

**Technical Details**

**Returns:** A boolean value:

true - if the strings are equal

false - if the strings are not equal

**Overrides:** equals in class Object

**equalsIgnoreCase(), startsWith(), endsWith() and compareTo()**

**String comparison using equals() and equalsIgnoreCase() methods**

**boolean equals(Object anObject)-** This method is used to compare the content of two strings. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

**boolean equalsIgnoreCase(String anotherString)-** Comparison of strings using equals() method is case sensitive, if you want case considerations to be ignored then use equalsIgnoreCase method. When two strings are compared using this method they are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

String comparison using compareTo() and compareToIgnoreCase() methods

int compareTo(String anotherString)- Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.

int compareToIgnoreCase(String str)- Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.

In lexicographical comparison if two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both.

Compare String portions using startsWith() and endsWith() methods

boolean startsWith(String str)– Tests if this string starts with the passed argument. Returns true if substring matches at the start, false otherwise.

boolean startsWith(String str, int toffset)– Tests if the substring of this string beginning at the specified index starts with the passed argument. Returns true if substring matches at the start, false otherwise.

boolean endsWith(String str)– Tests if this string ends with the passed argument. Returns true if substring matches at the end, false otherwise.

**Trimming strings with trim()**

The java string trim() method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

The string trim() method doesn't omits middle spaces.

Internal implementation

public String trim() {

int len = value.length;

int st = 0;

char[] val = value; /\* avoid getfield opcode \*/

while ((st < len) && (val[st] <= ' ')) {

st++;

}

while ((st < len) && (val[len - 1] <= ' ')) {

len--;

}

return ((st > 0) || (len < value.length)) ? substring(st, len) : this;

}

**Signature**

The signature or syntax of string trim method is given below:

public String trim()

**Returns**

string with omitted leading and trailing spaces

**Java String trim() method example**

public class StringTrimExample{

public static void main(String args[]){

String s1=" hello string ";

System.out.println(s1+"javatpoint");//without trim()

System.out.println(s1.trim()+"javatpoint");//with trim()

}}

Test it Now

hello string javatpoint

hello stringjavatpoint

**Replacing characters in strings with replace()**

The java string replace() method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Since JDK 1.5, a new replace() method is introduced, allowing you to replace a sequence of char values.

**Signature**

There are two type of replace methods in java string.

public String replace(char oldChar, char newChar)

and

public String replace(CharSequence target, CharSequence replacement)

**Parameters**

oldChar : old character

newChar : new character

target : target sequence of characters

replacement : replacement sequence of characters

**Returns**

replaced string

**Java String replace(char old, char new) method example**

public class ReplaceExample1{

public static void main(String args[]){

String s1="javatpoint is a very good website";

String replaceString=s1.replace('a','e');//replaces all occurrences of 'a' to 'e'

System.out.println(replaceString);

}}

Test it Now

jevetpoint is e very good website

**Java String replace(CharSequence target, CharSequence replacement) method example**

public class ReplaceExample2{

public static void main(String args[]){

String s1="my name is khan my name is java";

String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"

System.out.println(replaceString);

}}

Test it Now

my name was khan my name was java

**Splitting strings with split()**

The java string split() method splits this string against given regular expression and returns a char array.

**Signature**

There are two signature for split() method in java string.

public String split(String regex)

and,

public String split(String regex, int limit)

Parameter

regex : regular expression to be applied on string.

limit : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

**Returns**

array of strings

**Throws**

PatternSyntaxException if pattern for regular expression is invalid

Since

1.4

**Java String split() method example**

The given example returns total number of words in a string excluding space only. It also includes special characters.

public class SplitExample{

public static void main(String args[]){

String s1="java string split method by javatpoint";

String[] words=s1.split("\\s");//splits the string based on whitespace

//using java foreach loop to print elements of string array

for(String w:words){

System.out.println(w);

}

}}

Test it Now

java

string

split

method

by

javatpoint

**Converting Numbers to Strings with valueOf()**

String.valueOf()

Pass your integer (as an int or Integer) to this method and it will return a string:

String.valueOf(Integer(123));

public class main{

public static void main(String[] args){

Integer i = new Integer(123);

System.out.println("Before conversion: " + i.getClass().getName());

System.out.println("After conversion: " + String.valueOf(i).getClass().getName());

}

}

**Converting integer objects to Strings**We can convert Object to String in java using toString() method of Object class or String.valueOf(object) method.

You can convert any object to String in java whether it is user-defined class, StringBuilder, StringBuffer or anything else.

Here, we are going to see two examples of converting Object into String. In the first example, we are going to convert Emp class object into String which is an user-defined class. In second example, we are going to convert StringBuilder to String.

class Emp{}

public class ObjectToStringExample{

public static void main(String args[]){

Emp e=new Emp();

String s=e.toString();

String s2=String.valueOf(e);

System.out.println(s);

System.out.println(s2);

}}

Test it Now

Output:

Emp@2a139a55

Emp@2a139a55

Java Convert Object to String

We can convert Object to String in java using toString() method of Object class or String.valueOf(object) method.

Java Convert Object to String

You can convert any object to String in java whether it is user-defined class, StringBuilder, StringBuffer or anything else.

Here, we are going to see two examples of converting Object into String. In the first example, we are going to convert Emp class object into String which is an user-defined class. In second example, we are going to convert StringBuilder to String.

Java Object to String Example: Converting User-defined class

Let's see the simple code to convert String to Object in java.

class Emp{}

public class ObjectToStringExample{

public static void main(String args[]){

Emp e=new Emp();

String s=e.toString();

String s2=String.valueOf(e);

System.out.println(s);

System.out.println(s2);

}}

Test it Now

Output:

Emp@2a139a55

Emp@2a139a55

As you can see above, a reference id of Emp class is printed on the console.

Java Object to String Example: Converting StringBuilder

Let's see the simple code to convert StringBuilder object to String in java.

public class ObjectToStringExample2{

public static void main(String args[]){

String s="hello";

StringBuilder sb=new StringBuilder(s);

sb.reverse();

String rev=sb.toString();//converting StringBuilder to String

System.out.println("String is: "+s);

System.out.println("Reverse String is: "+rev);

}}

Test it Now

Output:

String is: hello

Reverse String is: olleh

Now you can write the code to check the palindrome string.

public class ObjectToStringExample3{

public static void main(String args[]){

String s="nitin";

StringBuilder sb=new StringBuilder(s);

sb.reverse();

String rev=sb.toString();//converting StringBuilder to String

if(s.equals(rev)){

System.out.println("Palindrome String");

}else{

System.out.println("Not Palindrome String");

}

}}

Test it Now

Output:

Palindrome String

So, you can convert any Object to String in java using toString() or String.valueOf(object) methods.

**Converting to uppercase and lowercase**

To convert or change uppercase string or character to lowercase string or character in Java Programming, use the ASCII values of character to convert any character from uppercase to lowercase as shown in the first program. And the second program uses the method toLowerCase() to convert string from uppercase to lowercase.

Convert a string to upper case and lower case letters:

String txt = "Hello World";

System.out.println(txt.toUpperCase());

System.out.println(txt.toLowerCase());

Definition and Usage

The toUpperCase() method converts a string to upper case letters.

Note: The toLowerCase() method converts a string to lower case letters.

Syntax

public String toUpperCase()

Parameters

None.

Technical Details

Returns: A String value, representing the new string converted to upper case

**7. Inheritance**

**Inheritance in Java**

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

**Why use inheritance in java**

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

**Terms used in Inheritance**

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**The syntax of Java Inheritance**

class Subclass-name extends Superclass-name

{

//methods and fields

}

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Next →← Prev

Inheritance in Java

Inheritance

Types of Inheritance

**Why multiple inheritance is not possible in Java in case of class?**

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

class Subclass-name extends Superclass-name

{

//methods and fields

}

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example

**Employee**

**Salary:float**

**Programmer**

**Bonus:int**

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

class Employee{

float salary=40000;

}

class Programmer extends Employee{

int bonus=10000;

public static void main(String args[]){

Programmer p=new Programmer();

System.out.println("Programmer salary is:"+p.salary);

System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

Test it Now

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

**Types of inheritance in java**

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

Inheritance in Java

Inheritance

Types of Inheritance

Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Terms used in Inheritance

**Class**: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

**Sub Class/Child Class**: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

**Super Class/Parent Class**: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

**Reusability**: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**The syntax of Java Inheritance**

class Subclass-name extends Superclass-name

{

//methods and fields

}

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example

Inheritance in Java

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

class Employee{

float salary=40000;

}

class Programmer extends Employee{

int bonus=10000;

public static void main(String args[]){

Programmer p=new Programmer();

System.out.println("Programmer salary is:"+p.salary);

System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

Test it Now

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

**Note: Multiple inheritance is not supported in Java through class.**

When one class inherits multiple classes, it is known as multiple inheritance. For Example:

Single Inheritance Example

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void bark(){System.out.println("barking...");}

}

class TestInheritance{

public static void main(String args[]){

Dog d=new Dog();

d.bark();

d.eat();

}}

Output:

barking...

eating...

**Multilevel Inheritance Example**

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void bark(){System.out.println("barking...");}

}

class BabyDog extends Dog{

void weep(){System.out.println("weeping...");}

}

class TestInheritance2{

public static void main(String args[]){

BabyDog d=new BabyDog();

d.weep();

d.bark();

d.eat();

}}

Output:

weeping...

barking...

eating...

**Hierarchical Inheritance Example**

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void bark(){System.out.println("barking...");}

}

class Cat extends Animal{

void meow(){System.out.println("meowing...");}

}

class TestInheritance3{

public static void main(String args[]){

Cat c=new Cat();

c.meow();

c.eat();

//c.bark();//C.T.Error

}}

Output:

meowing...

eating...

**Q)** **Why multiple inheritance is not supported in java?**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

class A{

void msg(){System.out.println("Hello");}

}

class B{

void msg(){System.out.println("Welcome");}

}

class C extends A,B{//suppose if it were

public static void main(String args[]){

C obj=new C();

obj.msg();//Now which msg() method would be invoked?

}

}

Test it Now

Compile Time Error

**2. A is a super class. B is a sub class of A. C is a sub class of B.**

**Create three methods in each class, 2 methods are specific to each class and third method (override method) should be in all three Classes A, B and C**

//Java Program to demonstrate the real scenario of Java Method Overriding

//where three classes are overriding the method of a parent class.

//Creating a parent class.

class Bank{

int getRateOfInterest(){return 0;}

}

//Creating child classes.

class SBI extends Bank{

int getRateOfInterest(){return 8;}

}

class ICICI extends Bank{

int getRateOfInterest(){return 7;}

}

class AXIS extends Bank{

int getRateOfInterest(){return 9;}

}

//Test class to create objects and call the methods

class Test2{

public static void main(String args[]){

SBI s=new SBI();

ICICI i=new ICICI();

AXIS a=new AXIS();

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}

}

Output

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

**Create a class with main method. Create an object for each class A, B and C in main method and call every method of each class using its own object/instance.**

**class Student{**

**int id;**

**String name;**

**}**

**class TestStudent3{**

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

**//Creating objects**

**Student s1=new Student();**

**Student s2=new Student();**

**//Initializing objects**

**s1.id=101;**

**s1.name="Sonoo";**

**s2.id=102;**

**s2.name="Amit";**

**//Printing data**

**System.out.println(s1.id+" "+s1.name);**

**System.out.println(s2.id+" "+s2.name);**

**}**

**}**

**Output:**

**101 Sonoo**

**102 Amit**

**What is an object in Java**

**object in Java**

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

An object is a real-world entity.

An object is a runtime entity.

The object is an entity which has state and behavior.

The object is an instance of a class.

Objects and Classes in Java

Object in Java

Class in Java

Instance Variable in Java

Method in Java

Example of Object and class that maintains the records of student

Anonymous Object

In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

ADVERTISEMENT

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java

object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Characteristics of Object in Java

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

An object is a real-world entity.

An object is a runtime entity.

The object is an entity which has state and behavior.

The object is an instance of a class.

**What is a class in Java**

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

Fields

Methods

Constructors

Blocks

Nested class and interface

**Instance variable in Java**

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

Code Reusability

Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

3 Ways to initialize object

There are 3 ways to initialize object in Java.

By reference variable

By method

By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

By new keyword

By newInstance() method

By clone() method

By deserialization

By factory method etc.

**Call an overridden method with super class reference to B and C class’s objects**

**Overriding in Java**

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

// A Simple Java program to demonstrate

// method overriding in java

// Base Class

class Parent {

void show()

{

System.out.println("Parent's show()");

}

}

// Inherited class

class Child extends Parent {

// This method overrides show() of Parent

@Override

void show()

{

System.out.println("Child's show()");

}

}

// Driver class

class Main {

public static void main(String[] args)

{

// If a Parent type reference refers

// to a Parent object, then Parent's

// show is called

Parent obj1 = new Parent();

obj1.show();

// If a Parent type reference refers

// to a Child object Child's show()

// is called. This is called RUN TIME

// POLYMORPHISM.

Parent obj2 = new Child();

obj2.show();

}

}

Output:

Parent's show()

Child's show()

**Rules for method overriding:**

Overriding and Access-Modifiers: The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

**Runtime Polymorphism with Data Members/Instance variables, Repeat the above**

**process only for data members.**

In Java, we can override methods only, not the variables (data members), so runtime polymorphism cannot be achieved by data members.

// Java program to illustrate the fact that

// runtime polymorphism cannot be achieved

// by data members

// class A

class A

{

int x = 10;

}

// class B

class B extends A

{

int x = 20;

}

// Driver class

public class Test

{

public static void main(String args[])

{

A a = new B(); // object of type B

// Data member of class A will be accessed

System.out.println(a.x);

}

}

Output:

10

Explanatio : In above program, both the class A(super class) and B(sub class) have a common variable ‘x’. Now we make object of class B, referred by ‘a’ which is of type of class A. Since variables are not overridden, so the statement “a.x” will always refer to data member of super class.

**8. Access Modifiers**

Access modifiers in Java are used to define the accessibility and scope of a class or data variable. An access modifier must be specified whenever a variable or method is being defined.

Main access modifiers

There are 4 access modifiers in Java:

Private: This access modifier ensures that a member can only be accessed from within the class. Java does not offer the Private word with a class.

Public: This access modifier allows the data variable or method to be accessed anywhere. A variable can be accessed within the class or outside the class.

Protected: In a child class, the protected modified variable can be accessed within the package and outside the package. The access level of a protected modifier is only inside and outside the package through the child class.

Default: The default access modified variable can only be accessed within the same package. The variable cannot be accessed outside the package. If no access modifier is mentioned, then the Default modifier is automatically added.

**Create a class with PRIVATE fields, private method and a main method. Print the fields in main method. Call the private method in main method.**

**Create a sub class and try to access the private fields and methods from sub class**

Java Reflection - Private Fields and Methods

**Accessing Private Fields**

To access a private field you will need to call the Class.getDeclaredField(String name) or Class.getDeclaredFields() method. The methods Class.getField(String name) and Class.getFields() methods only return public fields, so they won't work. Here is a simple example of a class with a private field, and below that the code to access that field via Java Reflection:

public class PrivateObject {

private String privateString = null;

public PrivateObject(String privateString) {

this.privateString = privateString;

}

}

PrivateObject privateObject = new PrivateObject("The Private Value");

Field privateStringField = PrivateObject.class.

getDeclaredField("privateString");

privateStringField.setAccessible(true);

String fieldValue = (String) privateStringField.get(privateObject);

System.out.println("fieldValue = " + fieldValue);

This code example will print out the text "fieldValue = The Private Value", which is the value of the private field privateString of the PrivateObject instance created at the beginning of the code sample.

Notice the use of the method PrivateObject.class.getDeclaredField("privateString"). It is this method call that returns the private field. This method only returns fields declared in that particular class, not fields declared in any superclasses.

Notice the line in bold too. By calling Field.setAcessible(true) you turn off the access checks for this particular Field instance, for reflection only. Now you can access it even if it is private, protected or package scope, even if the caller is not part of those scopes. You still can't access the field using normal code. The compiler won't allow it.

**Accessing Private Methods**

To access a private method you will need to call the Class.getDeclaredMethod(String name, Class[] parameterTypes) or Class.getDeclaredMethods() method. The methods Class.getMethod(String name, Class[] parameterTypes) and Class.getMethods() methods only return public methods, so they won't work. Here is a simple example of a class with a private method, and below that the code to access that method via Java Reflection:

public class PrivateObject {

private String privateString = null;

public PrivateObject(String privateString) {

this.privateString = privateString;

}

private String getPrivateString(){

return this.privateString;

}

}

PrivateObject privateObject = new PrivateObject("The Private Value");

Method privateStringMethod = PrivateObject.class.

getDeclaredMethod("getPrivateString", null);

privateStringMethod.setAccessible(true);

String returnValue = (String)

privateStringMethod.invoke(privateObject, null);

System.out.println("returnValue = " + returnValue);

This code example will print out the text "returnValue = The Private Value", which is the value returned by the method getPrivateString() when invoked on the PrivateObject instance created at the beginning of the code sample.

Notice the use of the method PrivateObject.class.getDeclaredMethod("privateString"). It is this method call that returns the private method. This method only returns methods declared in that particular class, not methods declared in any superclasses.

Notice the line in bold too. By calling Method.setAcessible(true) you turn off the access checks for this particular Method instance, for reflection only. Now you can access it even if it is private, protected or package scope, even if the caller is not part of those scopes. You still can't access the method using normal code. The compiler won't allow it.

**Create a class with DEFAULT fields and methods. Access these fields and methods**

**from any other class in the same package**

**. Default access modifier**

When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package. Lets see an example to understand this:

**Default Access Modifier Example in Java**

To understand this example, you must have the knowledge of packages in java.

In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.

**Addition.java**

package abcpackage;

public class Addition {

/\* Since we didn't mention any access modifier here, it would

\* be considered as default.

\*/

int addTwoNumbers(int a, int b){

return a+b;

}

}

Test.java

package xyzpackage;

/\* We are importing the abcpackage

\* but still we will get error because the

\* class we are trying to use has default access

\* modifier.

\*/

import abcpackage.\*;

public class Test {

public static void main(String args[]){

Addition obj = new Addition();

/\* It will throw error because we are trying to access

\* the default method in another package

\*/

obj.addTwoNumbers(10, 21);

}

}

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The method addTwoNumbers(int, int) from the type Addition is not visible

at xyzpackage.Test.main(Test.java:12)

**Create a class with PROTECTED fields and methods. Access these fields and methods**

**from any other class in the same package.**

**Also, Access the PROTECTED fields and methods from child class located in a different**

**package**

**Access the PROTECTED fields and methods from any class in different package**

A Java protected keyword is an access modifier. It can be assigned to variables, methods, constructors and inner classes.

The protected Keyword

While elements declared as private can be accessed only by the class in which they're declared, the protected keyword allows access from sub-classes and members of the same package.

By using the protected keyword, we make decisions about which methods and fields should be considered internals of a package or class hierarchy, and which are exposed to outside code.

3. Declaring protected Fields, Methods, and Constructors

First, let's create a class named FirstClass containing a protected field, method, and constructor:

public class FirstClass {

protected String name;

protected FirstClass(String name) {

this.name = name;

}

protected String getName() {

return name;

}

}

With this example, by using the protected keyword, we've granted access to these fields to classes in the same package as FirstClass and to sub-classes of FirstClass.

4. Accessing protected Fields, Methods, and Constructors

4.1 From the Same Package

Now, let's see how we can access protected fields by creating a new GenericClass declared in the same package as FirstClass:

public class GenericClass {

public static void main(String[] args) {

FirstClass first = new FirstClass("random name");

System.out.println("FirstClass name is " + first.getName());

first.name = "new name";

}

}

As this calling class is in the same package as FirstClass, it's allowed to see and interact with all the protected fields, methods, and constructors.

4.2. From a Different Package

Let's now try to interact with these fields from a class declared in a different package from FirstClass:

public class SecondGenericClass {

public static void main(String[] args) {

FirstClass first = new FirstClass("random name");

System.out.println("FirstClass name is "+ first.getName());

first.name = "new name";

}

}

As we can see, we get compilation errors:

freestar

The constructor FirstClass(String) is not visible

The method getName() from the type FirstClass is not visible

The field FirstClass.name is not visible

That's exactly what we were expecting by using the protected keyword. This is because SecondGenericClass is not in the same package as FirstClass and does not subclass it.

4.3 From a Sub-Class

Let's now see what happens when we declare a class extending FirstClass but declared in a different package:

public class SecondClass extends FirstClass {

public SecondClass(String name) {

super(name);

System.out.println("SecondClass name is " + this.getName());

this.name = "new name";

}

}

As expected, we can access all the protected fields, methods, and constructors. This is because SecondClass is a sub-class of FirstClass.

5. protected Inner Class

In the previous examples, we saw protected fields, methods, and constructors in action. There is one more particular case — a protected inner class.

Let's create this empty inner class inside our FirstClass:

freestar

package com.baeldung.core.modifiers;

public class FirstClass {

// ...

protected static class InnerClass {

}

}

As we can see, this is a static inner class, and so can be constructed from outside of an instance of FirstClass. However, as it is protected, we can only instantiate it from code in the same package as FirstClass.

5.1 From the Same Package

To test this, let's edit our GenericClass:

public class GenericClass {

public static void main(String[] args) {

// ...

FirstClass.InnerClass innerClass = new FirstClass.InnerClass();

}

}

As we can see, we can instantiate the InnerClass without any problem because GenericClass is in the same package as FirstClass.

5.2. From a Different Package

Let's try to instantiate an InnerClass from our SecondGenericClass which, as we remember, is outside FirstClass' package:

public class SecondGenericClass {

public static void main(String[] args) {

// ...

FirstClass.InnerClass innerClass = new FirstClass.InnerClass();

}

}

As expected, we get a compilation error:

The type FirstClass.InnerClass is not visible

5.3. From a Sub-Class

Let's try to do the same from our SecondClass:

public class SecondClass extends FirstClass {

public SecondClass(String name) {

// ...

FirstClass.InnerClass innerClass = new FirstClass.InnerClass();

}

}

We were expecting to instantiate our InnerClass with ease. However, we are getting a compilation error here too:

The constructor FirstClass.InnerClass() is not visible

Let's take a look at our InnerClass declaration:

freestar

protected static class InnerClass {

}

The main reason we are getting this error is that the default constructor of a protected class is implicitly protected. In addition, SecondClass is a sub-class of FirstClass but is not a sub-class of InnerClass. Finally, we also declared SecondClass outside FirstClass' package.

For all these reasons, SecondClass can't access the protected InnerClass constructor.

If we wanted to solve this issue and allow our SecondClass to instantiate an InnerClass object, we could explicitly declare a public constructor:

protected static class InnerClass {

public InnerClass() {

}

}

By doing this, we no longer get a compilation error, and we can now instantiate an InnerClass from SecondClass.

6. Conclusion

In this quick tutorial, we discussed the protected access modifier in Java. With it, we can ensure exposing only the required data and methods to sub-classes and classes in the same package.

As always, the example code is available over on GitHub.

**Create a class with PUBLIC fields and methods.**

**Access the public methods and fields from any class in the same package or different**

**package**

public Access Modifier

The Java access modifier public means that all code can access the class, field, constructor or method, regardless of where the accessing code is located. The accessing code can be in a different class and different package.

Here is a public access modifier example:

public class Clock {

public long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

The time field in the Clock class is marked with the public Java access modifier. Therefore, the ClockReader class can access the time field in the Clock no matter what package the ClockReader is located in.

Polymorphism in Java is a concept by which we can perform a single action in different ways. ... So polymorphism means many forms. There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.