**JAVA**

***Java is an Object oriented, statically typed and imperative programming language.***

***Note:***

*A****language****is****statically******typed****if the variable type is known at compile time.*

**Prereqisites:**

1. **JDK**

2. **IDE** - **ECLIPSE / INTELLIJ IDEA**

Java Concepts list:

* **Basics**
* **Modifiers**
* **Control Statements**
* **Class and its types**
* Methods and Constructors
* OOPS Concepts
* Enumerators
* Errrors and Exceptions
* File operations – Text/Excel/XML/JSON
* Data Structures
* Collections
* Multi-Threading
* Memory management
* Inbuilt classes and Interfaces (important ones)
* Key Programs

1. **BASICS**

***Key Features of the Java Language:***

1. ***Simplicity***

Java is simpler than most other languages that are used to create server applications,because of its consistent enforcement of the ***object model.***

The **large, standard set of class libraries** brings powerful tools to Java developers on all platforms.

1. ***Portability***

Java is **portable across platforms.**

It is possible to write ***platform-independent* code** in Java, and it is also simple to write programs that move seamlessly across systems.

***Java is independent of OS.***

*WORA -* ***Write Once and Run Anywhere.***

1. ***Multithreading***is a Java feature that allows ***concurrent execution*** *of two or more parts of a program****for maximum utilization of CPU****.*

Each part of such program is called a **thread.**

***threads are light-weight processes within a process.***

**JAR: Java Archive**

*A JAR (Java Archive) is a* ***package file format*** *typically used to* ***aggregate*** *many* ***Java******class files*** *and associated* ***metadata*** *and* ***resources*** *(text, images, etc.)* ***into one file******to distribute application software*** *or* ***libraries on the Java platform.***

*In simple words, a JAR file is a file that contains compressed version of .class files, audio files, image files or directories.*

***Note:***

When we create .jar files, it automatically receives the default manifest file.

There can be *only one manifest file in an archive*, and it always has the pathname.

example: META-INF/MANIFEST.MF

*This* ***manifest file*** *is useful to specify the* ***information*** *about* ***other files which are packaged.***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **JAR file** | | | | |
| *Action* | *Command* | *way/mode* | *example* | *Explanation* |
| Create | jar cf | jar cf jarfilenameinputfiles | C:\> jar cf pack.jar pack | cf represents create the file package pack is available in C:\directory , to convert it into a jar file into the pack.jar |
| View | jar tf | jar tf jarfilename | C:/> jar tf pack.jar | tf - table view of file contents |
| Extract | jar xf | jarfilename | C:\> jar xf pack.jar | xf - extract files from the jar files |
| Update | jar uf | jar uf jar-file input-file(s) | C:\>jar uf pack.jar | uf - update jar file |
| **Run** | **java -jar** |  | C:\>java -jar pack.jar |  |

**JDK: Java Development Kit**

A screenshot of a cell phone

Description automatically generated

*The Java Development Kit (****JDK****) is a* ***software development environment*** *used for developing* ***Java applications*** *and* ***applets****.*

**JDK** includes

* Java Runtime Environment (JRE),
* interpreter/loader (java),
* compiler (javac),
* archiver (jar),
* documentation generator (javadoc) and other tools needed in Java development.

***JDK physically exists,*** *JDK contains JRE + development tools.*

***Note:***

***An applet is a small Internet-based program written in Java, which can be downloaded by any computer.***

*An****applet****is a Java program that can be* ***embedded*** *into a* ***web page.***

*It* ***runs******inside*** *the* ***web browser*** *and* ***works at client side.***

*An applet is embedded in an HTML page using the****APPLET****or* ***OBJECT tag*** *and* ***hosted on a web server****.*

*Applets are used to make the* ***web site*** *more* ***dynamic*** *and* ***entertaining.***

*The applet is also* ***able to run in HTML.***

*The applet is usually* ***embedded in an HTML page*** *on a Web site and can be* ***executed from within a browser.***

**Interpreter:**

an interpreter is a computer **program** that **directly executes**, i.e. performs execution of instructions written in a programming or scripting language,without requiring them previously to have been compiled into a machine language program.

*An interpreter generally uses one of the following strategies for program execution:*

* parse the source code and perform its behaviour directly;
* translate source code into some efficient intermediate representation and immediately execute this;
* explicitly execute stored precompiled code by a compiler which is part of the interpreter system.

While interpretation and compilation are the two main means by which programming languages are implemented, they are not mutually exclusive,

as most interpreting systems also perform some translation work, just like compilers.

The terms "interpreted language" or "compiled language" signify that the canonical implementation of that language is an interpreter or a compiler, respectively.

***A high level language is ideally an abstraction independent of particular implementations.***

***Note:***

***Abstraction*** *- a concept or idea not associated with any specific instance ·*

*a general concept formed by extracting common features from specific examples ·*

***Note:***

***Source code*** *is the fundamental component of a computer program that is created by a programmer.*

***Pseudocode*** *is an informal high-level description of the operating principle of a computer program or other algorithm.*

*Pseudocode* uses the *structural conventions of a normal programming language, but is intended for human reading rather than machine reading.*

*Pseudocode is not an actual programming language.*

*So, it cannot be compiled into an executable program.*

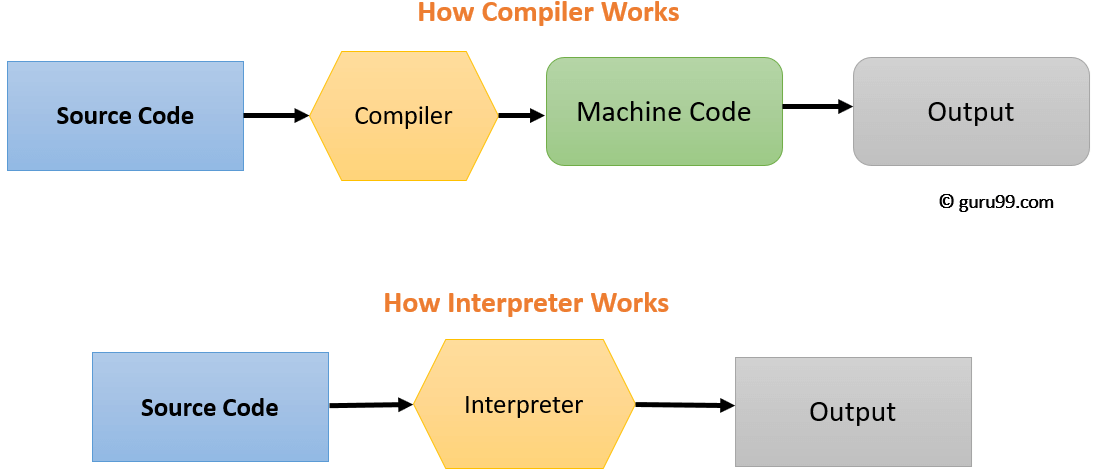
***Machine code*** *= Binary code- low level code understandable by machines.*

|  |  |  |  |
| --- | --- | --- | --- |
| *Code* | *Software* | *Readable by* | *File extensions* |
| Pseudo code | NA | Humans | ex: .txt/.xlsx/.pdf/.docx |
| Source code | IDE | Humans | .java for Java |
| Machine code | Compiler/ Interpreter | Machine | .class for Java |

***Compiler:***

*Compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language*

*(e.g., assembly language, object code, or machine code) to create an executable program.*



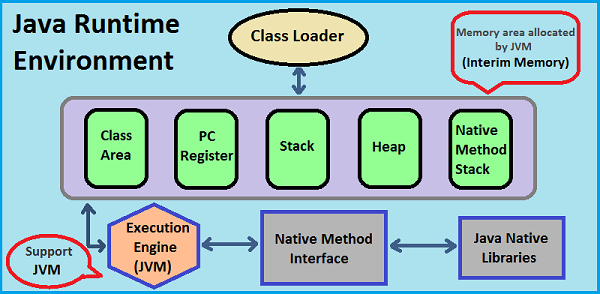
***Note:***

*An interpreter is a computer program, which converts each high-level program statement into the machine code.*

*This includes source code, pre-compiled code, and scripts.*

*Both compiler and interpreters do the same job which is converting higher level programming language to machine code.*

**JRE:**



*JRE is an acronym for Java Runtime Environment.*

It is used to provide ***runtime environment.***

It is the ***implementation of JVM.***

It **physically exists.**

It contains ***set of libraries + other files*** *that* ***JVM uses at runtime.***

**JVM:**

A screenshot of a cell phone

Description automatically generated

***JVM (Java Virtual Machine) is an abstract machine.***

***JVM*** *is a* ***specification*** *that provides* ***runtime environment*** *in* ***which java bytecode*** *can be executed.*

*JVMs are available for many hardware and software platforms.*

***JVM, JRE and JDK are platform dependent because configuration of each OS differs.***

There are ***three notions of the JVM:***

* ***specification***
* ***implementation***
* ***instance.***

The *JVM performs following main tasks:*

* ***Loads code***
* ***Verifies code***
* ***Executes code***
* ***Provides runtime environment***

**JVM Architecture:**

JVM (Java Virtual Machine) acts as a **run-time engine** to run Java applications.

***JVM*** *is the one that actually* ***calls the main method*** *present in a java code.*

*JVM is a part of JRE(Java Run Environment).*

*When we compile a .java file, a .class file(contains byte-code) with the same filename is generated by the* ***Java compiler.***

***the .class file goes into various steps when we run it.***

***These steps together describe the whole JVM.***

**JVM** mainly responsible for ***three activities.***

* **Loading**
* **Linking**
* **Initialization**

1. **Loading:**

* The **Class loader** ***reads*** *the* ***.class*** *file,* ***generate*** *the corresponding* ***binary data*** *and* ***save it in method area.***
* For each .class file, ***JVM stores*** *following* ***information******in method area.***

***Fully qualified name of the loaded class*** *and its* ***immediate parent class.***

Whether **.class file is related to *Class or Interface or Enum***

**Modifier, Variables and Method information** etc.

* *After loading .class file, JVM creates an* ***object of type Class*** *to represent this* ***file*** *in the* ***heap memory.***
* ***For every loaded .class file, only one object of Class is created.***

1. **Linking:** Performs
2. **Verification:**

*It ensures the correctness of .class file i.e. it check whether this file is properly formatted and generated by valid compiler or not.*

If verification fails, we get **run-time exception** **java.lang.VerifyError.**

1. **Preparation:**

***JVM*** *allocates* ***memory*** *for class* ***variables*** *and* ***initializing*** *the* ***memory to default values.***

1. **Resolution:**

It is the process of **replacing** **symbolic references from the type** with **direct references.**

It is done by **searching into method area** to **locate the referenced entity.**

***3. Initialization*:**

In this phase, ***all static variables*** *are* ***assigned*** *with their* ***values*** *defined in the code and static block(if any).*

This is executed from **top to bottom in a class** and from **parent to child** in class hierarchy.

***Class loaders: 3 in numbers.***

1. **Bootstrap class loader:**

Every JVM implementation must have a bootstrap class loader, capable of **loading trusted classes.**

It loads core java API classes present in **JAVA\_HOME/jre/lib** directory.

This path is popularly known as **bootstrap path.**

It is **implemented in native languages** like C, C++.

1. **Extension class loader:**

It is **child of bootstrap** class loader.

It loads the classes present in the extensions directories **JAVA\_HOME/jre/lib/ext** (**Extension path**) or

any other directory specified by the **java.ext.dirs** system property.

It is implemented in java by the **sun.misc.Launcher$ExtClassLoader class.**

1. **System/Application class loader:**

It is **child of extension** class loader.

It is responsible to load classes from **application class path.**

It internally uses Environment Variable which mapped to java.class.path.

It is also implemented in Java by the **sun.misc.Launcher$AppClassLoader class.**

***Note:***

*JVM follow Delegation-Hierarchy principle to load classes.*

*System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader.*

*If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader.*

*At last if system class loader fails to load class, then we get* ***run-time exception java.lang.ClassNotFoundException.***

**JVM Memory**

**Method area:**

*In method area,* ***all class level information*** *like* ***class name****,* ***immediate parent class name, methods and variables information*** *etc. are stored, including* ***static variables.***

*There is* ***only one method area per JVM****, and it is a* ***shared*** *resource.*

**Heap area:**

**Information** of all **objects** is stored in heap area.

There is also **one Heap Area per JVM.** It is also a **shared** resource.

**Stack area:**

***For every thread, JVM create one run-time stack which is stored here.***

**Every block of this stack is called activation record/stack frame** which store **methods calls.**

All local variables of that method are stored in their corresponding frame.

***After a thread terminate, it’s run-time stack will be destroyed by JVM.***

*It is* ***not a shared resource.***

**PC Registers:**

Store **address** of **current execution instruction of a thread.**

Obviously each thread has separate PC Registers.

**Native method stacks:**

***For every thread, separate native stack is created.***

It stores **native method information.**

**Execution Engine:**

*Execution engine* ***execute the .class (bytecode).***

It reads the byte-code line by line, use data and information present in various memory area and execute instructions.

It can be **classified in to three parts: -**

1. ***Interpreter*:**

It interprets the bytecode line by line and then executes.

The disadvantage here is that when one method is called multiple times, every time interpretation is required.

***2. Just-In-Time Compiler(JIT):***

It is used to ***increase efficiency of interpreter.***

It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,

JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.

1. **Garbage Collector:**

*It* ***destroy un-referenced objects.***

***In java, garbage means unreferenced objects.***

***Garbage Collection is the process of reclaiming the runtime unused memory automatically.***

In other words, it is a way to *destroy the unused objects.*

**Java Native Interface (JNI):**

It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution.

It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries:**

*It is a collection of the Native Libraries(C, C++) which are* ***required by the Execution Engine.***

***Javadoc:***

**Javadoc** is a tool which comes with JDK

Javadoc is for ***generating API documentation in HTML format from Java source code.***

**/\*\* documentation \*/**

This is a documentation comment and in general it’s called doc comment.

The JDK javadoc tool uses doc comments when preparing automatically generated documentation.

***Java Program structure***

*Java project* creation

*Package* declaration

Ex: package com.abcd[by Convention: reverse DNS system]

***Import statements***

We can import built in and user defined libraries using import keyword.

(in order to use existing classes, objects and methods)

Ex. import java.io.Console; //import console class from IO package

import *java.io.\*;* // import all classes from IO package

java - project

io - package

Console - class

***Note:***

***static import in java :***

*Static import is a feature introduced in the Java programming language that allows* ***members (fields and methods)*** *which have been scoped within their container class as* ***public static****, to be used in Java code without specifying the class in which the field has been defined.*

***Class declaration statement***

*Ex: public class SampleProg {*

*}*

*public - access modifier*

*class - Java keyword to declare a class*

*SampleProg - name of the class*

***Note:***

*1st letter of the class name should be upper case by convention.*

***Note:***

***“Use a project specific JRE”***

*To build the project automatically and independently “Use a project specific JRE”.*

***src - package***

src - will have the *source code* written.

to work on a project with hundreds (or thousands) of source files, having intuitive sub-packages are an absolute necessity to keep everything organized.

Using a "src" directory is a convention that the IDE, build tools, and other programmers all readily understand.

***Compile and Run Java Program in Cmd Prompt:***

write a program in a notepad ide and save as **ClassName.java**

(select file types as - "All types")

1. Open cmd

2. Navigate to the directory where .java file exists

3. Type **javac** file\_name.java (.class file generates)

4. Type **java** file\_name (output will display)

***Project Creation in Eclipse:***

Follow the below Steps:

File -> New -> Java Project

a) Provide Project name

b) Under JRE section select a radio button corresponding to 'Use a project specific JRE'

c) Click on 'Configure JREs'. Under installed JREs if anything present select that and remove

d) Click on Add. Select 'Standard VM'. Click on Next. In the JRE home text box provide installed

JDK directory location (e.g. C:\Program Files\Java\jdk1.7.0\_80). Click on FINISH.

e) Select the check box under Installed JREs. Click OK. Your project gets created. You will find it under project explorer.

f) Right click on the project. New -> Class

g) Provide the package name. Please note that the naming convention of package is reverse DNS order (e.g. com.sellabs)

h) Provide the class name. Please note that class name should start with upper case letter and method name should start with lower case letter.

i) select the check box corresponding to 'Generate comments' and click on Finish.

j) Now you may write your code as per requirement(s).

**Class:**

*A class is a user defined blueprint or prototype from which objects are created.*

*It represents the set of properties or methods that are common to all objects of one type.*

*In object-oriented programming, a class is an* ***extensible program-code-template*** *for creating objects, providing initial values for* ***state (member variables)*** *and implementations of* ***behaviour (member functions or methods).***

***Note:***

*a class that creates classes is called a metaclass.*

***A class is a template for objects.***

*A class defines object properties including a valid range of values, and a default value.*

*A class also describes object behaviour.*

**Simple class :** independent of any other class

***Derived classes :***

*a class that shares properties with another class but yet is distinct from the original.*

The new class derives properties from an existing class but also extends or adds its own properties.

This new class is called a "derived class" and is said to "inherit" its properties and functionality from the original class.

***Superclasses and subclasses:***

*A way to refer to a derived class is as a* ***"subclass****" of the original class.*

*The class from which the subclass is derived is known as its* ***"superclass".***

*The superclass of all subclasses is the* ***Base class.***

***Mixed classes:***

*A mixed class is a way to combine the functionality from other classes into a new class.*

A mixed class manages the properties of other classes and may only use a subset of the functionality of a class, whereas a derived class uses the complete set of functionalities of its superclasses and usually extends this functionality.

***Composite classes:***

*Composite class members can be* ***nested.***

only have to create **one object** in program to have the effective functionality of a whole tree of composite class members.

|  |  |
| --- | --- |
| ***Class Members*** | *Description* |
| Comments | Ignored for compilation |
| Object | derived from a class |
| main method | in built - for execution |
| Data types | Primitive + Secondary |
| Variables | Name of the memory location |
| Keywords | Reserve words - specific meaning |
| Modifiers | Access + Non-access |
| Methods | user defined |
| Constructor | Non-class member |

***Java Comments:***

The java comments are 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 for specific time.

There are **3 types of comments** in java.

***Single Line*** *Comment to comment only one line.*

// This is single line comment.

***Multi Line*** *Comment to comment multiple lines of code.*

Syntax:

/\*

This

is

multi line

comment

\*/

***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

\*/

Example:

/\*\* The Calculator class provides methods to get addition and subtraction of given 2 numbers.\*/

public class Calculator {

/\*\* The add() method returns addition of given numbers.\*/

public static int add(int a, int b){return a+b;}

/\*\* The sub() method returns subtraction of given numbers.\*/

public static int sub(int a, int b){return a-b;}

}

Compile it by javac tool:

javac Calculator.java

Create Documentation API by javadoc tool:

javadoc Calculator.java

Now, there will be HTML files created for your Calculator class in the current directory. Open the HTML files and see the explanation of Calculator class provided through documentation comment.

***OBJECT :***

*An object is* ***a member or an "instance" of a class.***

An object has a state in which all of its properties have values that either explicitly defined or that are defined by default settings.

// Syntax

ClassName objReferenceVariable = new ClassName();

// ClassName() - Object

// objReferenceVariable - memory where Object is stored

// new – for object creation

***MAIN METHOD:***

// Syntax:

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

/\* MAIN METHOD IS:

\* 1. *a class member*

\* 2. *does not belong to any specific class*

\* 3. non-return type

\* 4. parameterised

\* 5. mandatory for execution but not for compilation

\* **public** - An access modifier / scope - Throughout the project /

should use a public keyword before the main() method so that **JVM can identify the execution point of the program.**

If we use private, protected, and default before the main() method, it will not be visible to JVM.

\* **static** – a non-access modifier /

We should call the main() method **without creating an object.**

***Static methods are the method which invokes without creating the objects,***

so we do not need any object to call the main() method.

\* **void** - return nothing

void keyword acknowledges the compiler that main() method does not return any value.

\* **main** –

It is a **default signature** which is predefined in the **JVM**.

It is called by JVM to execute a program line by line and end the execution after completion of this method.

We can **also overload the main() method.**

**\* String[] args:**

The main() method also accepts some data from the user.

It accepts a group of strings, which is called a **string array.**

It is used to hold the command line arguments in the form of string values.

**main(String args[])**

args[] is the array name, and it is of String type.

It means that it can store a group of string. this array can also store a group of numbers but in the form of string only.

Values passed to the main() method is called arguments.

These arguments are stored into array, so the name args[] is generally used for it.

***Note:***

**main method is the entry point of any java program.**

***JVM is configured to look for a specific method signature to start running an application.***

if the main() method is written without String args[]?

The program will compile, but not run, because JVM will not recognize the main() method.

***JVM always looks for the main() method with a string type array as a parameter.***

Different ways of writing main() method are:

static public void main(String []x)

static public void main(String...args)

**String...args:**

It allows the method to accept zero or multiple arguments.

There should be exactly three dots between String and array; otherwise, it gives an error.

**Overloading of main() method**

We can also overload the main() method. We can define any number of main() method in the class, but the method signature must be different.

Example:

**class** OverloadMain

{

**public** **static** **void** main(**int** a)  //overloaded main method

{

System.out.println(a);

}

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

{

System.out.println("main method invoked");

main(6);

}

}

**Output:**

main method invoked

6

***Output Statement:***

**System.out.println()**

***is a Java statement that prints the argument passed, into the System.out which is generally stdout.***

**System – is a final class** in **java.lang** package.

**out – is a static member field of System class and is of type**

**PrintStream**.

Its access specifiers are ***public****and* ***final***.

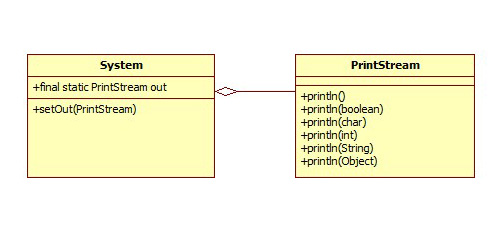
This gets instantiated during startup and gets mapped with standard output console of the host.

This stream is open by itself immediately after its instantiation and ready to accept data.

**println() – is a method of PrintStream class**.

println prints the argument passed to the standard console and a newline. There are multiple println methods with different arguments (overloading). ***Every println makes a call to print() method and adds a newline.*** print calls write() and the story goes on like that.

Structure of **System.out.println()**:



***Scanner Class in Java:***

Scanner is a class in **java.util** package used for obtaining the input of the primitive types like **int, double etc.** and **strings**.

To create an object of Scanner class, we pass the predefined object **System.in,** which represents the standard input stream.

We may pass an ***object of class File*** *if we want to read input from a file.*

To read numerical values of a certain data type XYZ, the function to use is **nextXYZ().**

For example, to read a value of type **short**, we can use **nextShort()**

*To read* ***strings****, we use* ***nextLine().***

*To read a* ***single character****, we use* ***next().charAt(0).***

next() function returns the next token/word in the input as a string and charAt(0) funtion returns the first character in that string.

// Java program to read data of various types using Scanner class.

import java.util.Scanner;

public class ScannerDemo1

{

public static void main(String[] args)

{

// Declare the object and initialize with

// predefined standard input object

**Scanner sc = new Scanner(System.in);**

// String input

String name = **sc.nextLine();**

// Character input

char gender = **sc.next().charAt(0);**

// Numerical data input

// byte, short and float can be read

// using similar-named functions.

int age = **sc.nextInt();**

long mobileNo = **sc.nextLong();**

double cgpa = **sc.nextDouble();**

// Print the values to check if input was correctly obtained.

System.out.println("Name: "+name);

System.out.println("Gender: "+gender);

System.out.println("Age: "+age);

System.out.println("Mobile Number: "+mobileNo);

System.out.println("CGPA: "+cgpa);

}

}

Sometimes, we have to check if the next value we read is of a certain type or if the input has ended (EOF marker encountered).

Then, we **check if the scanner’s input is of the type we want** with the help of **hasNextXYZ()** functions where **XYZ** is the **type** we are interested in.

The function returns true if the scanner has a token of that type, otherwise false. For example, in the above code, we have used hasNextInt().

To check for a **string**, we use **hasNextLine().**

Similarly, to check for a single **character**, we use **hasNext().charAt(0).**

// Java program to read some values using Scanner

// class and print their mean.

import java.util.Scanner;

public class ScannerDemo2

{

public static void main(String[] args)

{

// Declare an object and initialize with

// predefined standard input object

Scanner sc = new Scanner(System.in);

// Initialize sum and count of input elements

int sum = 0, count = 0;

// Check if an int value is available

while (sc.hasNextInt())

{

// Read an int value

int num = sc.nextInt();

sum += num;

count++;

}

int mean = sum / count;

System.out.println("Mean: " + mean);

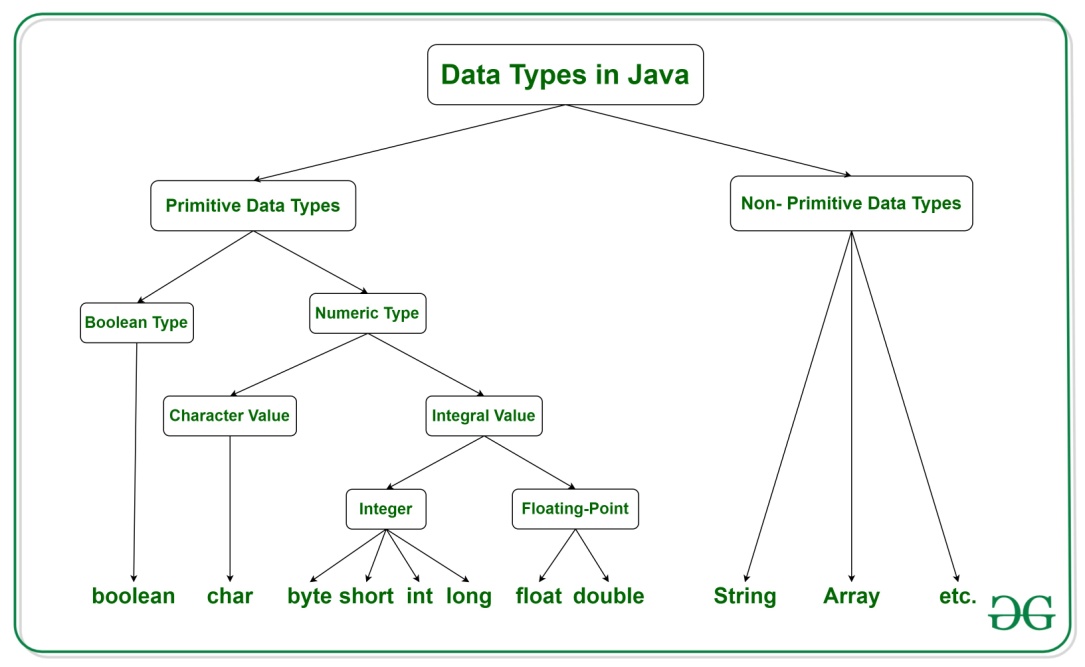
}

}

***DATA TYPES:***

*Data: A collection of information.*

***Data types specify the different sizes and values that can be stored in the variable.***



1. **Primitive Data Type**

*Primitive data are only single values and have no special capabilities.*

***Note:***

***boolean:****boolean data type represents only one bit of information****either true or false****, but the size of boolean data type is****virtual machine-dependent****.*

***Values of type boolean are not converted implicitly or explicitly (with casts) to any other type.***

**2. Non-Primitive Data Type or Reference Data Types**

The **Reference Data Types** will contain a **memory address** of **variable value** because the reference types won’t store the variable value directly in memory. They are strings, objects, arrays, etc.

**String**: Strings are defined as an array of characters. The difference between a character array and a string is the **string is terminated with a special character ‘\0’.**

**Class**: A class is a user-defined blueprint or prototype from which objects are created.  It represents the set of properties or methods that are common to all objects of one type.

In general, class declarations can include these components, in order:

**Modifiers** : A class can be public or has default access.

**Class name:** The name should begin with a initial letter (capitalized by convention).

**Superclass(if any):** The name of the class’s parent (superclass), if any, preceded by the keyword **extends**. A class (subclass) can extend only one parent.

**Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword **implements**. A class can implement more than one interface.

**Body:** The class body surrounded by braces, { }.

**Object**: It is a basic unit of Object-Oriented Programming and represents the real-life entities.

A typical ***Java program creates many objects, which interact by invoking methods.***

An object consists of :

**State**: It is represented by **attributes** of an object. It also reflects the **properties** of an object.

**Behavior**: It is represented by **methods** of an object. It also reflects the **response** of an object with other objects.

**Identity**: It gives a **unique name** to an object and enables one object to interact with other objects.

**Interface:** Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

***Interfaces specify what a class must do and not how.***

*It is the* ***blueprint of the class.***

An **Interface** is about **capabilities** like a Player may be an interface and any class implementing Player must be able to (or must implement) move().

So it ***specifies a set of methods that the class has to implement.***

If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared **abstract**.

A Java library example is, Comparator Interface.

If a class implements this interface, then it can be used to sort a collection.

**Array:**

An array is a group of like-typed variables that are referred to by a common name.

Following are some important point about Java arrays.

*In Java all* ***arrays*** *are* ***dynamically allocated.***

Since arrays are **objects** in **Java**, we can find their length using member length.

A Java array variable can also be declared like other variables with [] after the data type.

The variables in the array are ordered and each have an index beginning from 0.

* ***Java array can be used as a static field, a local variable or a method parameter.***
* ***The size of an array must be specified by an int value and not long or short.***
* ***The direct superclass of an array type is Object.***

Every **array** type **implements** the **interfaces** **Cloneable** and **Serializable**.

**IDENTIFIERS:**

 Identifiers are the ***names*** *of* ***variables****,* ***methods****,* ***classes****,* ***packages*** *and* ***interfaces****.*

In Java, all identifiers must begin with *a letter, an underscore, or a Unicodecurrency character.*

Any other symbol, such as a number, is not valid.

an identifier cannot have the same spelling as one of Java's reserved words.

*An Indetifier can only have alphanumeric characters( a-z , A-Z , 0-9 ) and underscore( \_ ).*

*The first character of an identifier can only contain alphabet( a-z , A-Z ) or underscore ( \_ ).*

***VARIABLES AND ITS TYPES:***

**datatype variableName = value;**

***A variable is a container which holds the value while the java program is executed.***

A variable is assigned with a datatype.

***Variable is a name of memory location.***

There are ***three types of variables in java: local, instance<global> and static****.*

*1.* ***Instance variable in java is used by Objects to store their states.***

Variables which are defined **without** the **static** keyword and are **Outside any method** declaration are **Object specific** and are known as instance variables.

*They are called so because their values are instance specific and are* ***not shared among instances.***

***Rules for Instance variable in Java:***

*Instance variables can use any of the four access levels*

*(* **Default**, ***Public, Private, Protected*** *)*

***Can*** *be* **final** and **transient**.

***Cannot*** *be* **abstract, synchronized, strictfp, native, static.**

*Instance variable will get default value means instance variable can be used without initializing it.*

*Same is not true for***Local Variable.**

**Note:**

**Instance Variable cannot have Static modifier as it will become Class level variable.**

|  |  |
| --- | --- |
| *INSTANCE VARIABLE TYPE* | *DEFAULT VALUE* |
| *Boolean* | *false* |
| Byte | (byte)0 |
| Short | (short) 0 |
| Int | 0 |
| Long | 0L |
| Char | u0000 |
| Float | 0.0f |
| Double | 0.0d |
| *Object* | *Null* |

***2. local variable*** *in Java is a variable that's declared* ***within the body of a method.***

can be used only within that method.

Other methods in the class aren't even aware that the variable exists.

*Local variables are* ***not given initial default values.***

***3. Static variable:***

Static Keyword Rules:

***Variable or Methods*** marked static belong to the **Class** rather than to any particular Instance.

**Static Method or variable** can be used without creating or referencing an instance of the Class.

***Note*:**

If there are instances, a static variable of a Class will be shared by all instances of that class, This will result in **only one copy**.

***Note:***

***A static Method can’t access a non-static variable nor can directly invoke non static Method.***

(*It can invoke or access Method or variable via*instances*).*

*Summary of static variable:*

**Static**is a Non-Access Modifier.

***The Static****modifier can be applied to a variable or Method or block or inner Class.*

**Static members** belong to Class only, not an instance.

A Static Method **cannot** access an instance variable.

***Static Methods cannot be*overriden a*s they are Class specific and don’t belong to an Instance.***

***Static Methods can be******redefined****.*

***Note:***

*If a Class contains any static blocks, then that block will be executed only when the Class is loaded in JVM.*

Creating multiple instances does not run the static block multiple time.

Only the constructor will be executed multiple time.

If Class.forName(“class\_name“) is called then the static block of the Class will get executed.

***Note:***

*A block statement is a sequence of zero or more statements enclosed in braces.*

A block statement is generally used to group together several statements, so they can be used in a situation that requires you to use a single statement.

***Note:***

***Class variable***

A class variable is **not an instance variable.**

It is a **special type** of **class attribute.**

***Note:***

*in Java, the terms "****field****" and "****variable****" are used interchangeably for* ***member variable.***

**TYPE CASTING:**

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

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

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

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

Widening Casting

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

Example:

Public class MyClass{

Public static void main(String[]args){

int myInt=9;

double myDouble=myInt;// Automatic casting: int to double

System.out.println(myInt);// Outputs 9

System.out.println(myDouble);// Outputs 9.0

}

}

Narrowing Casting

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

Example:

publicclassMyClass{:

publicstaticvoidmain(String[]args){

double myDouble=9.78;

int myInt=(int)myDouble;// Manual casting: double to int

System.out.println(myDouble);// Outputs 9.78

System.out.println(myInt);// Outputs 9

}

}

***Keywords***

*Reserve words having* ***specific meaning****, whose meaning cannot be changed.*

Keywords or Reserved words are the words in a language that are used for some **internal process** or represent some **predefined actions**.

programmers cannot use keywords as *names(identifiers)* for variables, methods, classes, or as any other identifier Doing this will result into a **compile time error.**

***All keywords are small case in characters.***

|  |  |
| --- | --- |
| ***Java Keywords*** | ***Description*** |
| **abstract** | Specifies that a class or method will be implemented later, in a subclass |
| **assert** | Assert describes a predicate (a true–false statement) placed in a Java program to indicate that the developer thinks that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results,  which typically causes execution to abort. |
| **boolean** | A data type that can hold True and False values only |
| **break** | A control statement for breaking out of loops |
| **byte** | A data type that can hold 8-bit data values |
| **case** | Used in switch statements to mark blocks of text |
| **catch** | Catches exceptions generated by try statements |
| **char** | A data type that can hold unsigned 16-bit Unicode characters. |
| **class** | Declares a new class. |
| **continue** | Sends control back outside a loop. |
| **default** | Specifies the default block of code in a switch statement |
| **do** | Starts a do-while loop |
| **double** | – A data type that can hold 64-bit floating-point numbers |
| **else** | Indicates alternative branches in an if statement |
| **enum** | – A Java keyword used to declare an enumerated type.  ***Enumerations extend the base class.*** |
| **extends** | Indicates that a class is derived from another class or interface |
| **final** | Indicates that a variable holds a constant value or  that a method will not be overridden |
| **finally** | Indicates a block of code in a try-catch structure that will always be executed |
| **float** | A data type that holds a 32-bit floating-point number |
| **for** | Used to start a for loop |
| **if** | Tests a true/false expression and branches accordingly |
| **implements** | Specifies that a class implements an interface |
| **import** | References other classes |
| **instanceof** | Indicates whether an object is an instance of a specific class or implements an interface |
| **int** | A data type that can hold a 32-bit signed integer |
| **interface** | Declares an interface |
| **long** | A data type that holds a 64-bit integer |
| **native** | Specifies that a method is implemented with **native** (**platform-specific)** code |
| **new** | Creates new objects |
| **null** | Indicates that a reference does not refer to anything |
| **package** | Declares a Java package |
| **private** | An access specifier indicating that a method or variable may be accessed only in the class it’s declared in |
| **protected** | An access specifier indicating that a method or variable may only be accessed in the class it’s declared in (or a subclass of the class it’s declared in or other classes in the same package) |
| **public** | An access specifier used for classes, interfaces, methods,  and variables indicating that an item is accessible  throughout the application (or where the class that defines it is accessible) |
| **return** | Sends control and possibly a return value back from a called method |
| **short** | A data type that can hold a 16-bit integer |
| **static** | Indicates that a variable or method is a class method  (rather than being limited to one particular object) |
| **strictfp** | A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability. |
| **super** | Refers to a class’s **base class** (used in a method or class constructor) |
| **switch** | A statement that executes code based on a test value |
| **Synchronized** | Specifies critical sections or methods in **multithreaded code** |
| **this** | Refers to the **current object** in a method or constructor |
| **throw** | Creates an exception |
| **throws** | Indicates what exceptions may be thrown by a method |
| **transient** | Specifies that a variable is not part of an object’s persistent state |
| **try** | Starts a block of code that will be tested for exceptions |
| **void** | Specifies that a method does not have a return value |
| **volatile** | Indicates that a variable may change asynchronously |
| **while** | Starts a while loop |
| The keywords **const** and **goto** are reserved, even they are not currently in use. | |

***OPERATORS***

***Arithmetic Operators***

Arithmetic operators are used in **mathematical expressions** in the same way that they are used in algebra.

***Relational Operators / Comparison***

**Bitwise Operators**

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation.

***Logical Operators***

***Assignment Operators***

**Miscellaneous Operators**

******

***Note:***

***Signed and Unsinged in Java***

*Signed variables, such as signed integers will allow you to represent numbers both in the positive and negative ranges.*

*Unsigned variables, such as unsigned integers, will only allow you to represent numbers in the positive.*

*Unsigned and signed variables of the same type (such as byte and int) both have the same range (range of 65,536 and 256 numbers, respectively), but unsigned can represent a larger magnitude number than the corresponding signed variable.*

For example, an unsigned byte can represent values from 0 to 255, while signed byte can represent -128 to 127. (Adding 128 + 127 = 255).

***Why Does Java have Signed and Unsigned Data Types?***

**Java only supports signed types (except char)** because it was assumed that one type was simpler for beginners to understand than having two types for each size.

***Escape sequence in Java***

**Escape characters** (also called **escape sequences** or **escape codes)** in general are used to **signal an alternative interpretation of a series of characters.**

In Java, **a character preceded by a backslash (\) is an escape sequence** and has **special meaning to the java compiler.**

**Escape sequences available in java** are:

**\t** - Insert a **tab** in the text at this point.  
**\b** - Insert a **backspace** in the text at this point.  
**\n** - Insert a **newline** in the text at this point.  
**\r** - Insert a **carriage return** in the text at this point.  
**\f** - Insert a **formfeed** in the text at this point.  
**\'** - Insert a **single quote** character in the text at this point.  
**\"** - Insert a **double quote** character in the text at this point.  
**\\** - Insert a **backslash** character in the text at this point.

***An escape sequence is a single character.***

***Note:***

***Carriage return****means to****return****to the beginning of the current line without advancing downward.*

***formfeed*** *is used for printing characters after it from new line starting just below previous character.*

1. ***MODIFIERS:***

***Modifiers are keywords that added to definitions (signatures) to change their meanings.***

*Note: there are two parts in a class and its members:*

*1. definitions (signatures)*

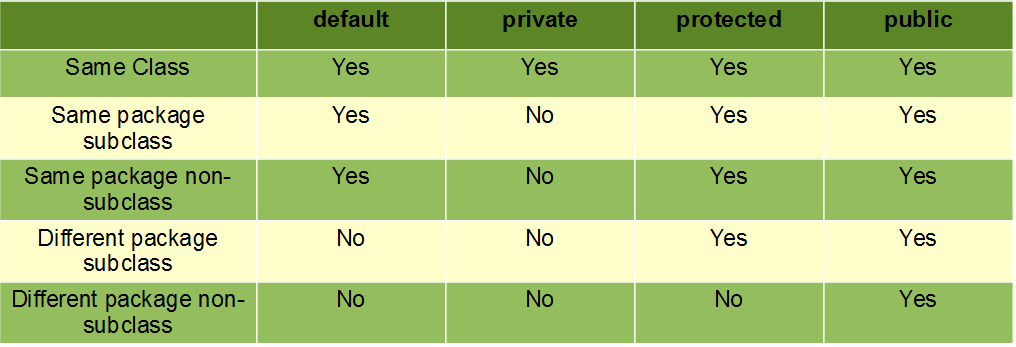
*2. Structure / body*

*Modifier types:*

***1. Access 2. Non-access***

***Access Modifiers:***

***Access modifiers*** *in Java helps to* ***restrict the scope*** *of a class, constructor , variable , method or data member.*



**Default:**

When no access modifier is specified for a class, method or data member –

It is said to be having the default access modifier by default.

The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are *accessible only within the* ***same package.***

**Private:**

The private access modifier is specified using the keyword **private**.

The methods or data members declared as private are *accessible only within* ***the class****in which they are declared.*

Any other class of same package will not be able to access these members.

***Note*:**

*Top level Classes or interface cannot be declared as private because private means “only visible within the enclosing class”.*

***protected*:**

The protected access modifier is specified using the keyword protected.

The methods or data members declared as protected are *accessible within* ***same package*** *or* ***sub classes in different package.***

***public*:**

The public access modifier is specified using the keyword public.

The public access modifier has the widest scope among all other access modifiers.

Classes, methods or data members which are declared as public are accessible from **every where in the program.**

There is no restriction on the scope of a public data members.

***Note:***

* *Avoid* ***public*** *fields except for* ***constants****.*

**Non-access modifiers in Java:**

**Non-access modifiers** do not change the accessibility of variables and methods, but they *do* ***provide them special properties.***

*Non-access modifiers can* ***alter the behaviour of elements in java.***

*There are* ***two levels*** *on which non-access modifiers can be applied,*

***Classes*** *and its* ***members****.*

**1.class:** two non-access modifiers:

**final**

**abstract**

***2. members***

*final*

*abstract*

*static*

*synchronized*

*native*

*transient*

*volatile*

*strictfp*

***1.final***

**final Classes**

When a class is set to final, *it* ***cannot be extended by any other class.***

*It is a* ***final segment of class hierarchy of the program.***

***finalclass Car{***

*//some code*

}

**final methods**

*A method when set to final* ***cannot be overridden by any child class.***

The final modifier prevents a method from being modified in a childclass.

class Car{

public final void displaySpeed(){

*// body of method*

}

}

***final variables***

When a variable is set to final, its **value cannot be changed.**

**Final variables are like constants.**

***A reference variable declared final can never be reassigned to refer to an different object.***

However **the data within the object can be changed.**

So **the state of the object can be changed but not the reference.**

***A final variable can be explicitly initialized only once.***

Most often *with variables,* ***final*** *is used along with* ***static*** *to make the* ***constant*** *a class variable.*

class Car{

public static final int SPEED =60;

public final void displaySpeed(){

*// body of method*

}

}

***Note:***

***You can instantiate a final class just like any other non-abstract class.***

***The only limitation is that you cannot create subclasses of a final class.***

***2.abstract Classes***

***If a class is declared as abstract, then that class can never be instantiated.***

*The sole purpose for using* ***abstract*** *is for the* ***class to be extended.***

***Note:***

***A class cannot be declared both abstract and final.***

If a class contains one or more abstract methods then the class should be declared abstract, or else a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

*An abstract class does not need to contain abstract methods.*

**abstract methods**

*Abstract methods are those methods which does* ***not*** *have a* ***body*** *but* ***only*** *a* ***signature.***

The method body is provided by sub class.

* ***Abstract methods can never be final.***
* ***Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.***

***Note*:**

*The abstract method ends with a semicolon.*

public abstract class SuperCar{

abstract void displaySpeed();*//abstract method*

***static***

***static variables***

If you declare a variable as static then it becomes static variable.

***Static variables are also called as Class variables*** *as it belongs to the class rather than objects(instances).*

***static methods***

Static methods are conceptually the same as static variables.

It is a method which belongs to the class and not to the object(instance).

***A static method can access only static data.***

It **cannot access non-static data (instance variables)**

***Synchronized***

*When a* ***method*** *is* ***synchronized*** *it can be* ***accessed*** *by only* ***one thread at a time.***

This can ***only be used to methods.***

*The synchronized modifier can be* ***applied with any of the four access level modifiers.***

public synchronized void display Speed(){}

***native***

*is applied to a* ***method*** *to indicate that the* ***method is implemented in native code using JNI(Java Native Interface).***

*It marks a method, that it will be implemented in other languages, not in Java.*

Native methods were used in the past to write ***performance critical sections*** but with Java getting faster this is now less common.

*Native methods are currently needed when need* ***to call a library from Java that is written in other language.***

*need to* ***access system or hardware resources*** *that are* ***only reachable from the other language***

***transient***

*transient is a Java keyword which marks a* ***member variable******not to be serialized*** *when it is* ***persisted to streams of bytes.***

Variables may be marked transient to indicate that they are not part of the persistent state of an object.

public transient int speed =60;*//transient value*

***volatile***

*Declaring a volatile Java variable means, the value of this variable will* ***never be cached thread-locally,*** *all reads and writes will* ***go straight to "main memory"***

**public volatile int speed =60**;*//volatile value*

***strictfp***

*strictfp is a keyword in the Java programming language that* ***restricts floating-point calculations to ensure portability.***

The strictfp command was introduced into Java with the Java virtual machine (JVM) version 1.2 and is available for use on all currently updated Java VMs.

***Strictfp ensures that you get exactly the same results from your floating point calculations on every platform.***

If strictfp is not used, the JVM implementation is free to use extra precision where available.

***strictfp can be used on classes, interfaces and non-abstract methods.***

When applied to a method, it causes all calculations inside the method to use strict floating-point math.

When applied to a class, all calculations inside the class use strict floating-point math.

public strictfp class MyFPclass{

*// ... contents of class here ...*

1. **CONTROL STATEMENTS:**

***the statement is the instruction given to the computer that performs specific types of work.*** *They can make* ***decisions*** *and* ***repetitive tasks.***

The statements are of the following **three types.**

|  |  |
| --- | --- |
| **Statement** | *Description* |
| **Type declaration** Statement | *To declare the* ***type of variables*** *being used in the program.* |
| **Arithmetic** Statement | *To complete the* ***arithmetic operation*** *between* ***constants*** *and* ***variables****.* |
| **Control** Statement | *To* ***control the sequence of execution of different statements of the program.*** |

*A control statement in java is a statement that determines whether the other statements will be executed or not.*

It controls the flow of a program.

There are **four types of control statements:**

1. ***Sequence Control Statement.***
2. ***Selection or Decision Control Statement.***
3. ***Repetition or Loop Control Statement.***
4. ***Case Control Statement.***

***Sequence Control Statement:***

Sequence control structure refers to the **line-by-line execution** by which statements are executed sequentially, in the same order in which they appear in the program.

***Selection or Decision Control Statement***

The decision control statements are the decision making statements that decides the **order of execution of statements based on the conditions.**

In the decision making statements the programmer specify which conditions are to be executed or tested with the statements to be executed if the **condition is true or false.**

***Repetition or Loop Control Statement.***

A repetition statement (also called a looping statement or a loop) allows you to specify that a program should **repeat an action** while some **condition remains true.**

***Case Control Statement***

A case or switch statement is a type of **selection control mechanism** used to allow the **value of a variable or expression** to **change the control flow of program execution via multiway branch.**

**Java If-else Statement**

The Java ***if*** *statement* is used to **test the condition.**

It checks **boolean condition: *true* or *false*.**

There are various types of if statement in java.

**1. if statement**

It executes the if block if condition is true.

**public** **class** IfCondition {

**static** **int** x=10;

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

**if**(x%2 ==0) {

System.out.println(x+ " : is an even number");

}

}

}

**2. if-else statement**

It executes the if block if condition is true otherwise else block is executed.

public class IfElse {

static int x = -10;

public static void main(String[] args) {

if (x >= 0) {

System.out.println(x + " :is a positive integer");

}

else {

System.out.println(x + " :is a negative integer");

}

}

}

***ternary operator (? :)*** *to perform the task of if...else statement.*

It is a **shorthand way** to check the condition.

If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

***Note:***

***Java ternary operator****is the only conditional****operator****that takes* ***three operands.***

**public** **class** TernaryOperator {

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

**int** n1=30,n2=20;

**int** max;

max = (n1>n2) ? n1:n2;

System.out.println("Maximum of the given " + n1 + " and "+ n2 +" is "+ max);

}

}

**3.if-else-if ladder**

*executes one condition from multiple statements.*

**public** **class** IfElseIf {

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

**int** number = -10;

// checks if number is greater than 0

**if** (number > 0) {

System.out.println("The number is positive.");

}

// checks if number is less than 0

**else** **if** (number < 0) {

System.out.println("The number is negative.");

} **else** {

System.out.println("The number is 0.");

}

}

}

**4. nested if** statement

The nested if statement represents the if block within another if block.

Here, the inner if block condition executes only when outer if block condition is true.

**public** **class** NestedIf {

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

**int** age=20;

String country="INDIA";

**if**(age >=18) {

**if**(country == "INDIA") {

System.out.println("Eligible to vote ");

}

}

}

}

**Java Switch Statement**

The Java switch statement **executes one statement from multiple conditions.**

It is like **if-else-if** ladder statement.

The switch statement works with **byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long.**

Since Java 7, you can use strings in the switch statement.

In other words, ***the switch statement tests the equality of a variable against multiple values.***

***Note:***

*There can be*one or N number of case values*for a switch expression.*

The **case value** must be of **switch expression type only.**

The **case value must be literal or constant.** It **doesn't allow variables.**

The **case values must be unique.**

In case of **duplicate value, it renders compile-time error.**

**Each case statement** can have a **break statement which is optional.**

When **control reaches to the break statement, it jumps the control after the switch expression.**

If a break statement is not found, it executes the next case.

The case value can have a **default label** which is **optional**.

**Syntax:**

**switch**(expression){

**case** value1:

 //code to be executed;

**break**;  //optional

**case** value2:

 //code to be executed;

**break**;  //optional

......

**default**:

 code to be executed **if** all cases are not matched;

}

***Note:***

***Java Switch Statement is fall-through****:*

***It means it executes all statements after the first match if a break statement is not present.***

**public** **class** SwitchExample {

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

**int** num = 100;

**switch** (num) {

**case** 10:

System.out.println(" 10");

**break**;

**case** 20:

System.out.println(" 20");

**break**;

**case** 30:

System.out.println(" 30");

**break**;

**default**:

System.out.println("number not in 10,20 or 30");

}

}

}

**Java Switch Statement with String**

Java allows us to use strings in switch expression since Java SE 7.

The **case statement** should be **string literal.**

**public** **class** SwitchString {

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

String country = "India";

**switch** (country) {

**case** "India":

System.out.println("India");

**break**;

**case** "Australia":

System.out.println("Australia");

**break**;

**case** "USA":

System.out.println("USA");

**break**;

**default**:

System.out.println(" country not in the required list");

}

}

}

**Java Nested Switch Statement**

We can use switch statement inside other switch statement in Java.

It is known as nested switch statement.

**public** **class** SwitchNested {

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

{

//Declaring a variable for switch expression

**int** year = 3;

**int** marks = 80;

**switch**(year) //Switch expression

{

//Case statements

**case** 1:

System.out.println("First year students are not eligible for scholarship ");

**break**;

**case** 2:

System.out.println("Second year students are not eligible for scholarship");

**break**;

**case** 3:

**switch**(marks)

{

**case** 50:

System.out.println("You are not eligible for scholarship");

**break**;

**case** 80:

System.out.println("Congrats!!!!! you are eligible for scholarship");

**break**;

}

**break**;

//Default case statement

**default**: System.out.println("Please enter valid year");

}

}

}

***Java Enum in Switch Statement***

**enum** Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

**public** **class** EnumSwitch {

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

**for** (Day d : Day.values()) {

printTodaysThought(d);

}

}

**public** **static** **void** printTodaysThought(Day theDay) {

**switch** (theDay) {

**case** MONDAY:

**case** TUESDAY:

**case** WEDNESDAY:

**case** THURSDAY:

System.out.println("Working for the man :)");

**break**;

**case** FRIDAY:

System.out.println("TGIF ");

**break**;

**case** SATURDAY:

**case** SUNDAY:

System.out.println("Ahh, the weekend ...");

**break**;

**default**:

System.out.println("What day is it?");

;

}

}

}

***Java Wrapper in Switch Statement:***

Java allows us to use **four wrapper classes: Byte, Short, Integer and Long in switch statement.**

***Loops in Java:***

*loops are used to* ***execute a set of instructions/functions repeatedly*** *when some* ***conditions become true.***

|  |  |  |  |
| --- | --- | --- | --- |
| 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); |

**Java For Loop**

There are **three types of for loops in java.**

1. ***Simple For Loop***
2. ***For-each or Enhanced For Loop***
3. ***Labeled For Loop***

***Java Simple For Loop***

**Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

**Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

**Statement**: The statement of the loop is executed each time until the second condition is false.

**Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

**public** **class** SimpleForLoop {

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

**for** (**int** i = 0; i <=10; i++) {

System.out.println(i);

}

}

}

***Java Nested For Loop***

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

**public** **class** NestedForLoop {

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

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

System.out.println("Outer loop iteration " + i);

**for** (**int** j = 1; j <= 5; ++j) {

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

}

}

}

}

***Java for-each Loop***

The for-each loop is used to **traverse array or collection** in java.

It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on **elements basis** not index. It returns element one by one in the defined variable.

**public** **class** ForEach {

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

**int** arr[] = {1,2,3,4};

**for** (**int** i : arr) {

System.out.println(i);

}

}

}

***Java Labeled For Loop***

We can have a name of each Java for loop.

To do so, we use **label before the for loop.**

It is useful if we have **nested for loop** so that we can **break/continue specific for loop.**

*Usually, break and continue keywords breaks/continues the innermost for loop only.*

***Syntax:***

labelname:

for(initialization;condition;incr/decr){

//code to be executed

}

**public** **class** LabelledLoop {

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

{

**int** i,j;

loop1: **for**(i=1;i<=10;i++)

{

System.out.println();

loop2: **for**(j=1;j<=10;j++)

{

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

**if**(j==5)

**break** loop1; //Statement 1

}

}

}}

***Java Break Statement***

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java ***break*** is used to **break loop** or **switch** statement.

*It breaks the current flow of the program at specified condition.*

In case of **inner loop, it breaks only inner loop.**

We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

jump-statement;

break;

**Java Break Statement with Loop**

**public** **class** BreakExample {

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

    //using for loop

**for**(**int** i=1;i<=10;i++){

**if**(i==5){

            //breaking the loop

**break**;

        }

        System.out.println(i);

    }

}

}

**Java Break Statement with Inner Loop**

It breaks inner loop only if you use break statement inside the inner loop.

**public** **class** BreakExample2 {

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

            //outer loop

**for**(**int** i=1;i<=3;i++){

                    //inner loop

**for**(**int** j=1;j<=3;j++){

**if**(i==2&&j==2){

                            //using break statement inside the inner loop

**break**;

                        }

                        System.out.println(i+" "+j);

                    }

            }

}

}

***Java Break Statement with labelled For Loop***

We can use break statement with a label. This feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer loop or inner.

**public** **class** BreakExample3 {

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

            aa:

**for**(**int** i=1;i<=3;i++){

                    bb:

**for**(**int** j=1;j<=3;j++){

**if**(i==2&&j==2){

                            //using break statement with label

**break** aa;

                        }

                        System.out.println(i+" "+j);

                    }

            }

}

}

**Java Break Statement in while loop**

**public** **class** BreakWhileExample {

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

    //while loop

**int** i=1;

**while**(i<=10){

**if**(i==5){

**break**;//it will break the loop

        }

        System.out.println(i);

        i++;

    }

}

}

**Java Break Statement in do-while loop**

public class BreakDoWhileExample {

public static void main(String[] args) {

    //declaring variable

    int i=1;

    //do-while loop

    do{

        if(i==5){

           break;//it will break the loop

        }

        System.out.println(i);

        i++;

    }while(i<=10);

}

}

***Java continue statement:***

The continue statement is used in loop control structure when you need to **jump to the next iteration of the loop immediately.**

**public** **class** ContinueLoop {

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

**int**[] intArray = { 1, 2, 3, 4, 5, 6, 7 };

// we want to process only even entries

**for** (**int** i : intArray) {

**if** (i % 2 != 0)

**continue**;

System.out.println("Processing entry " + i);

}

}

}

*The Java continue statement is used* ***to continue the loop.***

It continues the current flow of the program and skips the remaining code at the specified condition.

**In case of an inner loop, it continues the inner loop only.**

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

**Java Continue Statement with Inner Loop**

It continues inner loop only if you use the continue statement inside the inner loop.

**public** **class** ContinueInner {

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

**int** intx;

String[] arrstr = **new** String[3];

arrstr[0] = "a";

arrstr[1] = "b";

arrstr[2] = "c";

**for** (String currElement : arrstr) {

// Starting inner array

**for** (intx = 1; intx <= 3; intx++) {

// Should skip 2

**if** (intx == 2) {

**continue**;

}

System.out.println(intx);

}

System.out.println("The String array: " + currElement);

}

}

}

***Java Continue Statement with Labelled For Loop***

We can use continute statement with a label. This feature is introduced since JDK 1.5. So, we can continue any loop in Java now whether it is outer loop or inner.

**public** **class** ContinueLabel {

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

**int**[][] intArr = { { 1, -2, 3 }, { 0, -3 }, { 1, 2, 5 }, { 9, 2, 5 } };

process: **for** (**int** i = 0; i < intArr.length; i++) {

**boolean** allPositive = **true**;

**for** (**int** j = 0; j < intArr[i].length; j++) {

**if** (intArr[i][j] < 0) {

allPositive = **false**;

**continue** process;

}

}

**if** (allPositive) {

// process the array

System.out.println("Processing the array of all positive ints. " + Arrays.toString(intArr[i]));

}

allPositive = **true**;

}

}

}

**Java Continue Statement in while loop**

**public** **class** ContinueWhileLoop {

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

**int**[] intArray = { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

**int** i = 0;

**while** (i < 10) {

**if** (i % 3 != 0) {

i++;

**continue**;

}

System.out.println("Processing Entry " + intArray[i]);

i++;

}

}

}

***Java Continue Statement in do-while loop***

**public** **class** ContinueDoWhile {

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

**int**[] intArray = { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

**int** i = 0;

**do** {

**if** (i % 3 != 0) {

i++;

**continue**;

}

System.out.println("Processing Entry " + intArray[i]);

i++;

} **while** (i < 10);

}

}

1. **CLASS AND ITS TYPES**

* A *class* in Java is a **template** that is used to **create** and **define objects**, object **data types**, and **methods**.
* ***Classes as a whole are categories and objects are items within each category.***
* *A* ***class declaration*** *constitutes of the following parts:*
* *Modifiers*
* Class name
* Keywords
* The class body within curly brackets {}
* ***Classes are reference types that hold the object created dynamically in a heap.***
* ***All classes****have a* ***base type****of* ***System.Object.***
* *The****default access modifier*** *of a****class****is* ***Internal****.*
* *The****default access modifier******of methods and variables is Private.***

***Note:***

***Internal access modifier:***

*Internal members are accessible only within files in the same assembly*

*(. dll). In other words, access is limited exclusively to classes defined within the current* ***project assembly.***

* ***Classes are accessible within the same Java Project.***

***Types of Classes in java:***

1. **POJO Class**

POJO stands for **“Plain Old Java Object”.**

A class which contains **only private variables** and **public** **setter and getter methods** to use those variables is called POJO class.

It is **a pure data structure** that has **fields** and may override some methods from **Object** (e.g. equals) or some other interface like **serializable** but does ***not have the behaviour of its own****.*

**Properties of POJO class –**

**class should be public**

***Public setter*** *and* ***getter*** *methods are a must while writing a POJO class.*

***All instance variables should be private.***

*It should* ***not extend pre-specified classes.***

*It should* ***not implement pre-specified interfaces.***

*Should* ***not contain pre-specified annotations.***

*It* ***may not have a no-argument constructor.***

**2. Static Class**

***In Java, static is a keyword used to describe how objects are managed within the memory*.**

***A static object belongs specifically to the class, instead of instances of that class.***

*The sole purpose of the class is to provide blueprints of its inherited classes.*

***A static class can contain static members only.***

***cannot create an object for a static class.***

***Note:***

We cannot use the **static** keyword with a **class** unless it is an inner **class**. A **static** inner **class** is a nested **class** which is a **static** member of the outer **class**. It can be accessed without instantiating the outer **class**, using other **static** members.

***Note:***

differences between **static** and **non-static nested classes?**

* A static nested class may be instantiated without instantiating its outer class.
* Inner classes can access both static and non-static members of the outer class. A static class can access only the static members of the outer class.

**3. Concrete Class**

*Any* ***normal class*** *which does* ***not have*** *any* ***abstract method or a class*** having an **implementation for all of its methods** is basically a concrete class.

They ***cannot have any unimplemented methods.***

A concrete class can extend its parent class, an abstract class or implement an interface if it implements all their methods. It is a **complete class** that can be instantiated.

**4. Abstract Class**

An abstract class is declared with an **abstract keyword** and have **zero or more abstract methods.**

These classes are **incomplete classes**, therefore, to use an abstract class we strictly need to extend the abstract classes to a concrete class.

*It* ***can have constructors and static methods as well.***

*It* ***can have final methods*** *which will force the subclass to keep the body of the method unhung.*

**5. Final Class**

Once a variable, method or a class is declared as final, it’s value remains the same throughout.

The *final* keyword in a method declaration indicates that the method cannot be overridden by any subclasses i.e., ***a class that has been declared final cannot be subclassed.***

This helps a lot while **creating an immutable class** like the String class.

***A class cannot make a class immutable without making it final.***

**6. Inner class**

*Inner class means the class which is a member of another class.*

There are four types of **inner classes** in java.

*1)* ***Nested*** *Inner class  
2)* ***Method Local*** *inner classes  
3)* ***Anonymous*** *inner classes* *4)* ***Static******nested*** *classes*

**1. Nested Inner class**

It *can access any private instance variable of an outer class.*

Like any other instance variable, we can have *access modifiers*

private, protected, public and default modifier.

**2. Method Local inner classes**

An inner class can be declared within a method of an outer class.

**3. Anonymous inner classes**

In Java, a class can contain another class known as nested class.

It's possible to create a nested class without giving any name.

**A nested class that doesn't have any name is known as an anonymous class.**

*An anonymous class must be defined inside another class.* Hence, it is also known as an anonymous inner class. Its syntax is:

class outerClass {

// defining anonymous class

object1 = new Type(parameterList) {

// body of the anonymous class

};

}

Anonymous classes usually extend subclasses or implement interfaces.

Here, Type can be a superclass that an anonymous class extends,

an interface that an anonymous class implements

The above code creates an object, object1, of an anonymous class at runtime.

Note:

Anonymous classes are defined inside an expression. So, the semicolon is used at the end of anonymous classes to indicate the end of the expression.

**Difference between Normal/Regular class and Anonymous Inner class:**

A normal class can implement any number of interfaces but *anonymous inner class can implement only one interface at a time.*

A regular class can extend a class and implement any number of interface simultaneously. But *anonymous Inner class can extend a class or can implement an interface but not both at a time.*

For regular/normal class, we can write any number of constructors but we cant write any constructor for anonymous Inner class because anonymous class does not have any name and while defining constructor class name and constructor name must be same.

**Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class**  
Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

An anonymous class has access to the members of its enclosing class.

An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.

Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.

Anonymous classes also have the same restrictions as local classes with respect to their members:

We cannot declare static initializers or member interfaces in an anonymous class.

An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

Fields

Extra methods (even if they do not implement any methods of the supertype)

Instance initializers

Local classes

1. **Static nested classes**

Static nested classes are like a static member of the outer class.

**7. Wrapper class in Java:**

*In object-oriented programming, a wrapper class is a class that encapsulates types, so that those types can be used to create object instances and methods in another class that need those types.*

*A Wrapper class is a class whose object wraps or contains a primitive data types.*

When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

*wrapper classes provide a way to use primitive data types (int, char, short, byte, etc) as objects.*

These wrapper classes come under java.util package.

**Note:**

*String is not a wrapper class, simply because there is no parallel primitive type that it wraps.*

A string is a representation of a char sequence but not necessarily a 'wrapper'.

**Need of Wrapper Classes**

They convert primitive data types into objects.

Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

*The classes in java.util package handles only objects* and hence wrapper classes help in this case also.

Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

An object is needed to support synchronization in multithreading.

Primitive Data types and their Corresponding Wrapper class



***Autoboxing is the automatic conversion of the primitive types into their corresponding object wrapper classes.***

For example, converting an int to an Integer , a char to a Character

**Unboxing:**

It is just the reverse process of autoboxing.

**Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing.**

For example – conversion of Integer to int, Long to long, Double to double etc.

**METHODS:**

*A method is a block of code which only runs when it is called.*

Methods are used to perform certain actions, and they are also known as functions.

A method is a collection of statements that perform some specific task and return the result to the caller.

A method can perform some specific task without returning anything.

Methods allow us to reuse the code without retyping the code.

*In Java, every method must be part of some class.*

In general, method declarations has six components :

Modifier-:

Defines access type of the method i.e. from where it can be accessed in application.

The return type :

The data type of the value returned by the method or void if does not return a value.

Method Name

Parameter list :

Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis.

If there are no parameters, must use empty parentheses ().

Exception list :

The exceptions you expect by the method can throw, you can specify these exception(s).

Method body :

it is enclosed between braces.

The code need to be executed to perform intended operations.

Method signature:

It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters).

*The return type and exceptions are not considered as part of it.*

Calling a method

The method needs to be called for using its functionality.

A method returns to the code that invoked it when:

It completes all the statements in the method

It reaches a return statement

Throws an exception

Types of methods in Java:

There are three main types of methods:

interface methods – used in Interface and they are all abstract.

constructor methods– A constructor method is a special function that creates an instance of the class.

implementation methods– methods having both signature and body. These methods implements Interface/abstract methods.

|  |  |
| --- | --- |
| Method classification | |
| built in (libraries) | user defined |
| Static | Non static |
| return type | non return type |
| default | parameterised |

CONSTRUCTORS:

Constructor is a block of code that *initializes the newly created object.*

Constructor has same name as the class.

It is called when an instance of the class is created.

*At the time of calling constructor, memory for the object is allocated in the memory.*

It is a *special type of method which is used to initialize the object.*

*Every time an object is created using the new() keyword, at least one constructor is called.*

*It calls a default constructor if there is no constructor available in the class.*

*In such case, Java compiler provides a default constructor by default.*

**Note:**

It is called constructor because it constructs the values at the time of object creation.

It is not necessary to write a constructor for a class.

It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

*Constructor name must be the same as its class name*

*A Constructor must have no explicit return type*

*A Java constructor cannot be abstract, static, final, and synchronized*

Note:

We can use access modifiers while declaring a constructor.

It controls the object creation.

In other words, we can have private, protected, public or default constructor in Java.

There are two types of constructors in Java:

1. Default Constructor - doesn't have any parameter.

2. Parameterized Constructor

OOPS CONCEPTS:

Polymorphism:

*Poly = Many / Morph= forms*

a concept by which we can perform a single action in different ways.

*2 types of Polymorphism:*

*Compile Time – Method Over loading*

Methods having the same name within the same class which are having either different number of arguments or Same number of arguments but with different data types are said to be method overloaded and this process is known as “Method Overloading” or “Compile time Polymorphism”

Note:

Method overloading has nothing to do with return type of the methods.

Only one class is enough for “Method Overloading”.

*Run Time – Method Overriding*[**Dynamic Method Dispatch**]

Parent and child class are required

Child class will inherit all the properties of the Parent class.

“extends”is the keyword used to inherit a Class.

At least 2 classes are required for “Method Overriding”.

When two methods having same name and same number of arguments and same data type then the method in the Child class will be executed by

over riding the method in the Parent class.

This process is known as Method Overriding” or “Run Time polymorphism”

Upcasting:

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.

class A{}

class B extends A{}

A a=new B();//upcasting

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

Java Runtime Polymorphism with Multilevel Inheritance

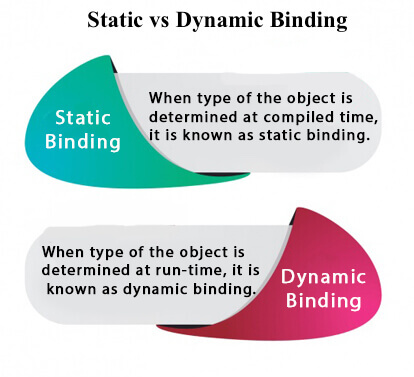
Static Binding and Dynamic Binding

*Connecting a method call to the method body is known as binding.*

There are two types of binding :

Static Binding (also known as Early Binding).

Dynamic Binding (also known as Late Binding).



variables have a type

Each variable has a type, it may be primitive and non-primitive.

**int** data=30;

Here data variable is a type of int.

References have a type

**class** Dog{

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

  Dog d1;//Here d1 is a type of Dog

 }

}

Objects have a type

|  |
| --- |
| An object is an instance of particular java class,but it is also an instance of its superclass. |

**class** Animal{}

**class** Dog **extends** Animal{

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

  Dog d1=**new** Dog();

 }

}

|  |
| --- |
| Here d1 is an instance of Dog class, but it is also an instance of Animal. |

static binding

If there is any private, final or static method in a class, there is static binding.

**class** Dog{

**private** **void** eat(){System.out.println("dog is eating...");}

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

  Dog d1=**new** Dog();

  d1.eat();

 }

}

Dynamic binding

Example of dynamic binding

**class** Animal{

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

}

**class** Dog **extends** Animal{

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

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

  Animal a=**new** Dog();

  a.eat();

 }

}

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.

So compiler doesn't know its type, only its base type.

2. Inheritance:

The process of obtaining/inheriting the properties of a Parent Class / Interface from a child class.

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. ... 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.

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.

Types of Inheritance :

Single Level

Class B

Class A

Hierarchical

A

B

C

3.Multilevel

A

B

C

….

Note:

4 ways of creating Object in Inheritance:

Parent obj = new Parent(); // Only Parent properties

Parent obj = new Child(); // Only Parent properties

Child obj = new Child(); // Both Parent and Child properties

Child obj = new Parent(); <Java Does not support>

Multiple <Java does not Support> -- “Diamond issue”

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.

**Aggregation in Java**

If a class have an entity reference, it is known as Aggregation.

Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

class Employee{

int id;

String name;

Address address;//Address is a class

...

}

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

For Code Reusability.

Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

**public** **class** Address {

String city,state,country;

**public** Address(String city, String state, String country) {

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

}

Emp.java

**public** **class** Emp {

**int** id;

String name;

Address address;

**public** Emp(**int** id, String name,Address address) {

**this**.id = id;

**this**.name = name;

**this**.address=address;

}

**void** display(){

System.out.println(id+" "+name);

System.out.println(address.city+" "+address.state+" "+address.country);

}

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

Address address1=**new** Address("gzb","UP","india");

Address address2=**new** Address("gno","UP","india");

Emp e=**new** Emp(111,"varun",address1);

Emp e2=**new** Emp(112,"arun",address2);

e.display();

e2.display();

}

}

Output:111 varun

gzb UP india

112 arun

gno UP india

Abstraction

Methods having only definition but not body.

Level of abstraction in a class is : 0 – 100%

*Data Abstraction is the property by virtue of which only the essential details are displayed to the user.*

The trivial or the non-essentials units are not displayed to the user.

In java, *abstraction is achieved by interfaces and abstract classes.*

Ex: A car is viewed as a car rather than its individual components.

*Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.*

The *properties and behaviours of an object differentiate it from other objects of similar type* and also help in *classifying/grouping the objects.*

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

We can *achieve 100% abstraction using interfaces.*

**Abstract classes and Abstract methods :**

An abstract class is a class that is declared with abstract keyword.

An abstract method is a method that is declared without an implementation.

An abstract class may or may not have all abstract methods. Some of them can be concrete methods

A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.

Any class that contains one or more abstract methods must also be declared with abstract keyword.

*There can be no object of an abstract class.*

*That is, an abstract class cannot be directly instantiated with the***new operator***.*

*An abstract class can have parametrized constructors and default constructor is always present in an abstract class.*

|  |
| --- |
| Abstract class Shape  {      String color;        // these are abstract methods      abstract doublearea();      publicabstractString toString();        // abstract class can have constructor      public Shape(String color) {          System.out.println("Shape constructor called");          this.color = color;      }        // this is a concrete method      public String getColor() {          returncolor;      }  }  class Circle extendsShape  {      double radius;        public Circle(String color,doubleradius) {            // calling Shape constructor          super(color);          System.out.println("Circle constructor called");          this.radius = radius;      }        @Override      doublearea() {          returnMath.PI \* Math.pow(radius, 2);      }        @Override      publicString toString() {          return"Circle color is "+ super.color +                         "and area is : "+ area();      }    }  classRectangle extendsShape{        doublelength;      doublewidth;        publicRectangle(String color,doublelength,doublewidth) {          // calling Shape constructor          super(color);          System.out.println("Rectangle constructor called");          this.length = length;          this.width = width;      }        @Override      doublearea() {          returnlength\*width;      }        @Override      publicString toString() {          return"Rectangle color is "+ super.color +                             "and area is : "+ area();      }    }  publicclassTest  {      publicstaticvoidmain(String[] args)      {          Shape s1 = newCircle("Red", 2.2);          Shape s2 = newRectangle("Yellow", 2, 4);            System.out.println(s1.toString());          System.out.println(s2.toString());      }  } |

Output:

Shape constructor called

Circle constructor called

Shape constructor called

Rectangle constructor called

Circle color is Redand area is : 15.205308443374602

Rectangle color is Yellowand area is : 8.0

**Encapsulation vs Data Abstraction**

*Encapsulation is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).*

While *encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.*

**Advantages of Abstraction**

It reduces the complexity of viewing the things.

Avoids code duplication and increases reusability.

Helps to increase security of an application or program as only important details are provided to the user.

Interface

*An interface is a reference type in Java.*.

It is a collection of abstract methods.

A class implements an interface, thereby inheriting the abstract methods of the interface. Along with abstract methods, an *interface may also contain constants, default methods, static methods, and nested types.*

Interfaces in Java:

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

*Interfaces specify what a class must do and not how.*

*It is the blueprint of the class.*

An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.

If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Syntax :

interface interface\_name {

// declare constant fields

// declare methods that abstract

// by default.

}

*To declare an interface, use****interface****keyword.*

It is used to provide total abstraction.

That means all the methods in an interface are declared with an empty body and are publicand *all fields are public, static and final by default.*

A class that implement interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Why do we use interface ?

It is used to *achieve total abstraction.*

Since java does not support multiple inheritance in case of class, but by *using interface it can achieve multiple inheritance .*

It is also used to *achieve loose coupling.*

Interfaces are used to implement abstraction.

So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

// A simple interface

interface Player

{

final int id = 10;

int move();

}

*// Java program to demonstrate working of interface.*

import java.io.\*;

// A simple interface

interface In1

{

// public, static and final

final int a = 10;

// public and abstract

void display();

}

// A class that implements the interface.

class TestClass implements In1

{

// Implementing the capabilities of

// interface.

public void display()

{

System.out.println("Geek");

}

// Driver Code

public static void main (String[] args)

{

TestClass t = new TestClass();

t.display();

System.out.println(a);

}

}

Output:

Geek

10

**A real-world example:**  
Let’s consider the example of vehicles like bicycle, car, bike………, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

import java.io.\*;

interface Vehicle {

// all are the abstract methods.

void changeGear(int a);

void speedUp(int a);

void applyBrakes(int a);

}

class Bicycle implements Vehicle{

int speed;

int gear;

// to change gear

@Override

public void changeGear(int newGear){

gear = newGear;

}

// to increase speed

@Override

public void speedUp(int increment){

speed = speed + increment;

}

// to decrease speed

@Override

public void applyBrakes(int decrement){

speed = speed - decrement;

}

public void printStates() {

System.out.println("speed: " + speed

+ " gear: " + gear);

}

}

class Bike implements Vehicle {

int speed;

int gear;

// to change gear

@Override

public void changeGear(int newGear){

gear = newGear;

}

// to increase speed

@Override

public void speedUp(int increment){

speed = speed + increment;

}

// to decrease speed

@Override

public void applyBrakes(int decrement){

speed = speed - decrement;

}

public void printStates() {

System.out.println("speed: " + speed

+ " gear: " + gear);

}

}

class GFG {

public static void main (String[] args) {

// creating an inatance of Bicycle

// doing some operations

Bicycle bicycle = new Bicycle();

bicycle.changeGear(2);

bicycle.speedUp(3);

bicycle.applyBrakes(1);

System.out.println("Bicycle present state :");

bicycle.printStates();

// creating instance of the bike.

Bike bike = new Bike();

bike.changeGear(1);

bike.speedUp(4);

bike.applyBrakes(3);

System.out.println("Bike present state :");

bike.printStates();

}

}

Output:

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

New features added in interfaces in JDK 8

Prior to JDK 8, interface could not define implementation.

We can now add *default implementation for interface methods.*

This default implementation has special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

// An example to show that interfaces can have methods from JDK 1.8 onwards.

interface In1

{

final int a = 10;

default void display()

{

System.out.println("hello");

}

}

// A class that implements the interface.

class TestClass implements In1

{

// Driver Code

public static void main (String[] args)

{

TestClass t = new TestClass();

t.display();

}

}

Output :

hello

Another feature that was added in JDK 8 is that we can now define *static methods in interfaces which can be called independently without an object.*

Note: these methods are not inherited.

// An example to show that interfaces can

// have methods from JDK 1.8 onwards

interface In1

{

final int a = 10;

static void display()

{

System.out.println("hello");

}

}

// A class that implements the interface.

class TestClass implements In1

{

// Driver Code

public static void main (String[] args)

{

In1.display();

}

}

Output :

hello

Important points about interface

*We can’t create instance(interface can’t be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.*

*A class can implement more than one interface.*

*An interface can extends another interface or interfaces (more than one interface) .*

*A class that implements interface must implements all the methods in interface.*

*All the methods are public and abstract.*

And *all the fields are public, static, and final.*

It is used to achieve multiple inheritance.

It is used to achieve loose coupling.

New features added in interfaces in JDK 9  
From Java 9 onwards, interfaces can contain following also

*Static methods*

*Private methods*

*Private Static methods.*

Note:

Short definition of loose coupling and tight coupling in java is that loose coupling means reducing dependencies of a class that use different class directly. Tight coupling means classes and objects are dependent on one another.

Data Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

*Encapsulation is defined as the wrapping up of data under a single unit.*

It is the mechanism that binds together code and the data it manipulates.

Other way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of class as private and the class is exposed to the end user or the world without providing any details behind implementation using the abstraction concept, so it is also known as **combination of data-hiding and abstraction.**

**Data Encapsulation = Data hiding + abstraction**

*Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.*

// Java program to demonstrate encapsulation

public class Encapsulate

{

// private variables declared these can only be accessed by

//public methods of class

private String geekName;

private int geekRoll;

private int geekAge;

// get method for age to access

// private variable geekAge

public int getAge()

{

return geekAge;

}

// get method for name to access

// private variable geekName

public String getName()

{

return geekName;

}

// get method for roll to access

// private variable geekRoll

public int getRoll()

{

return geekRoll;

}

// set method for age to access

// private variable geekage

public void setAge( int newAge)

{

geekAge = newAge;

}

// set method for name to access

// private variable geekName

public void setName(String newName)

{

geekName = newName;

}

// set method for roll to access

// private variable geekRoll

public void setRoll( int newRoll)

{

geekRoll = newRoll;

}

}

In the above program the class EncapsulateDemo is encapsulated as the variables are declared as private. The get methods like getAge() , getName() , getRoll() are set as public, these methods are used to access these variables. The setter methods like setName(), setAge(), setRoll() are also declared as public and are used to set the values of the variables.

The program to access variables of the class EncapsulateDemo is shown below:

public class TestEncapsulation

{

public static void main (String[] args)

{

Encapsulate obj = new Encapsulate();

// setting values of the variables

obj.setName("Harsh");

obj.setAge(19);

obj.setRoll(51);

// Displaying values of the variables

System.out.println("Geek's name: " + obj.getName());

System.out.println("Geek's age: " + obj.getAge());

System.out.println("Geek's roll: " + obj.getRoll());

// Direct access of geekRoll is not possible

// due to encapsulation

// System.out.println("Geek's roll: " + obj.geekName);

}

}

Output:

Geek's name: Harsh

Geek's age: 19

Geek's roll: 51

Advantages of Encapsulation:

Data Hiding: The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.

Increased Flexibility: We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program.

Reusability: Encapsulation also improves the re-usability and easy to change with new requirements.

Testing code is easy: Encapsulated code is easy to test for unit testing.

Constants:

A constant is a variable whose value cannot change once it has been assigned.

Java doesn't have built-in support for constants,

but the variable modifiers static and final can be used to effectively create one.

The static modifier causes the variable to be available without loading an instance of the class where it is defined.

The final modifier causes the variable to be unchangeable.

**Static Modifier:**

This allows a variable to be used without first creating an instance of the class; a static class member is associated with the class itself, rather than an object.

*All class instances share the same copy of the variable.*

*This means that another application or main() can easily use it.*

For example, class myClass contains a static variable days\_in\_week:

public class myClass {  
   static int days\_in\_week = 7;  
}

Because this variable is static, it can be used elsewhere without explicitly creating a myClass object:

public class myOtherClass {    
   static void main(String[] args) {  
       System.out.println(myClass.days\_in\_week);  
   }  
 }

**Final Modifier**

The final modifier means that the variable's value cannot change.

Once the value is assigned, it cannot be reassigned.

Primitive data types (i.e., int, short, long, byte, char, float, double, boolean) can be made immutable/unchangeable using the final modifier.

Together, these modifiers create a constant variable.

Constant = static +final

static final int DAYS\_IN\_WEEK = 7;

Note that we declared DAYS\_IN\_WEEK in all caps once we added the final modifier.

It's a *long-standing practice among Java programmers to define constant variables in all caps, as well as to separate words with underscores.*

Java doesn't require this formatting but it makes it easier for anyone reading the code to immediately identify a constant.

**Potential Problems With Constant Variables**

The way *the final keyword works in Java is that the variable's pointer to the value cannot change.*

*it's the pointer that cannot change the location to which it's pointing.*

There's no guarantee that the object being referenced will stay the same, only that the variable will always hold a reference to the same object.

If the referenced object is mutable (i.e. has fields that can be changed), then the constant variable may contain a value other than what was originally assigned.

ENUM:

Enumerations serve the purpose of representing a group of named constants in a programming language.

*Enums are used when we know all possible values at****compile time.***

It is *not necessary that the set of constants in an enum type stay****fixed****for all time.*

Declaration of enum in java :

*Enum declaration can be done outside a Class or inside a Class but not inside a Method.*

// A simple enum example where enum is declared

// outside any class (Note enum keyword instead of

// class keyword)

enum Color

{

RED, GREEN, BLUE;

}

public class Test

{

// Driver method

public static void main(String[] args)

{

Color c1 = Color.RED;

System.out.println(c1);

}

}

// enum declaration inside a class.

public class Test

{

enum Color

{

RED, GREEN, BLUE;

}

// Driver method

public static void main(String[] args)

{

Color c1 = Color.RED;

System.out.println(c1);

}

}

Note:

*First line inside enum should be list of constants* and then other things like methods, variables and constructor.

According to Java naming conventions, it is recommended that we name constant with all capital letters.

Keynote:

*Every enum internally implemented by using Class.*

/\* internally above enum Color is converted to

class Color

{

public static final Color RED = new Color();

public static final Color BLUE = new Color();

public static final Color GREEN = new Color();

}\*/

*Every enum constant represents an object of type enum.*

*enum type can be passed as an argument to switch statement.*

// A Java program to demonstrate working on enum in switch case (Filename Test. Java)

import java.util.Scanner;

// An Enum class

enum Day

{

SUNDAY, MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY;

}

// Driver class that contains an object of "day" and

// main().

public class Test

{

Day day;

// Constructor

public Test(Day day)

{

this.day = day;

}

// Prints a line about Day using switch

public void dayIsLike()

{

switch (day)

{

case MONDAY:

System.out.println("Mondays are bad.");

break;

case FRIDAY:

System.out.println("Fridays are better.");

break;

case SATURDAY:

case SUNDAY:

System.out.println("Weekends are best.");

break;

default:

System.out.println("Midweek days are so-so.");

break;

}

}

// Driver method

public static void main(String[] args)

{

String str = "MONDAY";

Test t1 = new Test(Day.valueOf(str));

t1.dayIsLike();

}

}

Output:

Mondays are bad.

Keynote 2:

Every enum constant is always implicitly *public static final.*

Since it is static, we can access it by using enum Name.

*Since it is final, we can’t create child enums.*

We *can declare main() method inside enum.* Hence we can invoke enum directly from the Command Prompt.

// A Java program to demonstrate that we can have main() inside enum class.

enum Color

{

RED, GREEN, BLUE;

// Driver method

public static void main(String[] args)

{

Color c1 = Color.RED;

System.out.println(c1);

}

}

Output :

RED

*Enum and Inheritance :*

All enums implicitly extend java.lang.Enum class.

As a class can only extend one parent in Java, so an enum cannot extend anything else.

*toString() method is overridden in java.lang.Enum class,which returns enum constant name.*

*enum can implement many interfaces.*

values(), ordinal() and valueOf() methods :

These methods are present inside java.lang.Enum.

*values()* method can be used to return all values present inside enum.

Order is important in enums.

By using *ordinal()* method, each enum constant index can be found, just like array index.

*valueOf()* method returns the enum constant of the specified string value, if exists.

// Java program to demonstrate working of values(), ordinal() and valueOf()

enum Color

{

RED, GREEN, BLUE;

}

public class Test

{

public static void main(String[] args)

{

// Calling values()

Color arr[] = Color.values();

// enum with loop

for (Color col : arr)

{

// Calling ordinal() to find index

// of color.

System.out.println(col + " at index "

+ col.ordinal());

}

// Using valueOf(). Returns an object of Color with given constant.

// Uncommenting second line causes exception

// IllegalArgumentException

System.out.println(Color.valueOf("RED"));

// System.out.println(Color.valueOf("WHITE"));

}

}

Output :

RED at index 0

GREEN at index 1

BLUE at index 2

RED

*enum and constructor :*

enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.

*We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.*

*enum and methods :*

*enum can contain concrete methods only i.e. no any abstract method.*

*// Java program to demonstrate that enums can have constructor*

*// and concrete methods.*

// An enum (Note enum keyword inplace of class keyword)

enum Color

{

RED, GREEN, BLUE;

// enum constructor called separately for each

// constant

private Color()

{

System.out.println("Constructor called for : " +

this.toString());

}

// Only concrete (not abstract) methods allowed

public void colorInfo()

{

System.out.println("Universal Color");

}

}

public class Test

{

// Driver method

public static void main(String[] args)

{

Color c1 = Color.RED;

System.out.println(c1);

c1.colorInfo();

}

}

Output:

Constructor called for : RED

Constructor called for : GREEN

Constructor called for : BLUE

RED

Universal Color

super (keyword):

The **super** keyword in Java is a reference variable which is used to refer *immediate parent class object.*

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.



super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

class Animal{

String color="white";

}

class Dog extends Animal{

String color="black";

void printColor(){

System.out.println(color);//prints color of Dog class

System.out.println(super.color);//prints color of Animal class

}

}

class TestSuper1{

public static void main(String args[]){

Dog d=new Dog();

d.printColor();

}}

Output:

black

white

super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method.

It should be used if subclass contains the same method as parent class. In other words, it is used if *method is overridden.*

**class** Animal{

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

}

**class** Dog **extends** Animal{

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

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

**void** work(){

**super**.eat();

bark();

}

}

**class** TestSuper2{

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

Dog d=**new** Dog();

d.work();

}}

Output:

eating...

barking...

super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

**class** Animal{

Animal(){System.out.println("animal is created");}

}

**class** Dog **extends** Animal{

Dog(){

**super**();

System.out.println("dog is created");

}

}

**class** TestSuper3{

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

Dog d=**new** Dog();

}}

Output:

animal is created

dog is created

Note:

*super() is added in each class constructor automatically by compiler if there is no super() or this().*

**example of super keyword where super() is provided by the compiler implicitly.**

**class** Animal{

Animal(){System.out.println("animal is created");}

}

**class** Dog **extends** Animal{

Dog(){

System.out.println("dog is created");

}

}

**class** TestSuper4{

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

Dog d=**new** Dog();

}}

Output:

animal is created

dog is created

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default.

*To initialize all the property, we are using parent class constructor from child class.* In such way, we are reusing the parent class constructor.

**class** Person{

**int** id;

String name;

Person(**int** id,String name){

**this**.id=id;

**this**.name=name;

}

}

**class** Emp **extends** Person{

**float** salary;

Emp(**int** id,String name,**float** salary){

**super**(id,name);//reusing parent constructor

**this**.salary=salary;

}

**void** display(){System.out.println(id+" "+name+" "+salary);}

}

**class** TestSuper5{

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

Emp e1=**new** Emp(1,"ankit",45000f);

e1.display();

}}

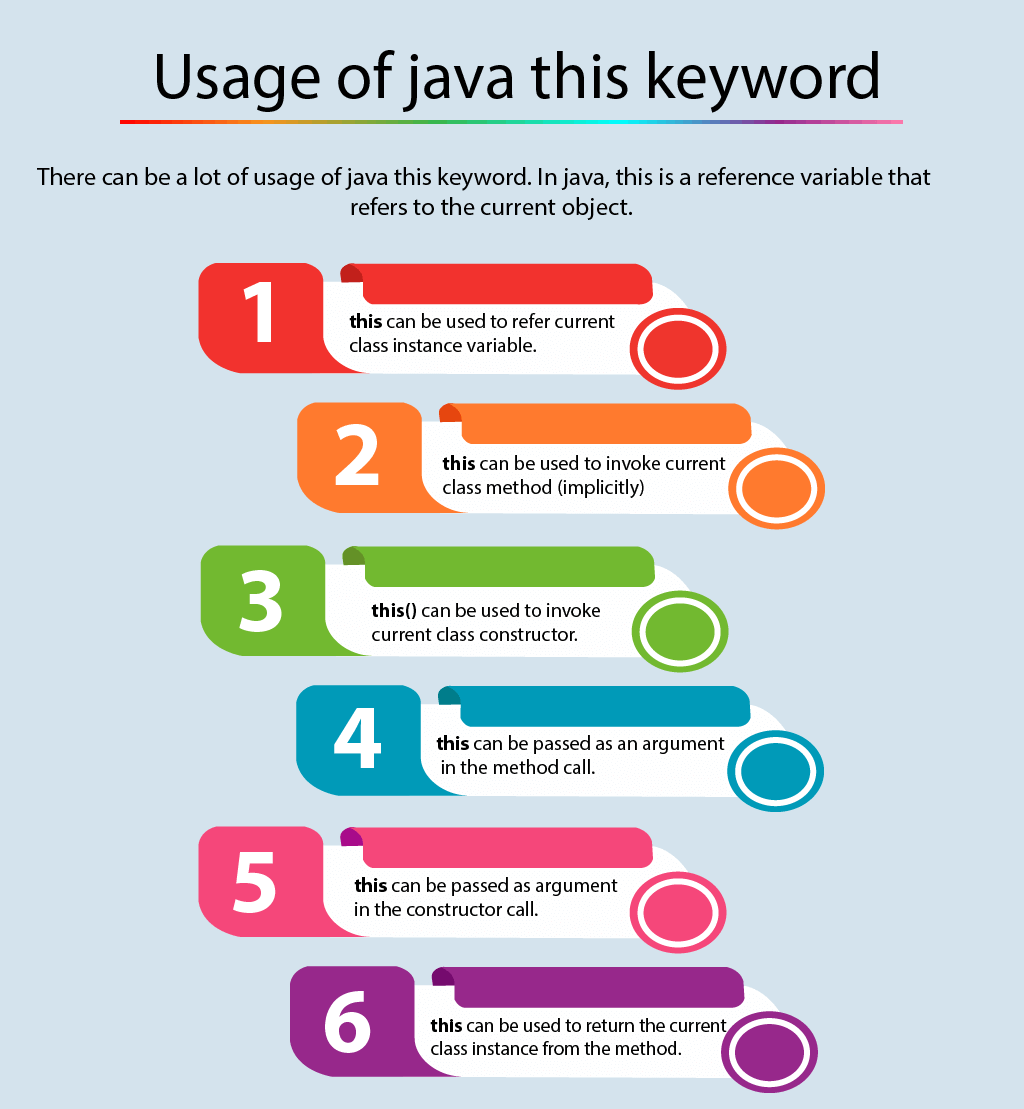
Output:

1 ankit 45000

Note:

super() is used to call Base class's constructor(i.e, Parent's class) while this() is used to call current class's constructor.

this :



this: to refer current class instance variable

The this keyword can be used to refer current class instance variable.

If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword.

this: to invoke current class method

If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

3) this() : to invoke current class constructor

It is used to reuse the constructor. In other words, it is used for constructor chaining.

Call to this() must be the first statement in constructor.

4) this: to pass as an argument in the method

It is mainly used in the event handling.

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

It is useful if we have to use one object in multiple classes.

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).

Syntax of this that can be returned as a statement

return\_type method\_name(){

**return** **this**;

}

ERRORS + EXCEPTIONS

Object class:

**Object** class is present in **java.lang** package.

The Object class is the parent class of all the classes in java by default.

(topmost class of java. )

Object class is beneficial to refer any object whose type is not known to user.

Every class in Java is directly or indirectly derived from the **Object** class.

If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived.

Therefore the Object class methods are available to all Java classes.

Object class acts as a root of inheritance hierarchy in any Java Program.

A screenshot of a cell phone

Description automatically generated

A picture containing screenshot

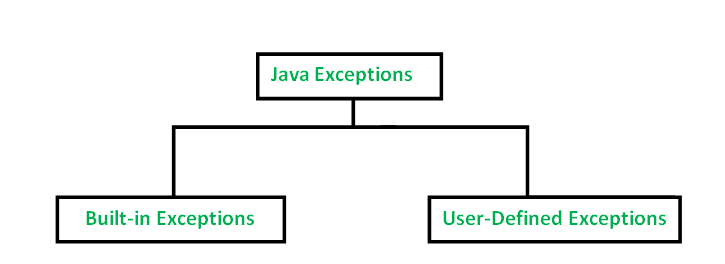
Description automatically generated

A screenshot of a cell phone

Description automatically generated

List of exceptions in java:

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations.

|  |  |
| --- | --- |
| *Java built in exceptions* | *description* |
| ArithmeticException | thrown when an exceptional condition has occurred in an arithmetic operation. |
| ArrayIndexOutOfBoundsException | to indicate that an array has been accessed with an illegal index.  The index is either negative or greater than or equal to the size of the array. |
| ClassNotFoundException | raised when we try to access a class whose definition is not found |
| FileNotFoundException | when a file is not accessible or does not open. |
| IOException | thrown when an input-output operation failed or interrupted |
| InterruptedException | It is thrown when a thread is waiting , sleeping ,or doing some processing , and it is interrupted. |
| NoSuchFieldException | thrown when a class does not contain the field (or variable) specified |
| NoSuchMethodException | It is thrown when accessing a method which is not found. |
| NullPointerException | raised when referring to the members of a null object. Null represents nothing |
| NumberFormatException | This exception is raised when a method could not convert a string into a numeric format. |
| RuntimeException | any exception which occurs during runtime. |
| StringIndexOutOfBounds  Exception | thrown by String class methods to indicate that an index is  either negative than the size of the string |

// Exception Handling and printing

*Exception is an error event that can happen during the execution of a program and disrupts its normal flow.*

Java provides a robust and object oriented way to handle exception scenarios, known as ***Java Exception Handling****.*

 Exceptions in java can arise from different kind of situations such as wrong data entered by user, hardware failure, network connection failure, Database server down etc.

*Java being an object oriented programming language, whenever an error occurs while executing a statement, creates an****exception object****and then the normal flow of the program halts and*[*JRE*](https://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm)*tries to find someone that can handle the raised exception.*

The exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, type of exception etc.

*When the exception occurs in a method, the process of creating the exception object and handing it over to runtime environment is called****“throwing the exception”****.*

Once runtime receives the exception object, it tries to find the handler for the exception.

*Exception Handler is the block of code that can process the exception object.*

The logic to find the exception handler is simple – starting the search in the method where error occurred, if no appropriate handler found, then move to the caller method and so on. So if methods call stack is A->B->C and exception is raised in method C, then the search for appropriate handler will move from C->B->A. If appropriate exception handler is found, exception object is passed to the handler to process it.

The handler is said to be **“catching the exception”**. If there are no appropriate exception handler found then program terminates printing information about the exception.

Note:

*Java Exception handling is a framework that is used to handle runtime errors only, compile time errors are not handled by exception handling in java.*

Java Exception Handling Keywords:

**throw** – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them.

Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null.

***throw****keyword is used to throw exception to the runtime to handle it.*

**throws** – When we are throwing any exception in a method and not handling it, then we need to use **throws** keyword in method signature (definition) to let caller program know the exceptions that might be thrown by the method.

The caller method might handle these exceptions or propagate it to its caller method using throws keyword.

We can provide multiple exceptions in the throws clause and it can be used with main() method also.

**try-catch** –

We use try-catch block for exception handling in our code.

try is the start of the block and catch is at the end of try block to handle the exceptions.

We can have multiple catch blocks with a try and try-catch block can be nested also.

catch block requires a parameter that should be of type Exception.

**finally** – finally block is optional and can be used only with try-catch block.

Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block.

*finally block gets executed always, whether exception occurred or not.*

Note:

*We can’t have catch or finally clause without a try statement.*

A try statement should have either catch block or finally block, it can have both blocks.

*We can’t write any code between try-catch-finally block.*

We can have multiple catch blocks with a single try statement.

try-catch blocks can be nested similar to if-else statements.

*We can have only one finally block with a try-catch statement.*

Java Exception Hierarchy

when any exception is raised an **exception object** is getting created.

Java Exceptions are hierarchical and inheritance is used to categorize different types of exceptions.

Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception.

Exceptions are further divided into checked exceptions and runtime exception.

**Errors**: Errors are exceptional scenarios that are out of scope of application and it’s not possible to anticipate and recover from them, for example hardware failure, JVM crash or out of memory error.

That’s why we have a separate hierarchy of errors and we should not try to handle these situations.

Some of the common Errors are OutOfMemoryError and StackOverflowError.

Note:

OutOfMemoryError:

occurs when java runtime is out of memory. In this case garbage collector is unable to free more space required by program and hence error is thrown.

There are two main reasons to get java.lang.OutOfMemoryError:

Poor programming – infinite loop, not clearing memory by closing resources etc.

Low Memory – java is running on less memory than required.

Java OutOfMemoryError – Poor Programming Example

package com.journaldev.exceptions;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

public class JavaOutOfMemoryErrorExample {

public static void main(String[] args) {

List<Integer> list = new ArrayList<>();

Random random = new Random();

while (true)

list.add(random.nextInt());

}

}

Java OutOfMemoryError – Low Memory Example

public class JavaOutOfMemoryErrorExample {

public static void main(String[] args) {

Integer[] array = new Integer[1000\*1000\*100];

System.out.println("Done");

}

}

StackOverflowError:

A StackOverflowError is a runtime error in java. It is thrown when the *amount of call stack memory allocated by JVM is exceeded.* A common case of a StackOverflowError being thrown, is when call stack exceeds due to excessive deep or infinite recursion.

The StackOverflowError extends the VirtualMachineError class, which indicates that the JVM is broken, or it has run out of resources and cannot operate. Furthermore, the the VirtualMachineError extends the Error class, which is used to indicate those serious problems that an application should not catch. A method may not declare such errors in its throw clause, because these errors are abnormal conditions that shall never occur.

Finally, the StackOverflowError exists since the 1.0 version of Java.

When a function call is invoked by a Java application, a **stack frame** is allocated on the call stack. The stack frame contains the parameters of the invoked method, its local parameters, and the return address of the method. The return address denotes the execution point from which, the program execution shall continue after the invoked method returns. If there is no space for a new stack frame then, the StackOverflowError is thrown by the Java Virtual Machine (JVM).

The most common case that can possibly exhaust a Java application’s stack is *recursion*. In recursion, a method invokes itself during its execution.

**Checked Exceptions**:

Checked Exceptions are exceptional scenarios that we can anticipate in a program and try to recover from it, for example FileNotFoundException.

We should catch this exception and provide useful message to user and log it properly for debugging purpose.

*Exception is the parent class of all Checked Exceptions* and if we are throwing a checked exception, we must catch it in the same method or we have to propagate it to the caller using throws keyword.

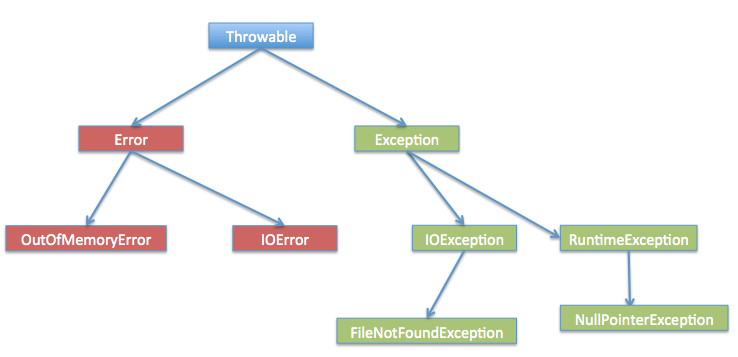
**Runtime Exception**:

Runtime Exceptions are cause by bad programming, for example trying to retrieve an element from the Array. We should check the length of array first before trying to retrieve the element otherwise it might throw ArrayIndexOutOfBoundException at runtime.

*RuntimeException is the parent class of all runtime exceptions.*

If we are throwing any runtime exception in a method, it’s not required to specify them in the method signature throws clause.

Runtime exceptions can be avoided with better programming.



Note:

NullPointerException is a RuntimeException.

In Java, a special null value can be assigned to an object reference. NullPointerException is thrown when program attempts to use an object reference that has the null value.

Invoking a method from a null object.

Accessing or modifying a null object’s field.

Taking the length of null, as if it were an array.

Accessing or modifying the slots of null object, as if it were an array.

Throwing null, as if it were a [Throwable](https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html) value.

try to synchronize over a null object.

In order to avoid the [NullPointerException](https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html), ensure that all your objects are initialized properly, before you use them. Notice that, when you declare a reference variable, you are really creating a pointer to an object. You must verify that the pointer is not null, before you request the method or a field from the object.

Also, if the exception is thrown, use the information residing in the exception’s stack trace. The execution’s stack trace is provided by the JVM, in order to enable the debugging of the application. Locate the method and the line where the exception was caught and then, figure out which reference equals to null in that specific line.

*Exception Handling in Java – Useful Methods*

*Java Exception and all of it’s subclasses doesn’t provide any specific methods, and all of the methods are defined in the base class Throwable.*

The exception classes are created to specify different kind of exception scenarios so that we can easily identify the root cause and handle the exception according to it’s type.

*Throwable class implements Serializable interface for interoperability.*

Some of the useful methods of Throwable class are;

**public String getMessage()** –

This method returns the message String of Throwable and the message can be provided while creating the exception through its constructor.

**Public String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program.

Throwable class implementation of this method simply use getMessage() method to return the exception message.

**Public synchronized Throwable getCause()** –

This method returns the cause of the exception or null id if the cause is unknown.

**Public String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.

**Public void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

*Java 7 Automatic Resource Management and Catch block improvements:*

If you are catching a lot of exceptions in a single try block, you will notice that catch block code looks very ugly and mostly consists of redundant code to log the error, keeping this in mind Java 7 one of the feature was improved catch block where we can catch multiple exceptions in a single catch block. The catch block with this feature looks like below:

catch(IOException | SQLException ex){

logger.error(ex);

throw new MyException(ex.getMessage());

}

There are some constraints such as the exception object is final and we can’t modify it inside the catch block.

Most of the time, we use finally block just to close the resources and sometimes we forget to close them and get runtime exceptions when the resources are exhausted. These exceptions are hard to debug and we might need to look into each place where we are using that type of resource to make sure we are closing it.

So java 7 one of the improvement was *try-with-resources where we can create a resource in the try statement itself and use it inside the try-catch block. When the execution comes out of try-catch block, runtime environment automatically close these resources.*

Sample of try-catch block with this improvement is:

try (MyResource mr = new MyResource()) {

System.out.println(“MyResource created in try-with-resources”);

} catch (Exception e) {

e.printStackTrace();

}

Exception Handling in Java – Creating Custom Exception Classes

Java provides a lot of exception classes for us to use but sometimes we may need to create our own custom exception classes to notify the caller about specific type of exception with appropriate message and any custom fields we want to introduce for tracking, such as error codes. For example, let’s say we write a method to process only text files, so we can provide caller with appropriate error code when some other type of file is sent as input.

Exception Handling in Java – Best Practices

**Use Specific Exceptions** – Base classes of Exception hierarchy doesn’t provide any useful information, That is why Java has so many exception classes, such as IOException with further sub-classes as FileNotFoundException, EOFException etc. We should always throw and catch specific exception classes so that caller will know the root cause of exception easily and process them. This makes debugging easy and helps client application to handle exceptions appropriately.

**Throw Early or Fail-Fast** – We should try to throw exceptions as early as possible.

While debugging we will have to look out at the stack trace carefully to identify the actual location of exception. If we change our implementation logic to check for these exceptions early as below;

**Catch Late** – Since java enforces to either handle the checked exception or to declare it in method signature, sometimes developers tend to catch the exception and log the error. But this practice is harmful because the caller program doesn’t get any notification for the exception. We should catch exception only when we can handle it appropriately.

While implementing any feature, we should always throw exceptions back to the caller and let them decide how to handle it.

**Closing Resources** – Since exceptions halt the processing of program, we should close all the resources in finally block or use Java 7 try-with-resources enhancement to let java runtime close it for you.

**Logging Exceptions** – We should always log exception messages and while throwing exception provide clear message so that caller will know easily why the exception occurred. We should always avoid empty catch block that just consumes the exception and doesn’t provide any meaningful details of exception for debugging.

**Single catch block for multiple exceptions** – Most of the times we log exception details and provide message to the user, in this case we should use java 7 feature for handling multiple exceptions in a single catch block. This approach will reduce our code size and it will look cleaner too.

**Using Custom Exceptions** – It’s always better to define exception handling strategy at the design time and rather than throwing and catching multiple exceptions, we can create a custom exception with error code and caller program can handle these error codes. Its also a good idea to create a utility method to process different error codes and use it.

**Naming Conventions and Packaging** – When you create your custom exception, make sure it ends with Exception so that it will be clear from name itself that it’s an exception. Also make sure to package them like it’s done in [JDK](https://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm), for example IOException is the base exception for all IO operations.

**Use Exceptions Judiciously** – Exceptions are costly and sometimes it’s not required to throw exception at all and we can return a avadoc variable to the caller program to indicate whether an operation was successful or not. This is helpful where the operation is optional and you don’t want your program to get stuck because it fails. For example, while updating the stock quotes in database from a third party webservice, we may want to avoid throwing exception if the connection fails.

**Document the Exceptions Thrown** – Use javadoc @throws to clearly specify the exceptions thrown by the method, it’s very helpful when you are providing an interface to other applications to use.

Note:

printStackTrace is a method of the Throwable class. This method displays error message in the console; where we are getting the exception in the source code. These methods can be used with catch block and they describe:

Name of the exception.

Description of the exception.

Location of the exception in the source code.

The three methods which describe the exception on the console (in which printStackTrace is one of them) are:

printStackTrace()

toString()

getMessage()

Example:

public class BabluGope {

public static void main(String[] args) {

try {

System.out.println(10/0);

} catch (ArithmeticException e) {

e.printStackTrace();

// System.err.println(e.toString());

//System.err.println(e.getMessage());

}

}

}

FILE OPERATIONS :

**What is File Handling in Java?**

File handling in Java implies reading from and writing data to a file.

The File class from the **java.io package**, allows us to work with different formats of files. In order to use the File class, you need to create an object of the class and specify the filename or directory name.

For example:

// Import the File class

Import java.io.File

// Specify the filename

File obj = new File("filename.txt");

Java uses the concept of a stream to make I/O operations on a file.

**What is a Stream?**

In Java, Stream is a sequence of data which can be of two types.

**1. Byte Stream**

This mainly incorporates with byte data. When an input is provided and executed with byte data, then it is called the file handling process with a byte stream.

**2. Character Stream**

Character Stream is a stream which incorporates with characters. Processing of input data with character is called the file handling process with a character stream.

**Java File Methods**

|  |  |  |
| --- | --- | --- |
| Method | Type | Description |
| canRead() | Boolean | It tests whether the file is readable or not |
| canWrite() | Boolean | It tests whether the file is writable or not |
| createNewFile() | Boolean | This method creates an empty file |
| delete() | Boolean | Deletes a file |
| exists() | Boolean | It tests whether the file exists |
| getName() | String | Returns the name of the file |
| getAbsolutePath() | String | Returns the absolute pathname of the file |
| length() | Long | Returns the size of the file in bytes |
| list() | String[] | Returns an array of the files in the directory |
| mkdir() | Boolean | Creates a directory |

**File Operations in Java**

can perform four operations on a file. They are as follows:

Create a File

Get File Information

Write To a File

Read from a File

**Create a File**

In this case, to create a file you can use the **createNewFile()** method.

This method returns **true** if the file was successfully created, and **false** if the file already exists.

**Get File information**

**Write to a File**

**FileWriter** class together with its **write()** method to write some text into the file.

**Read from a File**

Scanner class to read the contents of the text file.

PROPERTIES file

Properties class in Java

The properties object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key.

The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

**Recompilation is not required if the information is changed from a properties file:** If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

Read data from properties file:

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class. It is character-oriented class which is used for file handling in java.

load() method from Properties class.

get(Object Key) from Properties class.

Create and write data to a properties file:

setProperty("Key","Value");

store(**new** FileWriter("fileName.properties"),"commented title of the file");

EXCEL workbook with Java:

Note:

|  |  |  |  |
| --- | --- | --- | --- |
| No. | final | finally | finalize |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

Java final example

**class** FinalExample{

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

**final** **int** x=100;

x=200;//Compile Time Error

}}

Java finally example

**class** FinallyExample{

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

**try**{

**int** x=300;

}**catch**(Exception e){System.out.println(e);}

**finally**{System.out.println("finally block is executed");}

}}

Java finalize example

**class** FinalizeExample{

**public** **void** finalize(){System.out.println("finalize called");}

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

FinalizeExample f1=**new** FinalizeExample();

FinalizeExample f2=**new** FinalizeExample();

f1=**null**;

f2=**null**;

System.gc();

}}

Different ways to create objects in Java:

1) Using new Keyword :

public class UsingNewKeyword {

String name = "Object creation";

public static void main(String[] args) {

UsingNewKeyword obj = new UsingNewKeyword();

System.out.println(obj.name);

}

}

**2) Using New Instance :**

If we know the name of the class & if it has a public default constructor we can create an object –Class.forName.

Class.forName actually loads the Class in Java but doesn’t create any Object.

To Create an Object of the Class you have to use the new Instance Method of the Class.

public class NewInstanceExample

{

    String name = "GeeksForGeeks";

    public static void main(String[] args)

    {

        try

        {

            Class cls = Class.forName("NewInstanceExample");

            NewInstanceExample obj =

                    (NewInstanceExample) cls.newInstance();

            System.out.println(obj.name);

        }

        catch (ClassNotFoundException e)

        {

            e.printStackTrace();

        }

        catch (InstantiationException e)

        {

            e.printStackTrace();

        }

        catch (IllegalAccessException e)

        {

            e.printStackTrace();

        }

    }

}

**3) Using clone() method:** Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.  
To use clone() method on an object we need to implement **Cloneable** and define the clone() method in it.

// Java program to illustrate creation of Object

// using clone() method

public class CloneExample implements Cloneable

{

@Override

protected Object clone() throws CloneNotSupportedException

{

return super.clone();

}

String name = "GeeksForGeeks";

public static void main(String[] args)

{

CloneExample obj1 = new CloneExample();

try

{

CloneExample obj2 = (CloneExample) obj1.clone();

System.out.println(obj2.name);

}

catch (CloneNotSupportedException e)

{

e.printStackTrace();

}

}

}

**Note :**

Here we are creating the clone of an existing Object and not any new Object.

Class need to implement Cloneable Interface otherwise it will throw **CloneNotSupportedException**.

**4) Using deserialization :** Whenever we serialize and then deserialize an object, JVM creates a separate object. In **deserialization**, JVM doesn’t use any constructor to create the object.  
To deserialize an object we need to implement the Serializable interface in the class.

// Java program to illustrate Serializing

// an Object.

import java.io.\*;

class DeserializationExample implements Serializable

{

private String name;

DeserializationExample(String name)

{

this.name = name;

}

public static void main(String[] args)

{

try

{

DeserializationExample d =

new DeserializationExample("GeeksForGeeks");

FileOutputStream f = new FileOutputStream("file.txt");

ObjectOutputStream oos = new ObjectOutputStream(f);

oos.writeObject(d);

oos.close();

f.close();

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

Object of DeserializationExample class is serialized using writeObject() method and written to file.txt file.

// Java program to illustrate creation of Object

// using Deserialization.

import java.io.\*;

public class DeserializationExample

{

public static void main(String[] args)

{

try

{

DeserializationExample d;

FileInputStream f = new FileInputStream("file.txt");

ObjectInputStream oos = new ObjectInputStream(f);

d = (DeserializationExample)oos.readObject();

}

catch (Exception e)

{

e.printStackTrace();

}

System.out.println(d.name);

}

}

5) Using newInstance() method of Constructor class : This is similar to the newInstance() method of a class. There is one newInstance() method in the java.lang.reflect.Constructor class which we can use to create objects. It can also call parameterized constructor, and private constructor by using this newInstance() method.

Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor class.

// Java program to illustrate creation of Object

// using newInstance() method of Constructor class

import java.lang.reflect.\*;

public class ReflectionExample

{

private String name;

ReflectionExample()

{

}

public void setName(String name)

{

this.name = name;

}

public static void main(String[] args)

{

try

{

Constructor<ReflectionExample> constructor

= ReflectionExample.class.getDeclaredConstructor();

ReflectionExample r = constructor.newInstance();

r.setName("GeeksForGeeks");

System.out.println(r.name);

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

STRINGS

*In*[***Java***](https://www.javatpoint.com/java-tutorial)*, string is basically an object that represents sequence of char values.*

*An array of characters works same as Java string.*

For example:

**char**[] ch= {'j','a','v','a','t','p','o','i','n','t'};

String s=**new** String(ch);

is same as:

String s=**"javatpoint"**;

*Note* :

*“Strings are immutable in Java”*

*Immutable simply means unmodifiable or unchangeable.*

*Once string object is created its data or state can't be changed but a new string object is created.*

*Immutable class means that once an object is created, we cannot change its content.*

*In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable.*

*String is immutable means that we cannot change the object itself, but we can change the reference to the object.*

*Strings in Java are Objects that are backed internally by a char array.*

Since arrays are immutable(cannot grow), Strings are immutable as well.

Whenever a change to a String is made, an entirely new String is created.

*String objects are cached in String pool.*

*String pool*:

*a pool of Strings stored in Java Heap Memory.*

From Java 7 onwards, the Java String Pool is stored in the *Heap space, which is garbage collected by the JVM.*

Since *Strings are very popular as HashMap key*, it's important for them to be immutable so that they can retrieve the value object which was stored in HashMap.

*Caching Hashcode*

The **hashcode** of a **string** is frequently used in Java. For **example**, in a **HashMap** or **HashSet**.

Being **immutable** guarantees that hashcode will always be the same so that it can be cashed without worrying about the changes.

That means, there is no need to calculate hashcode every time it is used.

**Note:**

hashCode() is used for *bucketing* in Hash implementations like HashMap, HashTable, HashSet, etc.

The value received from hashCode() is used as the *bucket number* for storing elements of the set/map.

This bucket number is the *address* of the element inside the set/map.

When you do contains() it will take the hash code of the element, then look for the bucket where hash code points to. If more than 1 element is found in the same bucket (*multiple objects can have the same hash code)*, then it uses the equals() method to evaluate if the objects are equal, and then decide if contains() is true or false, or decide if element could be added in the set or not.

*Why string objects are immutable in java?*

Because java uses the concept of *string literal.*

*Suppose there are 5 reference variables,all refers to one object "sachin".*

*If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.*

The java.lang.String class implements **Serializable**, **Comparable** and **CharSequence** interfaces.

A close up of a logo

Description automatically generated

**Serializable:**

**Serialization in Java** *is a mechanism of writing the state of an object into a byte-stream.*

It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The *reverse operation* of **serialization** is called *deserialization* where *byte-stream is converted into an object.*

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object.

*A Java object is serializable if its class or any of its superclasses implements either the java. io. Serializable interface or its subinterface.*

Serializable is a marker interface (has no data member and method).

It is used to "mark" Java classes so that the objects of these classes may get a certain capability.

It must be implemented by the class whose object you want to persist.

**Note:** marker interface –

*Marker Interface in java is an interface with no fields or methods within it.*

It is used to convey to the JVM that the class implementing an interface of this category will have some special behavior.

Hence, *an empty interface in java is called a marker interface.*

Examples of marker interface are Serializable, Cloneable and Remote interface.

A class that implements the Cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class.

For serializing the object, we call the **writeObject()** method of **ObjectOutputStream**, and for deserialization we call the **readObject()** method of **ObjectInputStream** class.

*Advantages of Java Serialization :*

used to *transport object's state on the network* (which is known as marshaling).

The String class and all the wrapper classes implement the java.io.Serializable interface by default.

**Ex:**

import java.io.Serializable;

public class Student implements Serializable{

 int id;

 String name;

 public Student(int id, String name) {

  this.id = id;

  this.name = name;

 }

}

**Comparable** :

*Java Comparable interface is used to order the objects of the user-defined class.*

This interface is found in java. lang package and contains *only one method* named *compareTo(Object).*

It provides a *single sorting sequence only, i.e., can sort the elements on the basis of single data member only.*

**Java Comparable** interface used to *sort an array or list of objects* by their natural order.

*Natural ordering of elements is imposed by compareTo() method.*

*CharSequence Interface:*

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it.

It means, we can create strings in java by using these three classes.

A close up of a logo

Description automatically generated

*Note:*

*Generally, String is a sequence of characters.*

*But in Java, string is an object that represents a sequence of characters.*

*The java.lang.String class is used to create a string object.*

*There are two ways to create String object:*

*By string literal*

*By new keyword*

*By string literal :*

Java String literal is created by using double quotes.

*String s=****"welcome"****;*

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance

A close up of a logo

Description automatically generated

*Note:*

*String objects are stored in a special memory area known as the*

*"string constant pool".*

*Note:*

*Why Java uses the concept of String literal?*

*To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).*

By new keyword:

String s=new String("Welcome");

//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

**public** **class** StringExample{

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

String s1="java";//creating string by java string literal

**char** ch[]={'s','t','r','i','n','g','s'};

String s2=**new** String(ch);//converting char array to string

String s3=**new** String("example");//creating java string by new keyword

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

}}

*Immutable String in Java*

In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

**class** Testimmutablestring{

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

   String s="Sachin";

   s.concat(" Tendulkar");//concat() method appends the string at the end

   System.out.println(s);//will print Sachin because strings are immutable objects

 }

}

Output:Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

**class** Testimmutablestring1{

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

   String s="Sachin";

   s=s.concat(" Tendulkar");

   System.out.println(s);

 }

}

Output:Sachin Tendulkar

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

*Java String compare* on the basis of *content* and *reference*.

It is used in authentication (by equals() method), sorting (by compareTo() method), reference matching (by == operator) etc.

There are *three ways to compare string in java*:

*By equals() method*

*By = = operator*

*By compareTo() method*

String compare by equals() method

compares the original content of the string.

It compares values of string for equality.

String class provides two methods:

public boolean equals(Object another) compares this string to the specified object.

public boolean equalsIgnoreCase(String another) compares this String to another string, ignoring case.

**class** Teststringcomparison1{

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

   String s1="Sachin";

   String s2="Sachin";

   String s3=**new** String("Sachin");

   String s4="Saurav";

   System.out.println(s1.equals(s2));//true

   System.out.println(s1.equals(s3));//true

   System.out.println(s1.equals(s4));//false

 }

}

**class** Teststringcomparison2{

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

   String s1="Sachin";

   String s2="SACHIN";

   System.out.println(s1.equals(s2));//false

   System.out.println(s1.equalsIgnoreCase(s2));//true

 }

}

The java string equals() method compares the two given strings based on the content of the string. If any character is not matched, it returns false.

If all characters are matched, it returns true.

*The String equals() method overrides the equals() method of Object class.*

String compare by == operator

*The = = operator compares references not values.*

**class** Teststringcomparison3{

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

   String s1="Sachin";

   String s2="Sachin";

   String s3=**new** String("Sachin");

   System.out.println(s1==s2);//true (because both refer to same instance)

   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)

 }

}

String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

s1 == s2 :0

s1 > s2   :positive value

s1 < s2   :negative value

**class** Teststringcomparison4{

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

   String s1="Sachin";

   String s2="Sachin";

   String s3="Ratan";

   System.out.println(s1.compareTo(s2));//0

   System.out.println(s1.compareTo(s3));//1(because s1>s3)

   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )

 }

}

*The java string compareTo() method compares the given string with current string lexicographically.* I

t returns positive number, negative number or 0.

*It compares strings on the basis of Unicode value of each character in the strings.*

If first string is lexicographically greater than second string, it returns positive number (difference of character value). If first string is less than second string lexicographically, it returns negative number and if first string is lexicographically equal to second string, it returns 0.

**public** **class** CompareToExample{

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

String s1="hello";

String s2="hello";

String s3="meklo";

String s4="hemlo";

String s5="flag";

System.out.println(s1.compareTo(s2));//0 because both are equal

System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"

System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"

System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"

}}

*Java String compareTo(): empty string*

If you compare string with blank or empty string, it returns length of the string. If second string is empty, result would be positive. If first string is empty, result would be negative.

**public** **class** CompareToExample2{

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

String s1="hello";

String s2="";

String s3="me";

System.out.println(s1.compareTo(s2));

System.out.println(s2.compareTo(s3));

}}

Output:

5

-2

Substring in Java:

A part of string is called substring.

In other words, *substring is a subset of another string.*

In case of substring *startIndex is inclusive and endIndex is exclusive.*

*Note: Index starts from 0.*

You can get substring from the given string object by one of the two methods:

public String substring(int startIndex): This method returns new String object containing the substring of the given string from specified startIndex (inclusive).

public String substring(int startIndex, int endIndex): This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

String s="hello";

System.out.println(s.substring(0,2));//he

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

Example of java substring

**public** **class** TestSubstring{

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

   String s="SachinTendulkar";

   System.out.println(s.substring(6));//Tendulkar

   System.out.println(s.substring(0,6));//Sachin

 }

}

Java String class methods

The *java.lang.String* class provides a lot of methods to work on string.

1. charAt()

The java string charAt() method returns *a char value at the given index number*.

The index number starts *from 0 and goes to n-1,* where n is length of the string.

It returns StringIndexOutOfBoundsException if given index number is greater than or equal to this string length or a negative number.

**public** **class** CharAtExample{

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

String name="javatpoint";

**char** ch=name.charAt(4);//returns the char value at the 4th index

System.out.println(ch);

}}

Output:

t

StringIndexOutOfBoundsException with charAt()

**public** **class** CharAtExample{

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

String name="javatpoint";

**char** ch=name.charAt(10);//returns the char value at the 10th index

System.out.println(ch);

}}

Output:

Exception in thread "main" java.lang.StringIndexOutOfBoundsException:

String index out of range: 10

at java.lang.String.charAt(String.java:658)

at CharAtExample.main(CharAtExample.java:4)

public class CharAtExample3 {

    public static void main(String[] args) {

    String str = "Welcome to Javatpoint portal";

    int strLength = str.length();

    // Fetching first character

    System.out.println("Character at 0 index is: "+ str.charAt(0));

    // The last Character is present at the string length-1 index

    System.out.println("Character at last index is: "+ str.charAt(strLength-1));

    }

}

Output:

Character at 0 index is: W

Character at last index is: l

an example where we are accessing all the elements present at odd index.

**public** **class** CharAtExample4 {

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

        String str = "Welcome to Javatpoint portal";

**for** (**int** i=0; i<=str.length()-1; i++) {

**if**(i%2!=0) {

                System.out.println("Char at "+i+" place "+str.charAt(i));

            }

        }

    }

}

an example where we are counting frequency of a character in the string.

**public** **class** CharAtExample5 {

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

        String str = "Welcome to Javatpoint portal";

**int** count = 0;

**for** (**int** i=0; i<=str.length()-1; i++) {

**if**(str.charAt(i) == 't') {

                count++;

            }

        }

        System.out.println("Frequency of t is: "+count);

    }

}

2.concat

The java string concat() method **combines** specified string at the end of this string.

It returns combined string.

It is like appending another string.

Signature

public String concat(String anotherString)

Java String concat() method example

**public** **class** ConcatExample{

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

String s1="java string";

s1.concat("is immutable");

System.out.println(s1);

s1=s1.concat(" is immutable so assign it explicitly");

System.out.println(s1);

}}

Op:

java string

java string is immutable so assign it explicitly

**public** **class** ConcatExample2 {

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

        String str1 = "Hello";

        String str2 = "Javatpoint";

        String str3 = "Reader";

        // Concatenating one string

        String str4 = str1.concat(str2);

        System.out.println(str4);

        // Concatenating multiple strings

        String str5 = str1.concat(str2).concat(str3);

        System.out.println(str5);

    }

}

HelloJavatpoint

HelloJavatpointReader

concatenating spaces and special chars to the string object.

**public** **class** ConcatExample3 {

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

        String str1 = "Hello";

        String str2 = "Javatpoint";

        String str3 = "Reader";

        // Concatenating Space among strings

        String str4 = str1.concat(" ").concat(str2).concat(" ").concat(str3);

        System.out.println(str4);

        // Concatenating Special Chars

        String str5 = str1.concat("!!!");

        System.out.println(str5);

        String str6 = str1.concat("@").concat(str2);

        System.out.println(str6);

    }

}

Output:

Hello Javatpoint Reader

Hello!!!

Hello@Javatpoint

contains()

The java string contains() method searches the sequence of characters in this string.

It returns *true* if sequence of char values are found in this string otherwise returns *false*.

Signature

public boolean contains(CharSequence sequence)

Returns

true if sequence of char value exists, otherwise false.

Throws

NullPointerException : if sequence is null.

**class** ContainsExample{

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

String name="what do you know about me";

System.out.println(name.contains("do you know"));

System.out.println(name.contains("about"));

System.out.println(name.contains("hello"));

}}

true

true

false

The contains() method searches case sensitive char sequence.

**public** **class** ContainsExample2 {

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

        String str = "Hello Javatpoint readers";

**boolean** isContains = str.contains("Javatpoint");

        System.out.println(isContains);

        // Case Sensitive

        System.out.println(str.contains("javatpoint")); // false

    }

}

The contains() method is helpful to find a char-sequence in the string.

We can use it in control structure to produce search based result.

an example below.

public class ContainsExample3 {

    public static void main(String[] args) {

        String str = "To learn Java visit Javatpoint.com";

        if(str.contains("Javatpoint.com")) {

            System.out.println("This string contains javatpoint.com");

        }else

            System.out.println("Result not found");

    }

}

Output:

This string contains javatpoint.com

endsWith()

The java string endsWith() method checks if this string ends with given suffix. It returns true if this string ends with given suffix else returns false.

**public** **class** EndsWithExample{

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

String s1="java by javatpoint";

System.out.println(s1.endsWith("t"));

System.out.println(s1.endsWith("point"));

}}

Output:

true

true

**public** **class** EndsWithExample2 {

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

        String str = "Welcome to Javatpoint.com";

        System.out.println(str.endsWith("point"));

**if**(str.endsWith(".com")) {

            System.out.println("String ends with .com");

        }**else** System.out.println("It does not end with .com");

    }

}

Output:

false

String ends with .com

5. equalsIgnoreCase()

The **String equalsIgnoreCase()** method *compares the two given strings on the basis of content of the string irrespective of case of the string.*

It is like equals() method but doesn't check case.

If any character is not matched, it returns false otherwise it returns true.

**public** **class** EqualsIgnoreCaseExample{

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

String s1="javatpoint";

String s2="javatpoint";

String s3="JAVATPOINT";

String s4="python";

System.out.println(s1.equalsIgnoreCase(s2));//true because content and case both are same

System.out.println(s1.equalsIgnoreCase(s3));//true because case is ignored

System.out.println(s1.equalsIgnoreCase(s4));//false because content is not same

}}

true

true

false

an example where we are testing string equality among the strings.

**import** java.util.ArrayList;

**public** **class** EqualsIgnoreCaseExample2 {

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

        String str1 = "Mukesh Kumar";

        ArrayList<String> list = **new** ArrayList<>();

        list.add("Mohan");

        list.add("Mukesh");

        list.add("RAVI");

        list.add("MuKesH kuMar");

        list.add("Suresh");

**for** (String str : list) {

**if** (str.equalsIgnoreCase(str1)) {

                System.out.println("Mukesh kumar is present");

            }

        }

    }

}

Output:

Mukesh kumar is present

6. format()

The java string format() method returns the formatted string by given locale, format and arguments.

If you don't specify the locale in String.format() method, it uses default locale by calling *Locale.getDefault()* method.

The format() method of java language is like *sprintf()* function in c language and *printf()* method of java language.

Parameters

locale : specifies the locale to be applied on the format() method.

format : format of the string.

args : arguments for the format string. It may be zero or more.

Returns

formatted string

Throws

NullPointerException : if format is null.

IllegalFormatException : if format is illegal or incompatible.

**public** **class** FormatExample{

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

String name="sonoo";

String sf1=String.format("name is %s",name);

String sf2=String.format("value is %f",32.33434);

String sf3=String.format("value is %32.12f",32.33434);//returns 12 char fractional part filling with 0

System.out.println(sf1);

System.out.println(sf2);

System.out.println(sf3);

}}

name is sonoo

value is 32.334340

value is 32.334340000000

Java String Format Specifiers

|  |  |  |
| --- | --- | --- |
| Format Specifier | Data Type | Output |
| %a | floating point (except *BigDecimal*) | Returns Hex output of floating point number. |
| %b | Any type | "true" if non-null, "false" if null |
| %c | character | Unicode character |
| %d | integer (incl. byte, short, int, long, bigint) | Decimal Integer |
| %e | floating point | decimal number in scientific notation |
| %f | floating point | decimal number |
| %g | floating point | decimal number, possibly in scientific notation depending on the precision and value. |
| %h | any type | Hex String of value from hashCode() method. |
| %n | none | Platform-specific line separator. |
| %o | integer (incl. byte, short, int, long, bigint) | Octal number |
| %s | any type | String value |
| %t | Date/Time (incl. long, Calendar, Date and TemporalAccessor) | %t is the prefix for Date/Time conversions. More formatting flags are needed after this. See Date/Time conversion below. |
| %x | integer (incl. byte, short, int, long, bigint) | Hex string. |

This method supports various data types and formats them into a string type. Let us see an example.

**public** **class** FormatExample2 {

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

        String str1 = String.format("%d", 101);          // Integer value

        String str2 = String.format("%s", "Amar Singh"); // String value

        String str3 = String.format("%f", 101.00);       // Float value

        String str4 = String.format("%x", 101);          // Hexadecimal value

        String str5 = String.format("%c", 'c');          // Char value

        System.out.println(str1);

        System.out.println(str2);

        System.out.println(str3);

        System.out.println(str4);

        System.out.println(str5);

    }

}

101

Amar Singh

101.000000

65

c

Apart from formatting, we can set width, padding etc. of any value. Let us see an example where we are setting width and padding for an integer value.

**public** **class** FormatExample3 {

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

        String str1 = String.format("%d", 101);

        String str2 = String.format("|%10d|", 101);  // Specifying length of integer

        String str3 = String.format("|%-10d|", 101); // Left-justifying within the specified width

        String str4 = String.format("|% d|", 101);

        String str5 = String.format("|%010d|", 101); // Filling with zeroes

        System.out.println(str1);

        System.out.println(str2);

        System.out.println(str3);

        System.out.println(str4);

        System.out.println(str5);

    }

}

101

| 101|

|101 |

| 101|

|0000000101|

7. getBytes()

The java string getBytes() method returns the byte array of the string.

In other words, it returns *sequence of bytes.*

Signature

There *are 3 variant of getBytes() method*. The signature or syntax of string getBytes() method is given below:

**public** **byte**[] getBytes()

**public** **byte**[] getBytes(Charset charset)

**public** **byte**[] getBytes(String charsetName)**throws** UnsupportedEncodingException

Returns

sequence of bytes.

Java String getBytes() method example

**public** **class** StringGetBytesExample{

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

String s1="ABCDEFG";

**byte**[] barr=s1.getBytes();

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

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

}

}}

Output:

65

66

67

68

69

70

71

Java String getBytes() Method Example 2

This method returns a *byte array that again can be passed to String constructor to get String.*

**public** **class** StringGetBytesExample2 {

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

        String s1 = "ABCDEFG";

**byte**[] barr = s1.getBytes();

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

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

        }

        // Getting string back

        String s2 = **new** String(barr);

        System.out.println(s2);

    }

}

Output:

65

66

67

68

69

70

71

ABCDEFG

8. getChars()

The java string getChars() method copies the content of this string into specified char array.

There are *4 arguments passed in getChars() method.* The signature of getChars() method is given below:

Signature

**public** **void** getChars(**int** srcBeginIndex, **int** srcEndIndex, **char**[] destination, **int** dstBeginIndex)

Returns

*It doesn't return any value.*

Throws

It throws StringIndexOutOfBoundsException if beginIndex is greater than endIndex.

Java String getChars() method example

**public** **class** StringGetCharsExample{

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

 String str = **new** String("hello javatpoint how r u");

**char**[] ch = **new** **char**[10];

**try**{

         str.getChars(6, 16, ch, 0);

         System.out.println(ch);

      }**catch**(Exception ex){System.out.println(ex);}

}}

Output:

javatpoint

Java String getChars() Method Example 2

It throws an exception if index value exceeds array range. **public** **class** StringGetCharsExample2 {

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

        String str = **new** String("Welcome to Javatpoint");

**char**[] ch  = **new** **char**[20];

**try** {

            str.getChars(1, 26, ch, 0);

            System.out.println(ch);

        } **catch** (Exception e) {

            System.out.println(e);

        }

    }

}

Output:

java.lang.StringIndexOutOfBoundsException: offset 10, count 14, length 20

9. indexOf()

*The java string indexOf() method returns index of given character value or substring. If it is not found, it returns -1.*

*The index counter starts from zero.*

Signature

There are 4 types of indexOf method in java. The signature of indexOf methods are given below:

|  |  |  |
| --- | --- | --- |
| No. | Method | Description |
| 1 | int indexOf(int ch) | returns index position for the given char value |
| 2 | int indexOf(int ch, int fromIndex) | returns index position for the given char value and from index |
| 3 | int indexOf(String substring) | returns index position for the given substring |
| 4 | int indexOf(String substring, int fromIndex) | returns index position for the given substring and from index |

Returns

index of the stringJava String indexOf() method example

**public** **class** IndexOfExample{

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

String s1="this is index of example";

//passing substring

**int** index1=s1.indexOf("is");//returns the index of is substring

**int** index2=s1.indexOf("index");//returns the index of index substring

System.out.println(index1+"  "+index2);//2 8

//passing substring with from index

**int** index3=s1.indexOf("is",4);//returns the index of is substring after 4th index

System.out.println(index3);//5 i.e. the index of another is

//passing char value

**int** index4=s1.indexOf('s');//returns the index of s char value

System.out.println(index4);//3

}}

Op:

2 8

5

3

This method takes substring as an argument and returns index of first character of the substring.

**public** **class** IndexOfExample2 {

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

        String s1 = "This is indexOf method";

        // Passing Substring

**int** index = s1.indexOf("method"); //Returns the index of this substring

        System.out.println("index of substring "+index);

    }

}

index of substring 16

This method takes substring and index as arguments and returns index of first character occured after the given fromIndex.

**public** **class** IndexOfExample3 {

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

        String s1 = "This is indexOf method";

        // Passing substring and index

**int** index = s1.indexOf("method", 10); //Returns the index of this substring

        System.out.println("index of substring "+index);

        index = s1.indexOf("method", 20); // It returns -1 if substring does not found

        System.out.println("index of substring "+index);

    }

}

index of substring 16

index of substring -1

Java String indexOf(int char, int fromIndex) Method Example

This method takes char and index as arguments and returns index of first character occured after the given fromIndex.

**public** **class** IndexOfExample4 {

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

        String s1 = "This is indexOf method";

        // Passing char and index from

**int** index = s1.indexOf('e', 12); //Returns the index of this char

        System.out.println("index of char "+index);

    }

}

index of char 17

10. intern()

The java string intern() method returns the interned string.

It returns the *canonical representation of string.*

It can be used to return string from memory, if it is created by new keyword. It creates exact copy of heap string object in string constant pool.

Signature

**public** String intern()

Returns

interned string

ava String intern() method example

**public** **class** InternExample{

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

String s1=**new** String("hello");

String s2="hello";

String s3=s1.intern();//returns string from pool, now it will be same as s2

System.out.println(s1==s2);//false because reference variables are pointing to different instance

System.out.println(s2==s3);//true because reference variables are pointing to same instance

}}

**public** **class** InternExample2 {

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

        String s1 = "Javatpoint";

        String s2 = s1.intern();

        String s3 = **new** String("Javatpoint");

        String s4 = s3.intern();

        System.out.println(s1==s2); // True

        System.out.println(s1==s3); // False

        System.out.println(s1==s4); // True

        System.out.println(s2==s3); // False

        System.out.println(s2==s4); // True

        System.out.println(s3==s4); // False

    }

}

11. isEmpty()

The **java string isEmpty()** method checks if this string is empty or not.

It returns *true*, if length of string is 0 otherwise false. In other words, true is returned if string is empty otherwise it returns false.

The isEmpty() method of String class is included in java string since JDK 1.6.

Java String isEmpty() method example

**public** **class** IsEmptyExample{

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

String s1="";

String s2="javatpoint";

System.out.println(s1.isEmpty());

System.out.println(s2.isEmpty());

}}

true

false

Java String isEmpty() Method Example 2

**public** **class** IsEmptyExample2 {

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

        String s1="";

        String s2="Javatpoint";

        // Either length is zero or isEmpty is true

**if**(s1.length()==0 || s1.isEmpty())

            System.out.println("String s1 is empty");

**else** System.out.println("s1");

**if**(s2.length()==0 || s2.isEmpty())

            System.out.println("String s2 is empty");

**else** System.out.println(s2);

    }

}

Op:

String s1 is empty

Javatpoint

11. join()

The java string join() method returns a string joined with given delimiter.

*In string join method, delimiter is copied for each elements.*

*In case of null element, "null" is added.* The join() method is included in java string since JDK 1.8.

There are two types of join() methods in java string.

**public** **static** String join(CharSequence delimiter, CharSequence... elements)

and

**public** **static** String join(CharSequence delimiter, Iterable<? **extends** CharSequence> elements)

Parameters

delimiter : char value to be added with each element

elements : char value to be attached with delimiter

Returns

joined string with delimiter

Throws

NullPointerException if element or delimiter is null.

Java String join() method example

**public** **class** StringJoinExample{

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

String  joinString1=String.join("-","welcome","to","javatpoint");

System.out.println(joinString1);

}}

welcome-to-javatpoint

**public** **class** StringJoinExample2 {

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

        String date = String.join("/","25","06","2018");

        System.out.print(date);

        String time = String.join(":", "12","10","10");

        System.out.println(" "+time);

    }

}

25/06/2018 12:10:10

12. lastIndexOf()

The **java string lastIndexOf()** method returns last index of the given character value or substring. If it is not found, it returns -1.

The index counter starts from zero.

There are 4 types of lastIndexOf method in java. The signature of lastIndexOf methods are given below:

|  |  |  |
| --- | --- | --- |
| No. | Method | Description |
| 1 | int lastIndexOf(int ch) | returns last index position for the given char value |
| 2 | int lastIndexOf(int ch, int fromIndex) | returns last index position for the given char value and from index |
| 3 | int lastIndexOf(String substring) | returns last index position for the given substring |
| 4 | int lastIndexOf(String substring, int fromIndex) | returns last index position for the given substring and from index |

Parameters

ch: char value i.e. a single character e.g. 'a'

fromIndex: index position from where index of the char value or substring is retured

substring: substring to be searched in this string

Returns

last index of the string

Java String lastIndexOf() method example

**public** **class** LastIndexOfExample{

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

String s1="this is index of example";//there are 2 's' characters in this sentence

**int** index1=s1.lastIndexOf('s');//returns last index of 's' char value

System.out.println(index1);//6

}}

Java String lastIndexOf(int ch, int fromIndex) Method Example

Here, we are finding last index from the string by specifying *fromIndex*

**public** **class** LastIndexOfExample2 {

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

        String str = "This is index of example";

**int** index = str.lastIndexOf('s',5);

        System.out.println(index);

    }

}

Output:

3

Java String lastIndexOf(String substring) Method Example

It returns the last index of the substring.

**public** **class** LastIndexOfExample3 {

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

        String str = "This is last index of example";

**int** index = str.lastIndexOf("of");

        System.out.println(index);

    }

}

Output:

19

Java String lastIndexOf(String substring, int fromIndex) Method Example

It returns the last index of the substring from the fromIndex.

**public** **class** LastIndexOfExample4 {

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

        String str = "This is last index of example";

**int** index = str.lastIndexOf("of", 25);

        System.out.println(index);

        index = str.lastIndexOf("of", 10);

        System.out.println(index); // -1, if not found

    }

}

Output:

19

-1

13. length()

The java string length() method length of the string.

It returns count of total number of characters.

The length of java string is same as the unicode code units of the string.

Returns

length of characters

Java String length() method example

**public** **class** LengthExample{

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

String s1="javatpoint";

String s2="python";

System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string

System.out.println("string length is: "+s2.length());//6 is the length of python string

}}

14. 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.

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);

}}

OP:

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);

}}

my name was khan my name was java

Java String replace() Method Example 3

**public** **class** ReplaceExample3 {

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

        String str = "oooooo-hhhh-oooooo";

        String rs = str.replace("h","s"); // Replace 'h' with 's'

        System.out.println(rs);

        rs = rs.replace("s","h"); // Replace 's' with 'h'

        System.out.println(rs);

    }

}

oooooo-ssss-oooooo

oooooo-hhhh-oooooo

15. replaceAll()

The **java string replaceAll()** method returns a string replacing all the sequence of characters matching regex and replacement string.

Signature

**public** String replaceAll(String regex, String replacement)

Parameters

regex : regular expression

replacement : replacement sequence of characters

Java String replaceAll() example: replace character

an example to replace all the occurrences of **a single character**.

**public** **class** ReplaceAllExample1{

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

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

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

System.out.println(replaceString);

}}

jevetpoint is e very good website

Java String replaceAll() example: replace word

an example to replace all the occurrences of **single word or set of words**.

**public** **class** ReplaceAllExample2{

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

String s1="My name is Khan. My name is Bob. My name is Sonoo.";

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

System.out.println(replaceString);

}}

My name was Khan. My name was Bob. My name was Sonoo.

an example to remove all the occurrences of **white spaces**.

**public** **class** ReplaceAllExample3{

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

String s1="My name is Khan. My name is Bob. My name is Sonoo.";

String replaceString=s1.replaceAll("\\s","");

System.out.println(replaceString);

}}

MynameisKhan.MynameisBob.MynameisSonoo.

16. split()

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

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

Signature

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

**public** String split(String regex)

and,

**public** String split(String regex, **int** limit)

*Returns*

*array of strings*

Throws

PatternSyntaxException if pattern for regular expression is invalid

**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);

}

}}

OP:

java

string

split

method

by

javatpoint

*Java String split() method with regex and length example*

**public** **class** SplitExample2{

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

String s1="welcome to split world";

System.out.println("returning words:");

**for**(String w:s1.split("\\s",0)){

System.out.println(w);

}

System.out.println("returning words:");

**for**(String w:s1.split("\\s",1)){

System.out.println(w);

}

System.out.println("returning words:");

**for**(String w:s1.split("\\s",2)){

System.out.println(w);

}

}}

returning words:

welcome

to

split

world

returning words:

welcome to split world

returning words:

welcome

to split world

*Java String split() method with regex and length example 2*

Here, we are passing split limit as a second argument to this function. This limits the number of splitted strings.

**public** **class** SplitExample3 {

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

        String str = "Javatpointtt";

        System.out.println("Returning words:");

        String[] arr = str.split("t", 0);

**for** (String w : arr) {

            System.out.println(w);

        }

        System.out.println("Split array length: "+arr.length);

    }

}

Returning words:

Java

poin

Split array length: 2

17. startsWith()

The java string startsWith() method checks if this string starts with given prefix. It returns true if this string starts with given prefix else returns false.

Signature

The syntax or signature of startWith() method is given below.

public boolean startsWith(String prefix)

public boolean startsWith(String prefix, int offset)

Parameter

prefix : Sequence of character

Returns

true or false

Java String startsWith() method example

public class StartsWithExample{

public static void main(String args[]){

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

System.out.println(s1.startsWith("ja"));

System.out.println(s1.startsWith("java string"));

}}

Output:

true

true

Java String startsWith(String prefix, int offset) Method Example

This is overloaded method of startWith() method which is used to pass one extra argument (offset) to the function. This method works from the passed offset.

public class StartsWithExample2 {

    public static void main(String[] args) {

        String str = "Javatpoint";

        System.out.println(str.startsWith("J")); // True

        System.out.println(str.startsWith("a")); // False

        System.out.println(str.startsWith("a",1)); // True

    }

}

*18. substring()*

The java string substring() method returns a part of the string.

We pass begin index and end index number position in the java substring method where start index is inclusive and end index is exclusive. In other words, start index starts from 0 whereas end index starts from 1.

There are two types of substring methods in java string.

Signature

**public** String substring(**int** startIndex)

and

**public** String substring(**int** startIndex, **int** endIndex)

If you don't specify endIndex, java substring() method will return all the characters from startIndex.

Parameters

startIndex : starting index is inclusive

endIndex : ending index is exclusive

Returns

specified string

Throws

StringIndexOutOfBoundsException if start index is negative value or end index is lower than starting index.

Java String substring() method example

**public** **class** SubstringExample{

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

String s1="javatpoint";

System.out.println(s1.substring(2,4));//returns va

System.out.println(s1.substring(2));//returns vatpoint

}}

Java String substring() Method Example 2

**public** **class** SubstringExample2 {

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

        String s1="Javatpoint";

        String substr = s1.substring(0); // Starts with 0 and goes to end

        System.out.println(substr);

        String substr2 = s1.substring(5,10); // Starts from 5 and goes to 10

        System.out.println(substr2);

        String substr3 = s1.substring(5,15); // Returns Exception

    }

}

OP:

Javatpoint

point

Exception in thread "main" java.lang.StringIndexOutOfBoundsException: begin 5, end 15, length 10

19. toCharArray()

The java string toCharArray() method converts this string into character array. It returns a newly created character array, its length is similar to this string and its contents are initialized with the characters of this string.

Signature

The signature or syntax of string toCharArray() method is given below:

**public** **char**[] toCharArray()

Returns

character array

Java String toCharArray() method example

**public** **class** StringToCharArrayExample{

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

String s1="hello";

**char**[] ch=s1.toCharArray();

**for**(**int** i=0;i<ch.length;i++){

System.out.print(ch[i]);

}

}}

*20. toLowerCase()*

The java string toLowerCase() method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale.

Java String toLowerCase() method example

**public** **class** StringLowerExample{

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

String s1="JAVATPOINT HELLO stRIng";

String s1lower=s1.toLowerCase();

System.out.println(s1lower);

}}

*21.toUpperCase()*

The java string toUpperCase() method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

Java String toUpperCase() method example

**public** **class** StringUpperExample{

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

String s1="hello string";

String s1upper=s1.toUpperCase();

System.out.println(s1upper);

}}

21. 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.

*Note:*

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

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()

}}

OP:

hello string javatpoint

hello stringjavatpoint

Java String trim() Method Example 2

This example demonstrate the use of trim method. This method removes all the trailing spaces so the length of string also reduces. Let's see an example.

**public** **class** StringTrimExample {

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

        String s1 ="  hello java string   ";

        System.out.println(s1.length());

        System.out.println(s1); //Without trim()

        String tr = s1.trim();

        System.out.println(tr.length());

        System.out.println(tr); //With trim()

    }

}

22

hello java string

17

hello java string

*22. valueOf()*

The java string valueOf() method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

Java String valueOf() Complete Examples

an example where we are converting all primitives and objects into strings.

**public** **class** StringValueOfExample5 {

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

**boolean** b1=**true**;

**byte** b2=11;

**short** sh = 12;

**int** i = 13;

**long** l = 14L;

**float** f = 15.5f;

**double** d = 16.5d;

**char** chr[]={'j','a','v','a'};

        StringValueOfExample5 obj=**new** StringValueOfExample5();

        String s1 = String.valueOf(b1);

        String s2 = String.valueOf(b2);

        String s3 = String.valueOf(sh);

        String s4 = String.valueOf(i);

        String s5 = String.valueOf(l);

        String s6 = String.valueOf(f);

        String s7 = String.valueOf(d);

        String s8 = String.valueOf(chr);

        String s9 = String.valueOf(obj);

        System.out.println(s1);

        System.out.println(s2);

        System.out.println(s3);

        System.out.println(s4);

        System.out.println(s5);

        System.out.println(s6);

        System.out.println(s7);

        System.out.println(s8);

        System.out.println(s9);

    }

}

Output:

true

11

12

13

14

15.5

16.5

java

StringValueOfExample5@2a139a55

String vs StringBuilder vs StringBuffer in Java

// Java program to demonstrate difference between String,

// StringBuilder and StringBuffer

class Geeksforgeeks

{

// Concatenates to String

public static void concat1(String s1)

{

s1 = s1 + "forgeeks";

}

// Concatenates to StringBuilder

public static void concat2(StringBuilder s2)

{

s2.append("forgeeks");

}

// Concatenates to StringBuffer

public static void concat3(StringBuffer s3)

{

s3.append("forgeeks");

}

public static void main(String[] args)

{

String s1 = "Geeks";

concat1(s1); // s1 is not changed

System.out.println("String: " + s1);

StringBuilder s2 = new StringBuilder("Geeks");

concat2(s2); // s2 is changed

System.out.println("StringBuilder: " + s2);

StringBuffer s3 = new StringBuffer("Geeks");

concat3(s3); // s3 is changed

System.out.println("StringBuffer: " + s3);

}

}

Output:

String: Geeks

StringBuilder: Geeksforgeeks

StringBuffer: Geeksforgeeks

Explanation:  
1. Concat1 : In this method, we pass a string “Geeks” and perform “s1 = s1 + ”forgeeks”. The string passed from main() is not changed, this is due to the fact that String is immutable. Altering the value of string creates another object and s1 in concat1() stores reference of new string. References s1 in main() and cocat1() refer to different strings.

2. Concat2 : In this method, we pass a string “Geeks” and perform “s2.append(“forgeeks”)” which changes the actual value of the string (in main) to “Geeksforgeeks”. This is due to the simple fact that StringBuilder is mutable and hence changes its value.

2. Concat3 : StringBuffer is similar to StringBuilder except one difference that StringBuffer is thread safe, i.e., multiple threads can use it without any issue. The thread safety brings a penalty of performance.  
When to use which one :

If a string is going to remain constant throughout the program, then use String class object because a String object is immutable.

If a string can change (example: lots of logic and operations in the construction of the string) and will only be accessed from a single thread, using a StringBuilder is good enough.

If a string can change, and will be accessed from multiple threads, use a StringBuffer because StringBuffer is synchronous so you have thread-safety.

**Conversion between types of strings in Java:**

// Java program to demonstrate conversion from

// String to StringBuffer and StringBuilder.

public class Test

{

public static void main(String[] args)

{

String str = "Geeks";

// conversion from String object to StringBuffer

StringBuffer sbr = new StringBuffer(str);

sbr.reverse();

System.out.println(sbr);

// conversion from String object to StringBuilder

StringBuilder sbl = new StringBuilder(str);

sbl.append("ForGeeks");

System.out.println(sbl);

}

}

Output:

skeeG

GeeksForGeeks

**From StringBuffer and StringBuilder to String :** This conversions can be perform using toString() method which is overridden in both StringBuffer and StringBuilder classes.

while we use toString() method, a new String object(in Heap area) is allocated and initialized to character sequence currently represented by StringBuffer object, that means the subsequent changes to the StringBuffer object do not affect the contents of the String object.

// Java program to demonstrate conversion from

// String to StringBuffer and StringBuilder.

public class Test

{

public static void main(String[] args)

{

StringBuffer sbr = new StringBuffer("Geeks");

StringBuilder sbdr = new StringBuilder("Hello");

// conversion from StringBuffer object to String

String str = sbr.toString();

System.out.println("StringBuffer object to String : ");

System.out.println(str);

// conversion from StringBuilder object to String

String str1 = sbdr.toString();

System.out.println("StringBuilder object to String : ");

System.out.println(str1);

// changing StringBuffer object sbr

// but String object(str) doesn't change

sbr.append("ForGeeks");

System.out.println(sbr);

System.out.println(str);

}

}

Output:

StringBuffer object to String :

Geeks

StringBuilder object to String :

Hello

GeeksForGeeks

Geeks

**From StringBuffer to StringBuilder or vice-versa :**This conversion is tricky.There is no direct way to convert the same. In this case, We can use a String class object. We first convert StringBuffer/StringBuilder object to String using toString() method and then from String to StringBuilder/StringBuffer using constructors.

// Java program to demonstrate conversion from

// String to StringBuffer and StringBuilder.

public class Test

{

public static void main(String[] args)

{

StringBuffer sbr = new StringBuffer("Geeks");

// conversion from StringBuffer object to StringBuilder

String str = sbr.toString();

StringBuilder sbl = new StringBuilder(str);

System.out.println(sbl);

}

}

Output:

Geeks

Note:

Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable.

StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for single threaded program. If thread safety is needed, then StringBuffer is used.

**ARRAYS:**

*an array is a collection of similar type of elements (same data types) which have a contiguous memory location.*

*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.

we can get the length of the array using the length member.

*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.

Single Dimensional Array

Multidimensional Array

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];

//Program to illustrate how to declare, instantiate, initialize and

traverse the Java array.

class Testarray{

public static void main(String args[]){

int a[]=new int[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//traversing array

for(int i=0;i<a.length;i++)//length is the property of array

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

}}

declare, instantiate and initialize the java array together by:

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

//Java Program to illustrate the use of declaration, instantiation

and initialization of Java array in a single line

class Testarray1{

public static void main(String args[]){

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

//printing array

for(int i=0;i<a.length;i++)//length is the property of array

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

}}

|  |  |
| --- | --- |
| Array properties |  |
| length | returns length / size of an array which is equal to the total number of elements in an array |
| hashcode() | The hashcode of a Java Object is simply a number, it is 32-bit signed int,that allows an object to be managed by a hash-based data structure.  hash code is an unique id number allocated to an object by JVM.  *If two objects are equals then these two objects should return same hash code.*  Java Object hashCode() is a native method and returns the *integer hash code value of the object.*  The general contract of hashCode() method is: ...  An object hash code value can change in multiple executions of the same application.  If two objects are equal according to equals() method, then their hash code must be same. |
| clone() | Clone() method in Java.  *Object cloning refers to creation of exact copy of an object.*  It creates a new instance of the class of current object and initializes all its fields with exactly  the contents of the corresponding fields of this object.  Using Assignment Operator to create copy of reference variable.  The clone() method saves the extra processing task for creating the exact copy of an object.  If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.   An array type has a public method clone() , which overrides the clone() method of class Object .  *An array type inherits all methods except clone from Object class.*  *The clone method of an array type returns a duplicate copy of the same array.*  Also note that every Java array implements the interfaces Cloneable and Java.  Java provides a clone method that copies the attributes of one object into another using shallow copy.  Cloning Arrays in Java and other objects sometimes behave differently.  *Java Language Specification states that a clone of a multidimensional array is shallow, which is to say that it creates only a single new array.* |
| getClass().getName() | The java.lang.Object.getClass() method returns the *runtime class of an object.*  The *Class object is the object that is locked by static synchronized methods of the represented class.* getClass returns a Class object that represents the object's class.  getName then returns the name of that class as a string. |

For-each Loop for Java Array

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

}

//Java Program to print the array elements using for-each loop

class Testarray1{

public static void main(String args[]){

int arr[]={33,3,4,5};

//printing array using for-each loop

for(int i:arr)

System.out.println(i);

}}

Passing Array to a Method in Java

//Java Program to demonstrate the way of passing an array

//to method.

class Testarray2{

//creating a method which receives an array as a parameter

static void min(int arr[]){

int min=arr[0];

for(int i=1;i<arr.length;i++)

 if(min>arr[i])

  min=arr[i];

System.out.println(min);

}

public static void main(String args[]){

int a[]={33,3,4,5};//declaring and initializing an array

min(a);//passing array to method

}}

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.

//Java Program to demonstrate the way of passing an anonymous array

//to method.

public class TestAnonymousArray{

//creating a method which receives an array as a parameter

static void printArray(int arr[]){

for(int i=0;i<arr.length;i++)

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

}

public static void main(String args[]){

printArray(new int[]{10,22,44,66});//passing anonymous array to method

}}

Returning Array from the Method

//Java Program to return an array from the method

class TestReturnArray{

//creating method which returns an array

static int[] get(){

return new int[]{10,30,50,90,60};

}

public static void main(String args[]){

//calling method which returns an array

int arr[]=get();

//printing the values of an array

for(int i=0;i<arr.length;i++)

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

}}

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]);

}

}}

Multidimensional Array in Java

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

//Java Program to illustrate the use of multidimensional array

class Testarray3{

public static void main(String args[]){

//declaring and initializing 2D array

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

//printing 2D array

for(int i=0;i<3;i++){

 for(int j=0;j<3;j++){

   System.out.print(arr[i][j]+" ");

 }

 System.out.println();

}

}}

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.

**//Java Program to illustrate the jagged array**

**class** TestJaggedArray{

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

        //declaring a 2D array with odd columns

**int** arr[][] = **new** **int**[**3**][];

        arr[0] = **new** **int**[3];

        arr[1] = **new** **int**[4];

        arr[2] = **new** **int**[2];

        //initializing a jagged array

**int** count = 0;

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

**for**(**int** j=0; j<arr[i].length; j++)

                arr[i][j] = count++;

        //printing the data of a jagged array

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

**for** (**int** j=0; j<arr[i].length; j++){

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

            }

            System.out.println();//new line

        }

    }

}

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.

//Java Program to get the class name of array in Java

class Testarray4{

public static void main(String args[]){

//declaration and initialization of array

int arr[]={4,4,5};

//getting the class name of Java array

Class c=arr.getClass();

String name=c.getName();

//printing the class name of Java array

System.out.println(name);

}}

Copying a Java Array

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

**Syntax of arraycopy method**

**public** **static** **void** arraycopy(

Object src, **int** srcPos,Object dest, **int** destPos, **int** length

)

//Java Program to copy a source array into a destination array in Java

class TestArrayCopyDemo {

    public static void main(String[] args) {

        //declaring a source array

        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',

                'i', 'n', 'a', 't', 'e', 'd' };

        //declaring a destination array

        char[] copyTo = new char[7];

        //copying array using System.arraycopy() method

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);

        //printing the destination array

        System.out.println(String.valueOf(copyTo));

    }

}

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.*

//Java Program to clone the array

class Testarray1{

public static void main(String args[]){

int arr[]={33,3,4,5};

System.out.println("Printing original array:");

for(int i:arr)

System.out.println(i);

System.out.println("Printing clone of the array:");

int carr[]=arr.clone();

for(int i:carr)

System.out.println(i);

System.out.println("Are both equal?");

System.out.println(arr==carr);

}}

Addition of 2 Matrices in Java

//Java Program to demonstrate the addition of two matrices in Java

class Testarray5{

public static void main(String args[]){

//creating two matrices

int a[][]={{1,3,4},{3,4,5}};

int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices

int c[][]=new int[2][3];

//adding and printing addition of 2 matrices

for(int i=0;i<2;i++){

for(int j=0;j<3;j++){

c[i][j]=a[i][j]+b[i][j];

System.out.print(c[i][j]+" ");

}

System.out.println();//new line

}

}}

Multiplication of 2 Matrices in Java

//Java Program to multiply two matrices

public class MatrixMultiplicationExample{

public static void main(String args[]){

//creating two matrices

int a[][]={{1,1,1},{2,2,2},{3,3,3}};

int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two matrices

int c[][]=new int[3][3];  //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices

for(int i=0;i<3;i++){

for(int j=0;j<3;j++){

c[i][j]=0;

for(int k=0;k<3;k++)

{

c[i][j]+=a[i][k]\*b[k][j];

}//end of k loop

System.out.print(c[i][j]+" ");  //printing matrix element

}//end of j loop

System.out.println();//new line

}

}}

Data Structures and Collections

Graphical user interface, text

Description automatically generated

Arrays: fixed size, to resize the array, move the data to another array of correct size, delete the old array, re-create it with new size and copy the data.To overcome this, Collections are in place.

A screenshot of a cell phone

Description automatically generated

***collection is a concept and is different way of holding data.***

***Data Structures hold collections of data in various formats.***

***Data structure is all about store and operate on data efficiently.***

***Abstract Data Type in Data Structures:***

*Simple data type- primitive*

*Complex- user defined*

***Abstract Data Types – List,Queue,Stack and Map***

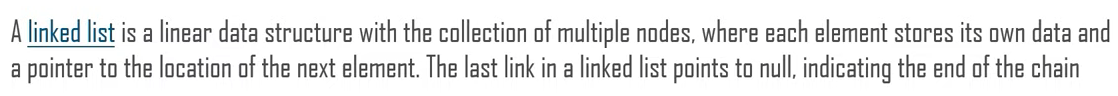
***Note:***

*few Data Types are called as abstract as they are concepts and have different implementations in each programming language.*

***Overview of various collections:***

***Text

Description automatically generated***

******

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

***List :***

*Get a* ***specific item*** *by giving its* ***position*** *in the list.*

***Search*** *the list for a* ***specific value.***

***Iterate*** *through the list.*

*Find those* ***items*** *within a* ***specific range.***

1. ***ArrayList***

Has all the features of list.

***Add*** *and* ***delete******items*** *in* ***any position.***

ArrayList are of **specific type** and are specified in angular brackets

***ArrayList <String> names = new ArrayList<> ();***

**Iterate** through an **ArrayList** using a **simplified loop.**

for (String s : names){

System.out.println(s) }

***B. LinkedList***

Uses the concept of **Data and reference.**

***each Element*** *is a* ***Node****.*

*Each node will have element info/value and address of next node.*

**First node** is called as **Head**.

**Last node** will have **null for address.**

Has all the features of list.

*Faster than ArrayList*

**Add** and **delete** items in any position more easily.

LinkedList are of **specific type** and are specified in angular brackets.

***LinkedList <String> names = new LinkedList <> ();***

**Iterate** through an **LinkedList** using a **simplified loop.**

for (String s : names){

System.out.println(s)

**Set**

*No duplicate values /* ***Unique*** *values*

Items can be added/removed easily.

Fast

***Order is not necessarily maintained.***

***Tree set type orders the data.***

***Queue***

It follows **FIFO**

***Abstract class*** *which relies on* ***other data types.***

***Queue<String> q = new LinkedList<String> ();***

***add, peek and poke*** *methods.*

***Map***

***Abstract class***

Good for **mathematical concept of sets** – intersection, subset, non – intersecting sections.

***Key value pair.***

*Each* ***key has to be unique.***

***Set is an interface , to implement it will use HashSet which is a class****.*

*Set<Integer> numbers = new HashSet<>(); // gives data in random order*

*Set<Integer> numbers = new TreeSet<>(); // data in proper order*

***Map is an interface to implement it will use HashMap which is a class.***

*Map<String, Integer> contacts = new HashMap<> ();*

***List – processing items.***

***Sets – unique items***

***Queue – for queue***

***Maps – manipulating sets of data.***

***Collections and Generics***

***collection – concept***

***Collection – Interface***

***Collections – Class***

*Collection is an interface extended by List which is also an interface been implemented by ArrayList which is a class.*

*Ex:*

*Collection values = new ArrayList ();*

*values.add(5);*

*values.add(8);*

*values.add(“String”);*

*values.add(“10.9087”)*

***Note: Array is faster than Collection, if size is fixed Array should be preferred.***

***Note: Data type by default is Object type.***

*To specify the data type, we use* ***Generics*** *in Collection.*

*Ex:*

*Collection<Integer> values = new ArrayList <Integer>();*

*values.add(5);*

*values.add(8);*

() – functions / methods

{} - definitions

[] – Arrays

**<>- Generics**

***Note: Collection does not work with index number.***

*To* ***add items in between*** *exiting ones use* ***List*** *or* ***Set*** *instead of Collection which works with index.*

***Generics:***

The idea is to allow type (Integer, String, … etc and user defined types) to be *a parameter to methods, classes and interfaces.*

They were designed to extend Java's type system to allow

"a type or method to operate on objects of various types while providing ***compile-time type safety".***

classes like HashSet, ArrayList, HashMap, etc use generics very well.

// To create an instance of generic class

BaseType<Type>obj = new BaseType<Type>()

*Note: In Parameter type we cannot use primitives like 'int','char' or 'double'.*

*// A Simple Java program to show working of user defined Generic classes*

// ***We use <> to specify Parameter type***

class Test<T>

{

// An object of type T is declared

T obj; `

Test(T obj) { this.obj = obj; } // constructor

public T getObject() { return this.obj; }

}

// Driver class to test above

class Main

{

public static void main (String[] args)

{

// instance of Integer type

Test <Integer> iObj = new Test<Integer>(15);

System.out.println(iObj.getObject());

// instance of String type

Test <String> sObj =

new Test<String>("GeeksForGeeks");

System.out.println(sObj.getObject());

}

}

Output:

15

GeeksForGeeks

pass multiple Type parameters in Generic classes.

// A Simple Java program to show multiple type parameters in Java Generics

// We use <> to specify Parameter type

class Test<T, U>

{

T obj1; // An object of type T

U obj2; // An object of type U

// constructor

Test(T obj1, U obj2)

{

this.obj1 = obj1;

this.obj2 = obj2;

}

// To print objects of T and U

public void print()

{

System.out.println(obj1);

System.out.println(obj2);

}

}

// Driver class to test above

class Main

{

public static void main (String[] args)

{

Test <String, Integer>obj =

new Test<String, Integer>("GfG", 15);

obj.print();

}

}

Output:

GfG

15

Generic Functions:

*We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.*

// A Simple Java program to show working of user defined

// Generic functions

class Test

{

// A Generic method example

static <T> void genericDisplay (T element)

{

System.out.println(element.getClass().getName() +

" = " + element);

}

// Driver method

public static void main(String[] args)

{

// Calling generic method with Integer argument

genericDisplay(11);

// Calling generic method with String argument

genericDisplay("GeeksForGeeks");

// Calling generic method with double argument

genericDisplay(1.0);

}

}

Output :

*java.lang.Integer = 11*

*java.lang.String = GeeksForGeeks*

*java.lang.Double = 1.0*

Advantages of Generics:

Code Reuse:

We can write a method/class/interface once and use for any type we want.

Type Safety :

*Generics make errors to appear during compile time than at run time.*

(It’s always better to know problems in your code at compile time rather .

than making your code fail at run time).

Suppose to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

*// A Simple Java program to demonstrate that NOT using generics can cause run time exceptions*

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creatinga an ArrayList without any type specified

ArrayList al = new ArrayList();

al.add("Sachin");

al.add("Rahul");

al.add(10); // Compiler allows this

String s1 = (String)al.get(0);

String s2 = (String)al.get(1);

// Causes Runtime Exception

String s3 = (String)al.get(2);

}

}

Output :

Exception in thread "main" java.lang.ClassCastException:

java.lang.Integer cannot be cast to java.lang.String

at Test.main(Test.java:19)

How generics solve this problem?  
At the time of defining ArrayList, we can specify that this list can take only String objects.

// Using generics converts run time exceptions into

// compile time exception.

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creating a an ArrayList with String specified

ArrayList <String> al = new ArrayList<String> ();

al.add("Sachin");

al.add("Rahul");

// Now Compiler doesn't allow this

al.add(10);

String s1 = (String)al.get(0);

String s2 = (String)al.get(1);

String s3 = (String)al.get(2);

}

}

Output:

15: error: no suitable method found for add(int)

al.add(10);

Individual Type Casting is not needed:

If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

// We don't need to typecast individual members of ArrayList

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creating a an ArrayList with String specified

ArrayList <String> al = new ArrayList<String> ();

al.add("Sachin");

al.add("Rahul");

// Typecasting is not needed

String s1 = al.get(0);

String s2 = al.get(1);

}

}

*Implementing generic algorithms:*

By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

Data Structures and Collection in detail:

Collections in Java:

*The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.*

Java Collections can achieve all the operations that you perform on a data such as *searching, sorting, insertion, manipulation, and deletion.*

*Java Collection means a single unit of objects.*

Java Collection framework provides many

interfaces (Set, List, Queue, Deque) and

classes ([**ArrayList**](https://www.javatpoint.com/java-arraylist), Vector, [**LinkedList**](https://www.javatpoint.com/java-linkedlist), **[PriorityQueue](https://www.javatpoint.com/java-priorityqueue)**, HashSet, LinkedHashSet, TreeSet).

Note:

“framework in Java”

It provides readymade architecture.

It represents a set of classes and interfaces.

It is optional.

Collection framework:

The Collection framework represents a unified architecture for storing and manipulating a group of objects.

It has:

Interfaces and its implementations, i.e.,

classes

Algorithm

Hierarchy of Collection Framework

Note:

*The****java.util****package contains all the classes and interfaces for the*

*Collection framework.*

A screenshot of a cell phone

Description automatically generated

*Methods of Collection interface*

|  |  |  |
| --- | --- | --- |
| No. | Method | Description |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public booleanaddAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public booleanremoveAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default booleanremoveIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public booleanretainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public booleancontainsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array.  Here, the runtime type of the returned array is that of the specified array. |
| 14 | public booleanisEmpty() | It checks if collection is empty. |
| 15 | default Stream<E>parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E>spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

List (Interface):

The Java.util.List is a child interface of Collection.

It is an ordered collection of objects in which duplicate values can be stored.

List preserves the insertion order, it allows positional access and insertion of elements.

List Interface is implemented by the classes of ArrayList, LinkedList, Vector and Stack.

public abstract interface List extends Collection

**Creating List Objects:**

List a = new ArrayList();

List b = new LinkedList();

List c = new Vector();

List d = new Stack();

Generic List Object:  
After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List. The type-safe List can be defined in the following way:

// Obj is the type of object to be stored in List

List<Obj> list = new ArrayList<Obj> ();

**Operations on List:**

**Positional Access**

List allows add, remove, get and set operations based on numerical positions of elements in List. List provides following methods for these operations:

void add(int index, Object O): This method adds given element at specified index.

boolean addAll(int index, Collection c): This method adds all elements from specified collection to list. First element gets inserted at given index. If there is already an element at that position, that element and other subsequent elements(if any) are shifted to the right by increasing their index.

Object remove(int index): This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.

Object get(int index): This method returns element at the specified index.

Object set(int index, Object new): This method replaces element at given index with new element. This function returns the element which was just replaced by new element.

// Java program to demonstrate positional access

// operations on List interface

import java.util.\*;

public class ListDemo {

public static void main(String[] args)

{

// Creating a list

List<Integer> l1 = new ArrayList<Integer>();

l1.add(0, 1); // adds 1 at 0 index

l1.add(1, 2); // adds 2 at 1 index

System.out.println(l1); // [1, 2]

// Creating another list

List<Integer> l2 = new ArrayList<Integer>();

l2.add(1);

l2.add(2);

l2.add(3);

// Will add list l2 from 1 index

l1.addAll(1, l2);

System.out.println(l1);

// Removes element from index 1

l1.remove(1);

System.out.println(l1); // [1, 2, 3, 2]

// Prints element at index 3

System.out.println(l1.get(3));

// Replace 0th element with 5

l1.set(0, 5);

System.out.println(l1);

}

}

Output:

[1, 2]

[1, 1, 2, 3, 2]

[1, 2, 3, 2]

2

[5, 2, 3, 2]

**Search:**  
List provides methods to search element and returns its numeric position. Following two methods are supported by List for this operation:

int indexOf(Object o): This method returns first occurrence of given element or -1 if element is not present in list.

int lastIndexOf(Object o):This method returns the last occurrence of given element or -1 if element is not present in list.

// Java program to demonstrate search

// operations on List interface

import java.util.\*;

public class ListDemo {

public static void main(String[] args)

{

// Type safe array list, stores only string

List<String> l = new ArrayList<String>(5);

l.add("Geeks");

l.add("for");

l.add("Geeks");

// Using indexOf() and lastIndexOf()

System.out.println("first index of Geeks:"

+ l.indexOf("Geeks"));

System.out.println("last index of Geeks:"

+ l.lastIndexOf("Geeks"));

System.out.println("Index of element"

+ " not present : "

+ l.indexOf("Hello"));

}

}

Output:

first index of Geeks:0

last index of Geeks:2

Index of element not present : -1

**Iteration:**

ListIterator(extends Iterator) is used to iterate over List element. List iterator is bidirectional iterator.

List Interface provides a method to get the List view of the portion of given List between two indices. Following is the method supported by List for range view operation.

List subList(int fromIndex, int toIndex):This method returns List view of specified List between fromIndex(inclusive) and toIndex(exclusive).

// Java program to demonstrate subList operation

// on List interface.

import java.util.\*;

public class ListDemo {

public static void main(String[] args)

{

// Type safe array list, stores only string

List<String> l = new ArrayList<String>(5);

l.add("GeeksforGeeks");

l.add("Practice");

l.add("GeeksQuiz");

l.add("IDE");

l.add("Courses");

List<String> range = new ArrayList<String>();

// Return List between 2nd(including)

// and 4th element(excluding)

range = l.subList(2, 4);

System.out.println(range);

}

}

Output:

[GeeksQuiz, IDE]

ArrayList (Class):

ArrayList is a part of collection framework and is present in java.util package.

ArrayList inherits AbstractList class and implements List interface.

ArrayList is initialized by a size, however the *size can increase if collection grows or shrunk if objects are removed from the collection.*

Java ArrayList allows us to randomly access the list.

ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases

Java ArrayList class uses a dynamic array for storing the elements.

Java ArrayList class can contain duplicate elements.

Java ArrayList class maintains insertion order.

Java ArrayList class is non synchronized.

In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

Constructors of Java ArrayList

ArrayList(): This constructor is used to build an empty array list

ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from collection c.

ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified

// Creating generic integer ArrayList

ArrayList<Integer>arrlist = new ArrayList<Integer>();

// Java program to demonstrate working of ArrayList in Java

import java.io.\*;

import java.util.\*;

class arrayli

{

public static void main(String[] args)

throws IOException

{

// size of ArrayList

int n = 5;

//declaring ArrayList with initial size n

ArrayList<Integer>arrli = new ArrayList<Integer>(n);

// Appending the new element at the end of the list

for (int i=1; i<=n; i++)

arrli.add(i);

// Printing elements

System.out.println(arrli);

// Remove element at index 3

arrli.remove(3);

// Displaying ArrayList after deletion

System.out.println(arrli);

// Printing elements one by one

for (int i=0; i<arrli.size(); i++)

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

}

}

Output:

[1, 2, 3, 4, 5]

[1, 2, 3, 5]

1 2 3 5

Methods in ArrayList:

forEach​(Consumer<? super E> action): Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.

retainAll​(Collection<?> c): Retains only the elements in this list that are contained in the specified collection.

removeIf​(Predicate<? super E> filter): Removes all of the elements of this collection that satisfy the given predicate.

contains​(Object o): Returns true if this list contains the specified element.

remove​(int index): Removes the element at the specified position in this list.

remove​(Object o): Removes the first occurrence of the specified element from this list, if it is present.

get​(int index): Returns the element at the specified position in this list.

subList​(int fromIndex, int toIndex): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

spliterator​(): Creates a late-binding and fail-fast Spliterator over the elements in this list.

set​(int index, E element): Replaces the element at the specified position in this list with the specified element.

size​(): Returns the number of elements in this list.

removeAll​(Collection<?> c): Removes from this list all of its elements that are contained in the specified collection.

ensureCapacity​(int minCapacity): Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

listIterator​(): Returns a list iterator over the elements in this list (in proper sequence).

listIterator​(int index): Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.

isEmpty​(): Returns true if this list contains no elements.

removeRange​(int fromIndex, int toIndex): Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.

void clear(): This method is used to remove all the elements from any list.

void add(int index, Object element): This method is used to insert a specific element at a specific position index in a list.

void trimToSize(): This method is used to trim the capacity of the instance of the ArrayLis to the list’s current size.

int indexOf(Object O): The index of the first occurrence of a specific element is either returned, or -1 in case the element is not in the list.

int lastIndexOf(Object O): The index the last occurrence of a specific element is either returned, or -1 in case the element is not in the list.

Object clone(): This method is used to return a shallow copy of an ArrayList.

Object[] toArray(): This method is used to return an array containing all of the elements in the list in correct order.

Object[] toArray(Object[] O): It is also used to return an array containing all of the elements in this list in the correct order same as the previous method.

boolean addAll(Collection C): This method is used to append all the elements from a specific collection to the end of the mentioned list, in such a order that the values are returned by the specified collection’s iterator.

boolean add(Object o): This method is used to append a specificd element to the end of a list.

boolean addAll(int index, Collection C): Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list.

Differences between Array and ArrayList:

An array is basic functionality provided by Java.

ArrayList is part of collection framework in Java.

Therefore array members are accessed using [],

while ArrayList has a set of methods to access elements and modify them.

// A Java program to demonstrate differences between array

// and ArrayList

import java.util.ArrayList;

import java.util.Arrays;

class Test

{

public static void main(String args[])

{

/\* ........... Normal Array............. \*/

int[] arr = new int[2];

arr[0] = 1;

arr[1] = 2;

System.out.println(arr[0]);

/\*............ArrayList..............\*/

// Create an arrayList with initial capacity 2

ArrayList<Integer>arrL = new ArrayList<Integer>(2);

// Add elements to ArrayList

arrL.add(1);

arrL.add(2);

// Access elements of ArrayList

System.out.println(arrL.get(0));

}

}

Output:

1

1

Array is a fixed size data structure while ArrayList is not.

One *need not to mention the size of Arraylist while creating its object.*

Even if we specify some initial capacity, we can add more elements.

// A Java program to demonstrate differences between array

// and ArrayList

import java.util.ArrayList;

import java.util.Arrays;

class Test

{

public static void main(String args[])

{

/\* ........... Normal Array............. \*/

// Need to specify the size for array

int[] arr = new int[3];

arr[0] = 1;

arr[1] = 2;

arr[2] = 3;

// We cannot add more elements to array arr[]

/\*............ArrayList..............\*/

// Need not to specify size

ArrayList<Integer>arrL = new ArrayList<Integer>();

arrL.add(1);

arrL.add(2);

arrL.add(3);

arrL.add(4);

// We can add more elements to arrL

System.out.println(arrL);

System.out.println(Arrays.toString(arr));

}

}

Output:

[1, 2, 3, 4]

[1, 2, 3]

Array can contain both primitive data types as well as objects of a class depending on the definition of the array.

However, ArrayList only supports object entries, not the primitive data types.

Note: When we do arraylist.add(1); : it converts the primitive int data type into an Integer object.

import java.util.ArrayList;

class Test

{

public static void main(String args[])

{

// allowed

int[] array = new int[3];

// allowed, however, need to be intialized

Test[] array1 = new Test[3];

// not allowed (Uncommenting below line causes

// compiler error)

// ArrayList<char>arrL = new ArrayList<char>();

// Allowed

ArrayList<Integer> arrL1 = new ArrayList<>();

ArrayList<String> arrL2 = new ArrayList<>();

ArrayList<Object> arrL3 = new ArrayList<>();

}

}

Since ArrayList can’t be created for primitive data types, members of ArrayList are always references to objects at different memory locations Therefore in ArrayList, the actual objects are never stored at contiguous locations. References of the actual objects are stored at contiguous locations.  
In array, it depends whether the arrays is of primitive type or object type.

In case of primitive types, actual values are contiguous locations, but in case of objects, allocation is similar to ArrayList.

LinkedList (Class):

Java LinkedList class uses a doubly linked list to store the elements.

It provides a linked-list data structure.

It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

Java LinkedList class can contain duplicate elements.

Java LinkedList class maintains insertion order.

Java LinkedList class is non synchronized.

In Java LinkedList class, manipulation is fast because no shifting needs to occur.

Java LinkedList class can be used as a list, stack or queue.

Linked List are linear data structures where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.

It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.  
To store the elements in a linked list we use a doubly linked list which provides a linear data structure.

**Constructors for Java LinkedList:**

LinkedList(): Used to create an empty linked list.

LinkedList(Collection C): Used to create a ordered list which contains all the elements of a specified collection, as returned by the collection’s iterator.

// Java code for Linked List implementation

import java.util.\*;

public class Test

{

public static void main(String args[])

{

// Creating object of class linked list

LinkedList<String> object = new LinkedList<String>();

// Adding elements to the linked list

object.add("A");

object.add("B");

object.addLast("C");

object.addFirst("D");

object.add(2, "E");

object.add("F");

object.add("G");

System.out.println("Linked list : " + object);

// Removing elements from the linked list

object.remove("B");

object.remove(3);

object.removeFirst();

object.removeLast();

System.out.println("Linked list after deletion: " + object);

// Finding elements in the linked list

boolean status = object.contains("E");

if(status)

System.out.println("List contains the element 'E' ");

else

System.out.println("List doesn't contain the element 'E'");

// Number of elements in the linked list

int size = object.size();

System.out.println("Size of linked list = " + size);

// Get and set elements from linked list

Object element = object.get(2);

System.out.println("Element returned by get() : " + element);

object.set(2, "Y");

System.out.println("Linked list after change : " + object);

}

}

Output:

Linked list : [D, A, E, B, C, F, G]

Linked list after deletion: [A, E, F]

List contains the element 'E'

Size of linked list = 3

Element returned by get() : F

Linked list after change : [A, E, Y]

**Methods for Java LinkedList:**

add​(int index, E element): This method Inserts the specified element at the specified position in this list.

add​(E e): This method Appends the specified element to the end of this list.

addAll​(int index, Collection c): This method Inserts all of the elements in the specified collection into this list, starting at the specified position.

addAll​(Collection c): This method Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection’s iterator.

addFirst​(E e): This method Inserts the specified element at the beginning of this list.

addLast​(E e): This method Appends the specified element to the end of this list.

clear​(): This method removes all of the elements from this list.

clone​(): This method returns a shallow copy of this LinkedList.

contains​(Object o): This method returns true if this list contains the specified element.

descendingIterator​(): This method returns an iterator over the elements in this deque in reverse sequential order.

[**element​():**](https://www.geeksforgeeks.org/linkedlist-element-method-in-java-with-%20examples/) This method retrieves, but does not remove, the head (first element) of this list.

get​(int index): This method returns the element at the specified position in this list.

getFirst​(): This method returns the first element in this list.

[**getLast​():**](https://www.geeksforgeeks.org/linkedlist-getlast-method-in-java/) This method returns the last element in this list.

indexOf​(Object o): This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

lastIndexOf​(Object o): This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

listIterator​(int index): This method returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.

offer​(E e): This method Adds the specified element as the tail (last element) of this list.

offerFirst​(E e): This method Inserts the specified element at the front of this list.

offerLast​(E e): This method Inserts the specified element at the end of this list.

peek​(): This method retrieves, but does not remove, the head (first element) of this list.

peekFirst​(): This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.

peekLast​(): This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.

poll​(): This method retrieves and removes the head (first element) of this list.

pollFirst​(): This method retrieves and removes the first element of this list, or returns null if this list is empty.

pollLast​(): This method retrieves and removes the last element of this list, or returns null if this list is empty.

pop​(): This method Pops an element from the stack represented by this list.

push​(E e): This method Pushes an element onto the stack represented by this list.

remove​(): This method retrieves and removes the head (first element) of this list.

remove​(int index): This method removes the element at the specified position in this list.

remove​(Object o): This method removes the first occurrence of the specified element from this list, if it is present.

removeFirst​(): This method removes and returns the first element from this list.

removeFirstOccurrence​(Object o): This method removes the first occurrence of the specified element in this list (when traversing the list from head to tail).

removeLast​(): This method removes and returns the last element from this list.

removeLastOccurrence​(Object o): This method removes the last occurrence of the specified element in this list (when traversing the list from head to tail).

set​(int index, E element): This method replaces the element at the specified position in this list with the specified element.

size​(): This method returns the number of elements in this list.

[**spliterator​():**](https://www.geeksforgeeks.org/linkedlist-spliterator-method-in-java/) This method Creates a late-binding and fail-fast Spliterator over the elements in this list.

toArray​(): This method returns an array containing all of the elements in this list in proper sequence (from first to last element).

toArray​(T[] a): This method returns an array containing all of the elements in this list in proper sequence (from first to last element);

the runtime type of the returned array is that of the specified array.

ArrayList vs LinkedList in Java:

ArrayList:-Implemented with the concept of dynamic array.

ArrayList<Type>arrL = new ArrayList<Type>();

Here Type is the data type of elements in ArrayList

to be created

 LinkedList:-Implemented with the concept of doubly linked list.

LinkedList<Type>linkL = new LinkedList<Type>();

Here Type is the data type of elements in LinkedList

to be created

Comparision between ArrayList and LinkedList:-

Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList in Java.

ArrayList also needs to be update its index if you insert something anywhere except at the end of array.

Removal also better in LinkedList than ArrayList due to same reasons as insertion.

LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

Both LinkedList and ArrayList require O(n) time to find if an element is present or not.

However we can do Binary Search on ArrayList if it is sorted and therefore can search in O(Log n) time.

// Java program to demonstrate difference between ArrayList and

// LinkedList.

import java.util.ArrayList;

import java.util.LinkedList;

public class ArrayListLinkedListExample

{

public static void main(String[] args)

{

ArrayList<String>arrlistobj = new ArrayList<String>();

arrlistobj.add("0. Practice.GeeksforGeeks.org");

arrlistobj.add("1. Quiz.GeeksforGeeks.org");

arrlistobj.add("2. Code.GeeksforGeeks.org");

arrlistobj.remove(1); // Remove value at index 2

System.out.println("ArrayList object output :" + arrlistobj);

// Checking if an element is present.

if (arrlistobj.contains("2. Code.GeeksforGeeks.org"))

System.out.println("Found");

else

System.out.println("Not found");

LinkedList llobj = new LinkedList();

llobj.add("0. Practice.GeeksforGeeks.org");

llobj.add("1. Quiz.GeeksforGeeks.org");

llobj.add("2. Code.GeeksforGeeks.org");

llobj.remove("1. Quiz.GeeksforGeeks.org");

System.out.println("LinkedList object output :" + llobj);

// Checking if an element is present.

if (llobj.contains("2. Code.GeeksforGeeks.org"))

System.out.println("Found");

else

System.out.println("Not found");

}

}

Output:

ArrayList object output :[0. Practice.GeeksforGeeks.org, 2. Code.GeeksforGeeks.org]

Found

LinkedList object output :[0. Practice.GeeksforGeeks.org, 2. Code.GeeksforGeeks.org]

Found

Vector (Class)

The Vector class implements a growable array of objects.

Vector implements a dynamic array that means it can grow or shrink as required.

Like an array, it contains components that can be accessed using an integer index.

They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.

It extends **AbstractList** and implements **List** interfaces.

Constructor:

Vector(): Creates a default vector of initial capacity is 10.

Vector(int size): Creates a vector whose initial capacity is specified by size.

Vector(int size, int incr): Creates a vector whose initial capacity is specified by size and increment is specified by incr.

It specifies the number of elements to allocate each time that a vector is resized upward.

Vector(Collection c): Creates a vector that contains the elements of collection c.

Important points regarding Increment of vector capacity:  
*If increment is specified, Vector will expand according to it in each allocation cycle but if increment is not specified then vector’s capacity get doubled in each allocation cycle.*

Vector defines three protected data member:

**int capacityIncreament:**Contains the increment value.

**int elementCount:**Number of elements currently in vector stored in it.

**Object elementData[]:**Array that holds the vector is stored in it.

**Methods in Vector:**

boolean add(Object obj): This method appends the specified element to the end of this vector.

**Syntax:** public boolean add(Object obj)

**Returns:** true if the specified element is added successfully into the Vector, otherwise it returns false.

**Exception:** NA.

// Java code illustrating add() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("geeks");

v.add("forGeeks");

v.add(3);

System.out.println("Vector is " + v);

}

}

Output:

[1, 2, geeks, forGeeks, 3]

void add(int index, Object obj): This method inserts the specified element at the specified position in this Vector.

**Syntax:** public void add(int index, Object obj)

**Returns:** NA.

**Exception:** IndexOutOfBoundsException, method throws this exception

if the index (obj position) we are trying to access is out of range

(index size()).

// Java code illustrating add() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector

Vector v = new Vector();

v.add(0, 1);

v.add(1, 2);

v.add(2, "geeks");

v.add(3, "forGeeks");

v.add(4, 3);

System.out.println("Vector is " + v);

}

}

Output:

Vector is: [1, 2, geeks, forGeeks, 3]

booleanaddAll(Collection c) This method appends all of the elements  
in the specified Collection to the end of this Vector.

**Syntax:** public booleanaddAll(Collection c)

**Returns:** Returns true if operation succeeded otherwise false.

**Exception:** NullPointerException thrown if collection is null

// Java code illustrating addAll()

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

ArrayList arr = new ArrayList();

arr.add(3);

arr.add("geeks");

arr.add("forgeeks");

arr.add(4);

// createn default vector

Vector v = new Vector();

// copying all element of array list int0 vector

v.addAll(arr);

// checking vector v

System.out.println("vector v:" + v);

}

}

Output:

vector v:[3, geeks, forgeeks, 4]

booleanaddAll(int index, Collection c) This method inserts all of the elements in the specified Collection into this Vector at the specified.  
position.

**Syntax:** public booleanaddAll(int index, Collection c)

**Returns:** true if this list changed as a result of the call.

**Exception:** IndexOutOfBoundsException -- If the index is out of range,

NullPointerException -- If the specified collection is null.

// Java code illustrating addAll()

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

ArrayList arr = new ArrayList();

arr.add(3);

arr.add("geeks");

arr.add("forgeeks");

arr.add(4);

// createn default vector

Vector v = new Vector();

v.add(2);

// copying all element of array list int0 vector

v.addAll(1, arr);

// checking vector v

System.out.println("vector v:" + v);

}

}

Output:

vector v:[2, 3, geeks, forgeeks, 4]

void clear() This method removes all of the elements from this vector.

**Syntax:** public void clear()

**Returns:** NA.

**Exception:** NA.

// Java code illustrating clear() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector

Vector v = new Vector();

v.add(0, 1);

v.add(1, 2);

v.add(2, "geeks");

v.add(3, "forGeeks");

v.add(4, 3);

System.out.println("Vector is: " + v);

// clearing the vector

v.clear();

// checking vector

System.out.println("after clearing: " + v);

}

}

Output:

Vector is: [1, 2, geeks, forGeeks, 3]

after clearing: []

Object clone() This method returns a clone of this vector.

**Syntax:** public Object clone()

**Returns:** a clone of this ArrayList instance.

**Exception:** NA.

// Java code illustrating clone()

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector

Vector v = new Vector();

Vector v\_clone = new Vector();

v.add(0, 1);

v.add(1, 2);

v.add(2, "geeks");

v.add(3, "forGeeks");

v.add(4, 3);

v\_clone = (Vector)v.clone();

// checking vector

System.out.println("Clone of v: " + v\_clone);

}

}

Output:

Clone of v: [1, 2, geeks, forGeeks, 3]

boolean contains(Object o): This method returns true if this vector contains the specified element.

**Syntax:** public boolean contains(object o)

**Returns:** true if the operation is succeeded otherwise false.

**Exception:** NA.

// Java code illustrating contains() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("geeks");

v.add("forGeeks");

v.add(3);

// check whether vector contains "forGeeks"

if (v.contains("forGeeks"))

System.out.println("forGeeks exists");

}

}

Output:

forGeeks exists

void ensureCapacity(int minCapacity): This method increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument .

Syntax: public void ensureCapacity(int minCapacity)

Returns: NA.

Exception: NA.

// Java code illustrating ensureCapacity() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector of capacity 10

Vector v = new Vector();

// ensuring capacity

v.ensureCapacity(22);

// cheking capacity

System.out.println("Minimum capacity: " + v.capacity());

}

}

Output:

Minimum capacity: 22

Object get(int index):This method returns the element at the specified position in this Vector.

**Syntax:** public Object get(int index)

**Returns:** returns the element at specified positions .

**Exception:** IndexOutOfBoundsException -- if the index is out of range.

// Java code illustrating get() methods

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector of capacity 10

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("Geeks");

v.add("forGeeks");

v.add(4);

// get the element at index 2

System.out.println("element at indexx 2 is: " + v.get(2));

}

}

Output:

element at indexx 2 is: Geeks

int indexOf(Object o): This method returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

**Syntax:** public int indexOf(Object o)

**Returns:** the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

**Exception:** NA.

// Java code illustrating indexOf() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector of capacity 10

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("Geeks");

v.add("forGeeks");

v.add(4);

// get the element at index of Geeks

System.out.println("index of Geeks is: " + v.indexOf("Geeks"));

}

}

Output:

index of Geeks is: 2

boolean isEmpty()**:**This method tests if this vector has no components.

**Syntax:** public booleanisEmpty()

**Returns:** true if vector is empty otherwise false.

**Exception:** NA.

// Java code illustrating isEmpty() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector of capacity 10

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("Geeks");

v.add("forGeeks");

v.add(4);

v.clear();

// check whether vector is empty or not

if (v.isEmpty())

System.out.println("Vector is clear");

}

}

Output:

Vector is clear

int lastIndexOf(Object o): This method returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

**Syntax:** public int lastIndexOf(Object o)

**Returns:** returns the index of the last occurrence of the

specified element in this list, or -1 if this list does not contain the element.

**Exception:** NA.

boolean remove(Object o)**:**This method removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.

**Syntax:** public boolean remove(Object o)

**Returns:** Returns the first occurrence of element.

**Exception:** NA.

boolean equals(Object o)**:**This method compares the specified Object with this Vector for equality.

**Syntax:** public boolean equal(Object o)

**Returns:** true if operation succeeded otherwise false.

**Exception:** NA.

// Java code illustrating equals() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector of capacity 10

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("Geeks");

v.add("forGeeks");

v.add(4);

// second vector

Vector v\_2nd = new Vector();

v\_2nd.add(1);

v\_2nd.add(2);

v\_2nd.add("Geeks");

v\_2nd.add("forGeeks");

v\_2nd.add(4);

if (v.equals(v\_2nd))

System.out.println("both vectors are equal");

}

}

Output:

both vectors are equal

Object firstElement()**:** This method returns the first component (the item at index 0) of this vector.

**Syntax:** public Object firstElement()

**Returns:** NA.

**Exception:** NoSuchElementException -- This exception is returned

if this vector has no components.

void trimToSize(): This method trims the capacity of this vector to be the vector’s current size.

Syntax: public void trimToSize()

Returns: NA.

Exception: NA.

// Java code illustrating trimToSize() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector of capacity 10

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("Geeks");

v.add("forGeeks");

v.add(4);

// checking initial capacity

System.out.println("Initial capacity: " + v.capacity());

// trim capacity to size

v.trimToSize();

// checking capacity after triming

System.out.println("capacity after triming: " + v.capacity());

}

}

Output:

Initial capacity: 10

capacity after triming: 5

String toString(): The toString() method is used to return a string representation of this Vector, containing the String representation of each element.

Syntax: public String toString()

Returns: a string representation of this collection.

Exception: NA

object[ ] toArray()**:***This method returns a array representation of this Vector, containing the String representation of each element.*

**Syntax:** public object[ ] toArray()

**Returns:** an array containing all of the elements in this collection.

**Exception:** NA.

int size(): This method returns the number of components in this vector.

**Syntax:** public int size()

**Returns:** returns the number of components in this vector.

**Exception:** NA

void setSize(int newSize): This method sets the size.

Syntax: public void setSize(int newSize)

Returns: NA.

Exception: ArrayIndexOutOfBoundsException -- This exception is thrown

if the new size is negative.

void setElementAt(Object obj, int index): This method sets the component at the specified index of this vector to be the specified object.

Syntax: public void setElementAt(E obj, int index)

Returns: NA.

Exception: ArrayIndexOutOfBoundsException -- This exception is thrown

if the accessed index is out of range.

// Java code illustrating setElementAt() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

// create default vector of capacity 10

Vector v = new Vector();

v.add(1);

v.add(2);

v.add("Geeks");

v.add("forGeeks");

v.add(4);

// set 4 at the place of 2

v.setElementAt(4, 1);

System.out.println("vector: " + v);

}

}

Output:

vector: [1, 4, Geeks, forGeeks, 4]

retainAll(Collection c): This method retains only the elements in this Vector that are contained in the specified Collection.

Syntax: public booleanretainAll(Collection c)

Returns: true if this Vector is changed as a result of the call.

Exception: NullPointerException -- This exception is thrown if the

specified collection is null.

// Java code illustrating retainAll() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

Vector vec = new Vector(7);

Vector vecretain = new Vector(4);

// use add() method to add elements in the vector

vec.add(1);

vec.add(2);

vec.add(3);

vec.add(4);

vec.add(5);

vec.add(6);

vec.add(7);

// this elements will be retained

vecretain.add(5);

vecretain.add(3);

vecretain.add(2);

System.out.println("Calling retainAll()");

vec.retainAll(vecretain);

// let us print all the elements available in vector

System.out.println("Numbers after removal :- ");

Iterator itr = vec.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

}

}

Output:

Calling retainAll()

Numbers after removal :-

2

3

5

void removeAllElements(): This method removes all components from this vector and sets its size to zero.

Syntax: public void removeAllElements()

Returns: NA.

Exception: NA.

Object lastElement()**:**This method returns the last component of the vector.

**Syntax:** public Object lastElement()

**Returns:** returns the last component of the vector,

i.e., the component at index size() - 1.

**Exception:** NoSuchElementException -- This exception is thrown

if this vector is empty

int hashCode(): This method returns the hash code value for this Vector.

**Syntax:** public int hashCode()

**Returns:** returns the hash code value(int) for this list.

**Exception:** NA.

booleanremoveElement(Object obj): This method removes the first occurrence of the argument from this vector.

Syntax: public booleanremoveElement(Object obj)

Returns: true if operation is succeeded otherwise false.

Exception:

void copyInto(Object[ ] anArray)**:**This method copies the components of this vector into the specified array.

**Syntax:** public void copyInto(Object[] anArray)

**Returns:** NA.

**Exception:** NullPointerException -- if the given array is null.

// Java code illustrating copyInto() method

import java.util.\*;

class Vector\_demo {

public static void main(String[] arg)

{

Vector vec = new Vector(7);

// use add() method to add elements in the vector

vec.add(1);

vec.add(2);

vec.add(3);

vec.add(4);

vec.add(5);

vec.add(6);

vec.add(7);

Integer[] arr = new Integer[7];

// copy componnent of vector int array arr

vec.copyInto(arr);

System.out.println("elements in array arr: ");

for (Integer num :arr) {

System.out.println(num);

}

}

}

int capacity()**:**This method returns the current capacity of this vector.

**Syntax:** public int capacity()

**returns:**  returns the current capacity of the

vector as an integer value. Here capacity means the length of

its internal data array, kept in the field elementData of this vector.

**Exception:** NA.

void insertElementAt(Object obj, int index)**:** This method inserts the specified object as a component in this vector at the specified index.

**Syntax:** public void insertElementAt(E obj, int index)

**Returns:** NA.

**Exception:** ArrayIndexOutOfBoundsException -- This exception is thrown

if the index is invalid.

Stack (Class):

Stack models and implements Stack data structure.

The class is based on the basic principle of last-in-first-out. [LIFO]

In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek.

The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions.

The class can also be referred to as the subclass of Vector.

hierarchy of Stack class

A screenshot of a cell phone

Description automatically generated

The class supports one **default constructor** **Stack()** which is used to **create an empty stack**.

// Java code for stack implementation

import java.io.\*;

import java.util.\*;

class Test

{

// Pushing element on the top of the stack

static void stack\_push(Stack<Integer> stack)

{

for(int i = 0; i< 5; i++)

{

stack.push(i);

}

}

// Popping element from the top of the stack

static void stack\_pop(Stack<Integer> stack)

{

System.out.println("Pop :");

for(int i = 0; i< 5; i++)

{

Integer y = (Integer) stack.pop();

System.out.println(y);

}

}

// Displaying element on the top of the stack

static void stack\_peek(Stack<Integer> stack)

{

Integer element = (Integer) stack.peek();

System.out.println("Element on stack top : " + element);

}

// Searching element in the stack

static void stack\_search(Stack<Integer> stack, int element)

{

Integer pos = (Integer) stack.search(element);

if(pos == -1)

System.out.println("Element not found");

else

System.out.println("Element is found at position " + pos);

}

public static void main (String[] args)

{

Stack<Integer> stack = new Stack<Integer>();

stack\_push(stack);

stack\_pop(stack);

stack\_push(stack);

stack\_peek(stack);

stack\_search(stack, 2);

stack\_search(stack, 6);

}

}

Output:

Pop :

4

3

2

1

0

Element on stack top : 4

Element is found at position 3

Element not found

**Methods in Stack class:**

Object push(**Object element**) : Pushes an element on the top of the stack.

Object pop() : Removes and returns the top element of the stack.

An ‘EmptyStackException’ exception is thrown if we call pop() when the invoking stack is empty.

Object peek() : Returns the element on the top of the stack, but does not remove it.

boolean empty() : It returns true if nothing is on the top of the stack. Else, returns false.

int search(**Object element**) : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

Set (Interface) :

Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.

Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).

Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

// Java code for adding elements in Set

import java.util.\*;

public class Set\_example

{

public static void main(String[] args)

{

// Set deonstration using HashSet

Set<String> hash\_Set = new HashSet<String>();

hash\_Set.add("Geeks");

hash\_Set.add("For");

hash\_Set.add("Geeks");

hash\_Set.add("Example");

hash\_Set.add("Set");

System.out.print("Set output without the duplicates");

System.out.println(hash\_Set);

// Set deonstration using TreeSet

System.out.print("Sorted Set after passing into TreeSet");

Set<String> tree\_Set = new TreeSet<String>(hash\_Set);

System.out.println(tree\_Set);

}

}

Output:

Set output without the duplicates[Set, Example, Geeks, for]

Sorted Set after passing into TreeSet[Example, For, Geeks, Set]

Note: As we can see the duplicate entry “Geeks” is ignored in the final output, Set interface doesn’t allow duplicate entries.

Now we will see some of the basic operations on the Set i.e. Union, Intersection and Difference.

Let’s take an example of two integer Sets:

[1, 3, 2, 4, 8, 9, 0]

[1, 3, 7, 5, 4, 0, 7, 5]

Union  
In this, we could simply add one Set with other. Since the Set will itself not allow any duplicate entries, we need not take care of the common values.

Expected Output:

Union : [0, 1, 2, 3, 4, 5, 7, 8, 9]

Intersection  
We just need to retain the common values from both Sets.

Expected Output:

Intersection : [0, 1, 3, 4]

Difference  
We just need to remove all the values of one Set from the other.  
  
Expected Output:

Difference : [2, 8, 9]

// Java code for demonstrating union, intersection and difference

// on Set

import java.util.\*;

public class Set\_example

{

public static void main(String args[])

{

Set<Integer> a = new HashSet<Integer>();

a.addAll(Arrays.asList(new Integer[] {1, 3, 2, 4, 8, 9, 0}));

Set<Integer> b = new HashSet<Integer>();

b.addAll(Arrays.asList(new Integer[] {1, 3, 7, 5, 4, 0, 7, 5}));

// To find union

Set<Integer> union = new HashSet<Integer>(a);

union.addAll(b);

System.out.print("Union of the two Set");

System.out.println(union);

// To find intersection

Set<Integer> intersection = new HashSet<Integer>(a);

intersection.retainAll(b);

System.out.print("Intersection of the two Set");

System.out.println(intersection);

// To find the symmetric difference

Set<Integer> difference = new HashSet<Integer>(a);

difference.removeAll(b);

System.out.print("Difference of the two Set");

System.out.println(difference);

}

}

Output:

Union of the two Set[0, 1, 2, 3, 4, 5, 7, 8, 9]

Intersection of the two Set[0, 1, 3, 4]

Difference of the two Set[2, 8, 9]

Map:

A screenshot of a cell phone

Description automatically generated

Java HashMap is a hash table based implementation of Java's Map interface.

A Map is a collection of key-value pairs.

Java HashMap allows null values and the null key.

HashMap is an unordered collection. It does not guarantee any specific order of the elements.

Maps are used for when you want to associate a key with a value and Lists are an ordered collection.

Map is an interface in the Java Collection Framework and a HashMap is one implementation of the Map interface. HashMap are efficient for locating a value based on a key and inserting and deleting values based on a key.

It's part of the collection framework but it doesn't implement the java.util.Collection interface.

It's a different branch of the hierarchy.

If you want, you can view it on the same level of the hierarchy as the Collection interface. ... Because Map is three collections:

Keys, values and key-value pairs.

Each key in a HashMap must be unique. When "adding a duplicate key" the old value (for the same key, as keys must be unique) is simply replaced;

Any non-nullobject can be used as a key or as a value. ... We cannot have more than one Null key in HashMap because Keys are unique therefor only one Null key and manyNull values are allowed.

*Few characteristics of the Map Interface are:*

A Map can not contain duplicate keys and each key can map to at most one value.

Some implementations allow null key and null value like the HashMap and LinkedHashMap, but some do not like the TreeMap.

The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.

There are two interfaces for implementing Map in java:

Map and SortedMap, and

three classes: HashMap, TreeMap and LinkedHashMap.

Why and When to use Maps?

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some examples are:

A map of error codes and their descriptions.

A map of zip codes and cities.

A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.

A map of classes and students. Each class (key) is associated with a list of students (value).

Methods in Map Interface:

public Object put(Object key, Object value): This method is used to insert an entry in this map.

public void putAll(Map map): This method is used to insert the specified map in this map.

public Object remove(Object key): This method is used to delete an entry for the specified key.

public Object get(Object key):This method is used to return the value for the specified key.

public boolean containsKey(Object key): This method is used to search the specified key from this map.

public Set keySet(): This method is used to return the Set view containing all the keys.

public Set entrySet(): This method is used to return the Set view containing all the keys and values.

// Java program to demonstrate working of Map interface

importjava.util.\*;

classHashMapDemo

{

   publicstaticvoidmain(String args[])

   {

       Map<String,Integer> hm =

                        new HashMap<String,Integer>();

       hm.put("a", new Integer(100));

       hm.put("b", new Integer(200));

       hm.put("c", new Integer(300));

       hm.put("d", new Integer(400));

       // Returns Set view

       Set<Map.Entry<String,Integer>> st = hm.entrySet();

       for(Map.Entry<String,Integer> me : st)

       {

           System.out.print(me.getKey()+":");

           System.out.println(me.getValue());

       }

   }

}

Output:

a:100

b:200

c:300

d:400

|  |  |
| --- | --- |
| Class | Description |
| HashMap | HashMap is the implementation of Map, but it doesn't maintain any order. |
| LinkedHashMap | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| TreeMap | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

A Map doesn't allow duplicate keys, but you can have duplicate values.

HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

Useful methods of Map interface:

|  |  |
| --- | --- |
| Method | Description |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| Set keySet() | It returns the Set view containing all the keys. |
| Set<Map.Entry<K,V>>entrySet() | It returns the Set view containing all the keys and values. |
| void clear() | It is used to reset the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V>remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V>mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V>remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| booleancontainsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| booleancontainsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| int hashCode() | It returns the hash code value for the Map |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V>remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

Map.Entry Interface:

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

Methods of Map.Entry interface:

|  |  |
| --- | --- |
| Method | Description |
| K getKey() | It is used to obtain a key. |
| V getValue() | It is used to obtain value. |
| int hashCode() | It is used to obtain hashCode. |
| V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with the other existing objects. |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>>comparingByKey() | It returns a comparator that compare the objects in natural order on key. |
| static <K,V> Comparator<Map.Entry<K,V>>comparingByKey(Comparator<? super K>cmp) | It returns a comparator that compare the objects by key using the given Comparator. |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>>comparingByValue() | It returns a comparator that compare the objects in natural order on value. |
| static <K,V> Comparator<Map.Entry<K,V>>comparingByValue(Comparator<? super V>cmp) | It returns a comparator that compare the objects by value using the given Comparator. |

Java Map Example: Non-Generic (Old Style)

//Non-generic

**import** java.util.\*;

**public** **class** MapExample1 {

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

    Map map=**new** HashMap();

    //Adding elements to map

    map.put(1,"Amit");

    map.put(5,"Rahul");

    map.put(2,"Jai");

    map.put(6,"Amit");

    //Traversing Map

    Set set=map.entrySet();//Converting to Set so that we can traverse

    Iterator itr=set.iterator();

**while**(itr.hasNext()){

        //Converting to Map.Entry so that we can get key and value separately

        Map.Entry entry=(Map.Entry) itr.next();

        System.out.println(entry.getKey()+" "+entry.getValue());

    }

}

}

Output:

1 Amit

2 Jai

5 Rahul

6 Amit

Java Map Example: Generic (New Style)

**import** java.util.\*;

**class** MapExample2{

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

  Map<Integer,String> map=**new** HashMap<Integer,String>();

  map.put(100,"Amit");

  map.put(101,"Vijay");

  map.put(102,"Rahul");

  //Elements can traverse in any order

**for**(Map.Entry m:map.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

}

Output:

102 Rahul

100 Amit

101 Vijay

Java Map Example: comparingByKey()

import java.util.\*;

class MapExample3{

 public static void main(String args[]){

Map<Integer,String> map=new HashMap<Integer,String>();

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

      //Returns a Set view of the mappings contained in this map

      map.entrySet()

      //Returns a sequential Stream with this collection as its source

      .stream()

      //Sorted according to the provided Comparator

      .sorted(Map.Entry.comparingByKey())

      //Performs an action for each element of this stream

      .forEach(System.out::println);

 }

}

Output:

100=Amit

101=Vijay

102=Rahul

Java Map Example: comparingByKey() in Descending Order

import java.util.\*;

class MapExample4{

 public static void main(String args[]){

Map<Integer,String> map=new HashMap<Integer,String>();

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

      //Returns a Set view of the mappings contained in this map

      map.entrySet()

      //Returns a sequential Stream with this collection as its source

      .stream()

      //Sorted according to the provided Comparator

      .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))

      //Performs an action for each element of this stream

      .forEach(System.out::println);

 }

}

Output:

102=Rahul

101=Vijay

100=Amit

Java Map Example: comparingByValue()

import java.util.\*;

class MapExample5{

 public static void main(String args[]){

Map<Integer,String> map=new HashMap<Integer,String>();

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

      //Returns a Set view of the mappings contained in this map

      map.entrySet()

      //Returns a sequential Stream with this collection as its source

      .stream()

      //Sorted according to the provided Comparator

      .sorted(Map.Entry.comparingByValue())

      //Performs an action for each element of this stream

      .forEach(System.out::println);

 }

}

Output:

100=Amit

102=Rahul

101=Vijay

Java Map Example: comparingByValue() in Descending Order

import java.util.\*;

class MapExample6{

 public static void main(String args[]){

Map<Integer,String> map=new HashMap<Integer,String>();

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

     //Returns a Set view of the mappings contained in this map

     map.entrySet()

     //Returns a sequential Stream with this collection as its source

     .stream()

     //Sorted according to the provided Comparator

     .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))

     //Performs an action for each element of this stream

     .forEach(System.out::println);

 }

}

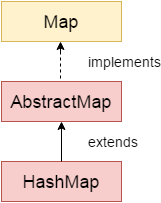
Output:

101=Vijay

102=Rahul

100=Amit

Java HashMap class:



Java HashMap class implements the map interface by using a hash table. It inherits AbstractMap class and implements Map interface.

Points to remember:

Java HashMap class contains values based on the key.

Java HashMap class contains only unique keys.

Java HashMap class may have one null key and multiple null values.

Java HashMap class is non synchronized.

Java HashMap class maintains no order.

The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

HashMap class declaration

**public** **class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

HashMap class Parameters

K: It is the type of keys maintained by this map.

V: It is the type of mapped values.

Constructors of Java HashMap class:

|  |  |
| --- | --- |
| Constructor | Description |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map<? extends K,? extends V> m) | It is used to initialize the hash map by using the elements of the given Map object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value, capacity. |
| HashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and load factor of the hash map by using its arguments. |

Methods of Java HashMap class:

|  |  |
| --- | --- |
| Method | Description |
| void clear() | It is used to remove all of the mappings from this map. |
| booleanisEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V>remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V>mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V>remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| booleancontainsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| booleancontainsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValueif the map contains no mapping for the key. |
| booleanisEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V>remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

Java HashMap example to add() elements

import java.util.\*;

class HashMap1{

 public static void main(String args[]){

   HashMap<Integer,String> hm=new HashMap<Integer,String>();

    System.out.println("Initial list of elements: "+hm);

      hm.put(100,"Amit");

      hm.put(101,"Vijay");

      hm.put(102,"Rahul");

      System.out.println("After invoking put() method ");

      for(Map.Entry m:hm.entrySet()){

       System.out.println(m.getKey()+" "+m.getValue());

      }

      hm.putIfAbsent(103, "Gaurav");

      System.out.println("After invoking putIfAbsent() method ");

      for(Map.Entry m:hm.entrySet()){

           System.out.println(m.getKey()+" "+m.getValue());

          }

      HashMap<Integer,String> map=new HashMap<Integer,String>();

      map.put(104,"Ravi");

      map.putAll(hm);

      System.out.println("After invoking putAll() method ");

      for(Map.Entry m:map.entrySet()){

           System.out.println(m.getKey()+" "+m.getValue());

          }

 }

}

Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

104 Ravi

Java HashMap example to remove() elements

import java.util.\*;

public class HashMap2 {

   public static void main(String args[]) {

    HashMap<Integer,String> map=new HashMap<Integer,String>();

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

      map.put(103, "Gaurav");

    System.out.println("Initial list of elements: "+map);

    //key-based removal

    map.remove(100);

    System.out.println("Updated list of elements: "+map);

    //value-based removal

    map.remove(101);

    System.out.println("Updated list of elements: "+map);

    //key-value pair based removal

    map.remove(102, "Rahul");

    System.out.println("Updated list of elements: "+map);

   }

}

Output:

Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {102=Rahul, 103=Gaurav}

Updated list of elements: {103=Gaurav}

Java HashMap example to replace() elements

import java.util.\*;

class HashMap3{

 public static void main(String args[]){

   HashMap<Integer,String> hm=new HashMap<Integer,String>();

      hm.put(100,"Amit");

      hm.put(101,"Vijay");

      hm.put(102,"Rahul");

      System.out.println("Initial list of elements:");

     for(Map.Entry m:hm.entrySet())

     {

        System.out.println(m.getKey()+" "+m.getValue());

     }

     System.out.println("Updated list of elements:");

     hm.replace(102, "Gaurav");

     for(Map.Entry m:hm.entrySet())

     {

        System.out.println(m.getKey()+" "+m.getValue());

     }

     System.out.println("Updated list of elements:");

     hm.replace(101, "Vijay", "Ravi");

     for(Map.Entry m:hm.entrySet())

     {

        System.out.println(m.getKey()+" "+m.getValue());

     }

     System.out.println("Updated list of elements:");

     hm.replaceAll((k,v) -> "Ajay");

     for(Map.Entry m:hm.entrySet())

     {

        System.out.println(m.getKey()+" "+m.getValue());

     }

 }

}

Initial list of elements:

100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay

Difference between HashSet and HashMap:

*HashSet contains only values whereas HashMap contains an entry(key and value).*

Java HashMap Example: Book

import java.util.\*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;

    this.quantity = quantity;

}

}

public class MapExample {

public static void main(String[] args) {

    //Creating map of Books

    Map<Integer,Book> map=new HashMap<Integer,Book>();

    //Creating Books

    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

    //Adding Books to map

    map.put(1,b1);

    map.put(2,b2);

    map.put(3,b3);

    //Traversing map

    for(Map.Entry<Integer, Book> entry:map.entrySet()){

        int key=entry.getKey();

        Book b=entry.getValue();

        System.out.println(key+" Details:");

        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

    }

}

}

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

Java LinkedHashMap class:

Java TreeMap class:

Java Hashtable class:

Difference between HashMap and Hashtable:

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| HashMap | Hashtable |
| 1) HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is synchronized. It is thread-safe and can be shared with many threads. |
| 2) HashMap allows one null key and multiple null values. | Hashtable doesn't allow any null key or value. |
| 3) HashMap is a new class introduced in JDK 1.2. | Hashtable is a legacy class. |
| 4) HashMap is fast. | Hashtable is slow. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is traversed by Iterator. | Hashtable is traversed by Enumerator and Iterator. |
| 7) Iterator in HashMap is fail-fast. | Enumerator in Hashtable is not fail-fast. |
| 8) HashMap inherits AbstractMap class. | Hashtable inherits Dictionary class. |

Que:

The Queue interface is available in java.util package and extends the Collection interface.

The queue collection is used to hold the elements about to be processed and provides various operations like the insertion, removal etc.

It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list i.e. it follows the FIFO or the First-In-First-Out principle.

Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.

It is to be noted that both the implementations are not thread safe.

**PriorityBlockingQueue** is one alternative implementation if thread safe implementation is needed.

Few important characteristics of Queue are:

The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.

The Java Queue supports all methods of Collection interface including insertion, deletion etc.

LinkedList, ArrayBlockingQueue and PriorityQueue are the most frequently used implementations.

If any null operation is performed on BlockingQueues, NullPointerException is thrown.

BlockingQueues have thread-safe implementations.

The Queues which are available in java.util package are Unbounded Queues

The Queues which are available in java.util.concurrent package are the Bounded Queues.

All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively.

The Deques support element insertion and removal at both ends.

Methods in Queue:

add()- This method is used to add elements at the tail of queue.

More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation.

peek()- This method is used to view the head of queue without removing it. It returns Null if the queue is empty.

element()- This method is similar to peek().

It throws *NoSuchElementException* when the queue is empty.

remove()- This method removes and returns the head of the queue.

It throws *NoSuchElementException* when the queue is empty.

poll()- This method removes and returns the head of the queue. It returns null if the queue is empty.

size()- This method return the no. of elements in the queue.

A green and red text

Description automatically generated

// Java program to demonstrate working of Queue interface in Java

import java.util.LinkedList;

import java.util.Queue;

public class QueueExample

{

public static void main(String[] args)

{

Queue<Integer> q = new LinkedList<>();

// Adds elements {0, 1, 2, 3, 4} to queue

for (int i=0; i<5; i++)

q.add(i);

// Display contents of the queue.

System.out.println("Elements of queue-"+q);

// To remove the head of queue.

int removedele = q.remove();

System.out.println("removed element-" + removedele);

System.out.println(q);

// To view the head of queue

int head = q.peek();

System.out.println("head of queue-" + head);

// Rest all methods of collection interface,

// Like size and contains can be used with this

// implementation.

int size = q.size();

System.out.println("Size of queue-" + size);

}

}

Regex in Java – Regular expressions.

*A regular expression is a sequence of characters that constructs a search pattern.*

Java Regex is an API which is used to define a pattern for searching and manipulating Strings.

It is widely used to define the constraint on strings such as password and email validation.

Java Regex API provides 1 interface and 3 classes in java.util.regex package.

MatchResult interface

Matcher class

Pattern class

PatternSyntaxException class

The Matcher and Pattern classes provide the facility of Java regular expression.

Multi Threading:

Thread- a unit of a Process.

Threads – for maximum utilisation of CPU power.

main() is a thread.

Thread is a class in Java:

Thread obj = new Thread();

Multi Threading applications:

In Gaming, in async requests, accessing app from multiple clients simultaneously.

Ways to create a thread:

Create a class and inherit Thread class (Class Multi extends Thread).

Using Runnable Interface which has a run() method.

sleep() method for suspending thread for some specified time.

start() is a method in Thread class which internally calls run() method and it is used to run threads ( methods )parallelly.

Note:

scheduler in OS runs the threads based on some criteria like thread priority, time takn by each thread for execution.

Runnable:

It is a functional interface has only one method i.e. run().

To call the run() method of the runnable class (class which implements Runnable) pass the object to the Thread class.

Thread t1= new Thread(obj);

Multi Threading using lambda expression:

join() and isAlive() methods:

isAlive() – detects thread is in Running state or not.

join() to join threads with main thread.

Note:

setName() – to set the name for the thread.

getName() – method used to get the name of the thread.

default name is : Thread – 0 for first thread and 1,2, and do on.

For second,third and so on.

We can give the name for the thread in the Thread constructor

Thread(Runnable object,”Thread name”)

Thread priority:

Range of priority: 1 to 10.

Least / Thread.MIN\_PRIORITY

5- Normal / Thread.NORM\_PRIORITY

10 – highest / Thread.MAX\_PRIORITY

setPriority()

getPrioroty()

Note:

currentThread() *will identify the current thread.*

Synchronized :

Used only with method to make only one thread can use it at a time.

*Memory management in Java:*

memory management is the *process of allocation and de-allocation of objects.*

Java does memory management automatically.

Java uses an automatic memory management system called a garbage collector.

A screenshot of a cell phone

Description automatically generated

JVM memory is divided into separate parts.

At broad level, JVM Heap memory is physically divided into two parts – **Young Generation** and **Old Generation**.

**Young Generation:**

young generation is the place where all the new objects are created.

*When the young generation is filled, garbage collection is performed.*

This garbage collection is called **Minor GC**.

Young Generation is divided into three parts –

**Eden Memory** and two **Survivor Memory** spaces.

Most of the newly created objects are located in the Eden memory space.

*When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces*.

Minor GC also checks the survivor objects and move them to the other survivor space. *So at a time, one of the survivor space is always empty.*

*Objects that are survived after many cycles of GC, are moved to the Old generation memory space.*

Usually, it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

Old Generation:

Old Generation memory contains the objects that are long-lived and survived after many rounds of Minor GC. Usually, garbage collection is performed in Old Generation memory when it’s full. Old Generation Garbage Collection is called **Major GC** and usually takes a longer time.

Permanent Generation:

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application.

Note that Perm Gen is not part of Java Heap memory.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

Out of order execution

*Performance driven changes done by JVM/Compiler/CPU.*

Field visibility

*Generally applicable for Multithreading / Concurrency.*

Field visibility issue can be overcome by volatile.

*Using volatile ensures the value of x from a cache of a core is flushed to the shared cache (RAM) and can be loaded to cache of another core which wants to access the modified value of a member.*

*JMM [Java Memory Model] is a specification which ensures visibility of fields / variables amidst reordering of instructions.*

*“ Happens – before “ relationship for volatile*

*Summary of Java Memory Management:*

Java Memory:

2 sections 🡪 1. Stack 2. Heap

Stack:

A FILO Data structure managed by JVM.

*When variables are created inside a method they are pushed to the stack and when method is been called they are popped out of stack in FILO order.*

All the local variables are created in Stack and they are destroyed once we reach the closing curly bracket of the Class.

Stack is for primitive data types.

Heap:

Memory location for data for longer time. Ex: objects, secondary data types.

Java Memory rules:

Objects are stored on the heap.

Variables are reference to the object.

Local variables are stored on the stack.

Sharing data in Java Memory:

Values and References:

*Passing values of local variables in Java is always by copy of the value of that variable to any method. [Passing variables by Value].*

*Passing variables by reference is not possible in Java.*

*For objects passed in to methods, the reference to the object is passed by value.*

*final keyword:*

*Escaping references:*

*Spotting escaping references and handling them.*

*Encapsulation: Data + functionalities together.*

*References can be escaped if we have methods returning a pointer to the object (variable).*

*All collection in Java will accept the existing collection.*

*Immutable Collections are the solutions for Escaping references* for Collections*.*

*[*Collections.unmodifiableMap(Collection col) ]

Garbage Collection:

Comparable vs Comparator

Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered.

For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.

Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.

If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using

Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo() method.

Collection vs Collections

Iterator

Spliterator

Hashing technique

Synchronous and Asynchronous

fail fast

lambda expression

Code complexity

Big O notation

Arrays class

Character class

lambda expression

Serialization and Deserialisation

Multithreading

waits in Java

System class

Math class in Java

Important inbuilt Classes in Java:

Character:

Java provides a wrapper class **Character** in java.lang package. An object of type Character contains a single field, whose type is char.

**Creating a Character object :**

Character ch = new Character('a');

The above statement creates a Character object which contain ‘a’ of type char. There is only one constructor in Character class which expect an argument of char data type.

If we pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character class object. This feature is called Autoboxing and Unboxing.

**Note :** The Character class is immutable like String class i.e once it’s object is created, it **cannot** be changed.

**Methods in Character Class :**

**boolean isLetter(char ch)** : This method is used to determine whether the specified char value(ch) is a letter or not. The method will return true if it is letter([A-Z],[a-z]), otherwise return false. In place of character, we can also pass ASCII value as an argument as char to int is implicitly typecasted in java.

**Syntax :**

boolean isLetter(char ch)

**Parameters :**

ch - a primitive character

**Returns :**

returns true if ch is a alphabet, otherwise return false

**boolean isDigit(char ch)** : This method is used to determine whether the specified char value(ch) is a digit or not. Here also we can pass ASCII value as an argument.

**Syntax :**

boolean isDigit(char ch)

**Parameters :**

ch - a primitive character

**Returns :**

returns true if ch is a digit, otherwise return false

boolean isWhitespace(char ch**)** : It determines whether the specified char value(ch) is white space. A whitespace includes space, tab, or new line.

**Syntax :**

boolean isWhitespace(char ch)

**Parameters :**

ch - a primitive character

**Returns :**

returns true if ch is a whitespace.

otherwise return false

**boolean isUpperCase(char ch)** : It determines whether the specified char value(ch) is uppercase or not.

**Syntax :**

boolean isUpperCase(char ch)

**boolean isLowerCase(char ch)** : It determines whether the specified char value(ch) is lowercase or not.

**Syntax :**

boolean isLowerCase(char ch)

**char toUpperCase(char ch)** : It returns the uppercase of the specified char value(ch). If an ASCII value is passed, then the ASCII value of it’s uppercase will be returned.

**Syntax :**

char toUpperCase(char ch)

**Parameters :**

ch - a primitive character

**Returns :**

returns the uppercase form of the specified char value.

**char toLowerCase(char ch)** : It returns the lowercase of the specified char value(ch).

**Syntax :**

char toLowerCase(char ch)

**Parameters :**

ch - a primitive character

**Returns :**

returns the lowercase form of the specified char value.

**toString(char ch)** : It returns a String class object representing the specified character value(ch) i.e a one-character string. Here we **cannot** pass ASCII value.

**Syntax :**

String toString(char ch)

**Parameters :**

ch - a primitive character

**Returns :**

returns a String object.

Integer

Integer class is a wrapper class for the primitive type int which contains several methods to effectively deal with a int value like converting it to a string representation, and vice-versa. An object of Integer class can hold a single int value.  
**Constructors:**

**Integer(int b):**Creates a Integer object initialized with the value provided.

**Syntax :** public Integer(int b)

**Parameters :**

b : value with which to initialize

**Integer(String s):**Creates a Integer object initialized with the int value provided by string representation. Defalut radix is taken to be 10.

**Syntax :** public Integer(String s)

throws NumberFormatException

**Parameters :**

s : string representation of the int value

**Throws :**

NumberFormatException : If the string provided does not represent any int value.

Programs:

Basic:

// Leap Year

// Reverse a given integer.

// Fibonacci series

// Integer Palindrome

// Factorial

// Prime number

// Armstrong number

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.

// Pyramid

Strings:

// String to Float

// Even length words in a String

// String reverse

// String Palindrome

// Type of characters and their count in a String / Characters Segregation.

// Number of words in a String.

// Duplicate words in Strings

// Sum of integers in a String.

// Calculate maximum value using ‘+’ or ‘\*’ sign between two numbers in a string

// Frequency of Characters

// Duplicate Characters

// Anagram

Two strings are anagrams if they are written using the same exact letters, ignoring space, punctuation, and capitalization. Each letter should have the same count in both strings. For example, Army and Mary are an anagram of each other.

Arrays:

// Bubble Sort

Bubble sort algorithm is known as the simplest sorting algorithm.

In bubble sort algorithm, array is traversed from first element to last element. Here, current element is compared with the next element. If current element is greater than the next element, it is swapped.

// Sort Array of Strings using sort method and without Sort Method.

// Average value of array elements.

// Insert an element into an array

// copy an array by iterating the array.

// Reverse an array of integer values

// duplicate values of an array of integer values.