WHAT IS JAVA ?

* **Object Oriented**: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
* **Platform independent**: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
* **Simple**:Java is designed to be easy to learn. If you understand the basic concept of OOP,Java would be easy to master.
* **Secure**: With Java's secure feature, it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
* **Architectural-neutral**:Java compiler generates an architecture-neutral object file format, which makes the compiled code to be executable on many processors, with the presence of Java runtime system.
* **Portable**: Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler inJava is written in ANSI C with a clean portability boundary which is a POSIX subset.
* **Robust**:Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
* **Multithreaded**: With Java's multithreaded feature, it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
* **Interpreted**:Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and lightweight process.
* **High Performance**: With the use of Just-In-Time compilers, Java enables high performance.
* **Distributed**:Java is designed for the distributed environment of the internet.
* **Dynamic**: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

BASICS OF JAVA?

* **Object** - Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
* **Class** - A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.
* **Methods** - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
* **Instance** **Variables** - Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

**Type casting :** Type casting is a way of converting data from one data type to another data type

Example : **class** Simple{

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

**byte** a=10;

**byte** b=10;

//byte c=a+b;//Compile Time Error: because a+b=20 will be int

**byte** c=(**byte**)(a+b);

System.out.println(c);

Conditional Statements

**If satements**

**Ex : public** **class** IfExample {

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

//defining an 'age' variable

**int** age=20;

    //checking the age

**if**(age>18){

       System.out.print("Age is greater than 18");

**If Else**

**Ex : public** **class** IfElseExample {

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

//defining a variable

**int** number=13;

//Check if the number is divisible by 2 or not

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

        System.out.println("even number");

    }**else**{

        System.out.println("odd number");

**JAVA DATATYPES**

* Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.
* **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

****

**OOPS CONCEPTS**

**Inheritance :** When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.*

**Polymorphism :** If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

* In Java, we use method overloading and method overriding to achieve polymorphism.
* Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc

**Abstraction :** Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing. In Java, we use abstract class and interface to achieve abstraction.

**Encapsulation :** Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

* A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

**VARIABLES**

* **Local variables**: Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
* **Instance variables**: Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
* **Class variables**: Class variables are variables declared within a class, outside any method, with the static keyword.

CONSTRUCTORS

* Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.
* Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.
* **Default constructor (no-arg constructor)** : A constructor is called "Default Constructor" when it doesn't have any parameter.
* **Parametrized constructor** : A constructor which has a specific number of parameters is called a parameterized constructor.

Example //Java Program to create and call a default constructor

**class** Bike1{

//creating a default constructor

Bike1(){System.out.println("Bike is created");}

//main method

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

//calling a default constructor

Bike1 b=**new** Bike1();

}

}

public class Puppy

public Puppy(String name){ // This constructor has one parameter, name. }

}

Operators

* **Arithmetic Operators** : Addition Subtraction Multiplication Division Modulus Increment Decrement
* **Relational Operators :** Greater than,less than ,not equals to
* Bitwise Operators : AND XOR Ones Complement Right Shift
* Logical Operators : AND OR NOT XOR
* Assignment Operators : Equals
* Misc Operators

LOOPS IN JAVA

* **while Loop**

while(Boolean\_expression)

{

//Statements

}

* **do...while Loop**

do

{

//Statements

}while(Boolean\_expression);

* **for Loop**

for(initialization;Boolean\_expression; update)

{

//Statements

}

* **Enhanced for loop in Java**

for(declaration : expression)

{

//Statements

}

BREAK/CONTINUE

* **The break Keyword** : The break keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement

public class Test

{ public static void main(String args[])

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

for(int x : numbers)

{

if(x ==30){ break;

} System.out.print( x );

System.out.print("\n"); } } }

* **The continue Keyword:** The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

for(int x : numbers)

{

if( x ==30)

{ continue; }

**SWITCH**

* A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

**Ex** switch(expression)

{

case value :

//Statements break;

//optional case value :

//Statements break;

//optional //You can have any number of case statements.

default://Optional //Statements

}

ARRAY

* **Java array :**  is an object which contains elements of a similar data types

// Short form

// (can only be used as initializer in a declaration)

int[] arr1 = { 1, 2, 77 };

// Initialize with 10 zeroes

int[] arr2 = new int[10];

// General form

int[] arr3 = new int[] { 1, 2, 77 };

**OVERLOADING/OVERRIDING**

* **Method overloading** is a feature of Java in which a class has more than one method of the same name and their parameters are different.”
* **Overriding** is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

**STATIC THIS AND SUPER KEYWORDS**

**Static :**

* If you declare any variable as static, it is known as a static variable.
* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.

**This :**

* Invoke current class constructor
* Invoke current class method
* Return the current class object
* Pass an argument in the method call
* Pass an argument in the constructor call

**SUPER**

* 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
* **THERE ARE FOUR ACCESS MODIFIERS KEYWORDS IN JAVA AND THEY ARE:**

|  |  |
| --- | --- |
| Modifier | Description |
| Default | declarations are visible only within the package (package private) |
| Private | declarations are visible within the class only |
| Protected | declarations are visible within the package or all subclasses |
| Public | declarations are visible everywhere |

**Abstraction** is the process of separating ideas from specific instances of those ideas at work.

abstraction is the process of separating ideas from specific instances of those ideas at work

**Abstract methods :** public abstract class GraphicObject {

// declare fields

// declare nonabstract methods

abstract void draw();

}

**Interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**

**syntax**

1. **interface** <interface\_name>{
3. // declare constant fields
4. // declare methods that abstract
5. // by default.
6. }

**Expection handling**

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
|  |  |
| throws | The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |



1. **public** **class** JavaExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //code that may raise exception
5. **int** data=100/0;
6. }**catch**(ArithmeticException e){System.out.println(e);}
7. //rest code of the program
8. System.out.println("rest of the code...");
9. }
10. }

FILES IN JAVA

**Java I/O** (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

**Stream**

* A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
* In Java, 3 streams are created for us automatically. All these streams are attached with the console.
* **System.out:**standard output stream
* **System.in:**standard input stream
* **System.err:**standard error stream

**ByteStream:** This is used to process data byte by byte (8 bits). Though it has many classes, the FileInputStream and the FileOutputStream are the most popular ones. The FileInputStream is used to read from the source and FileOutputStream is used to write to the destination. Here is the list of various ByteStream Classes:

| Stream class | Description |
| --- | --- |
| [BufferedInputStream](https://www.geeksforgeeks.org/java-io-bufferedinputstream-class-java/) | It is used for Buffered Input Stream. |
| [DataInputStream](https://www.geeksforgeeks.org/java-io-datainputstream-class-java-set-1/) | It contains method for reading java standard datatypes. |
| [FileInputStream](https://www.geeksforgeeks.org/java-io-fileinputstream-class-java/) | This is used to reads from a file |
| [InputStream](https://www.geeksforgeeks.org/java-io-inputstream-class-in-java/) | This is an abstract class that describes stream input. |
| [PrintStream](https://www.geeksforgeeks.org/java-io-printstream-class-java-set-1/) | This contains the most used print() and println() method |
| [BufferedOutputStream](https://www.geeksforgeeks.org/java-io-bufferedoutputstream-class-java/) | This is used for Buffered Output Stream. |
| [DataOutputStream](https://www.geeksforgeeks.org/dataoutputstream-in-java/) | This contains method for writing java standard data types. |
| [FileOutputStream](https://www.geeksforgeeks.org/creating-a-file-using-fileoutputstream/) | This is used to write to a file. |
| [OutputStream](https://www.geeksforgeeks.org/java-io-outputstream-class-java/) | This is an abstract class that describe stream output. |

**Input stream**

* InputStream is an abstract class of Byte Stream that describe stream input and it is used for reading and it could be a file, image, audio, video, webpage, etc. it doesn’t matter. Thus, InputStream read data from source one item at a time.
* OutputStream is an abstract class of Byte Stream that describes stream output and it is used for writing data to a file, image, audio, etc. Thus, OutputStream writes data to the destination one at a time.

**Buffered Reader**

* Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.

**InputStreamreader**

* An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

**BufferedWriter**

* Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits [Writer](https://www.javatpoint.com/java-writer-class) class. The buffering characters are used for providing the efficient writing of single [arrays](https://www.javatpoint.com/array-in-java), characters, and [strings](https://www.javatpoint.com/java-string).

**File Writer**

* Java FileWriter class is used to write character-oriented data to a [file](https://www.javatpoint.com/java-file-class). It is character-oriented class which is used for file handling in [java](https://www.javatpoint.com/java-tutorial).
* Unlike FileOutputStream class, you don't need to convert string into byte [array](https://www.javatpoint.com/array-in-java) because it provides method to write string directly.

**Outputstreamreader**

* The OutputStreamWriter class of the java.io package can be used to convert data in character form into data in bytes form. It extends the abstract class Writer . ... It is also known as a bridge between byte streams and character streams. This is because the OutputStreamWriter converts its characters into bytes.

**Inputstreamreader**

* An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset . The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

**CharacterStream:** In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character. Though it has many classes, the FileReader and the FileWriter are the most popular ones. FileReader and FileWriter are character streams used to read from the source and write to the destination respectively. Here is the list of various CharacterStream Classes:

| Stream class | Description |
| --- | --- |
| [BufferedReader](https://www.geeksforgeeks.org/java-io-bufferedreader-class-java/) | It is used to handle buffered input stream. |
| [FileReader](https://www.geeksforgeeks.org/file-handling-java-using-filewriter-filereader/) | This is an input stream that reads from file. |
| [InputStreamReader](https://www.geeksforgeeks.org/java-io-inputstreamreader-class/) | This input stream is used to translate byte to character. |
| OutputStreamReader | This output stream is used to translate character to byte. |
| [Reader](https://www.geeksforgeeks.org/java-io-reader-class-java/) | This is an abstract class that define character stream input. |
| [PrintWriter](https://www.geeksforgeeks.org/java-io-printwriter-class-java-set-1/) | This contains the most used print() and println() method |
| [Writer](https://www.geeksforgeeks.org/java-io-writer-class-java/) | This is an abstract class that define character stream output. |
| [BufferedWriter](https://www.geeksforgeeks.org/io-bufferedwriter-class-methods-java/) | This is used to handle buffered output stream. |
| [FileWriter](https://www.geeksforgeeks.org/file-handling-java-using-filewriter-filereader/) | This is used to output stream that writes to file |

1. **import** java.io.\*;
2. **public** **class** FileDemo {
3. **public** **static** **void** main(String[] args) {
5. **try** {
6. File file = **new** File("javaFile123.txt");
7. **if** (file.createNewFile()) {
8. System.out.println("New File is created!");
9. } **else** {
10. System.out.println("File already exists.");
11. }
12. } **catch** (IOException e) {
13. e.printStackTrace();
14. }
15. **import** java.io.\*;
16. **public** **class** FileExample {
17. **public** **static** **void** main(String[] args) {
18. File f=**new** File("/Users/sonoojaiswal/Documents");
19. String filenames[]=f.list();
20. **for**(String filename:filenames){
21. System.out.println(filename);
22. }
23. }

**WRAPPER CLASS**

The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |

## Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

1. **public** **class** WrapperExample1{
2. **public** **static** **void** main(String args[]){
3. //Converting int into Integer
4. **int** a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer explicitly
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
8. System.out.println(a+" "+i+" "+j);
9. }}

## Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

**Wrapper class Example: Wrapper to Primitive**

1. //Java program to convert object into primitives
2. //Unboxing example of Integer to int
3. **public** **class** WrapperExample2{
4. **public** **static** **void** main(String args[]){
5. //Converting Integer to int
6. Integer a=**new** Integer(3);
7. **int** i=a.intValue();//converting Integer to int explicitly
8. **int** j=a;//unboxing, now compiler will write a.intValue() internally
10. System.out.println(a+" "+i+" "+j);
11. }}

**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).

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(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 boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(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 boolean containsAll(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 boolean isEmpty() | 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. |

1. **import** java.util.\*;
2. **class** TestJavaCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
    * + - **[ArrayList](https://beginnersbook.com/java-collections-tutorials/" \l "1)**
        - [**LinkedList**](https://beginnersbook.com/java-collections-tutorials/#2)
        - [**Vector**](https://beginnersbook.com/java-collections-tutorials/#3)
        - [**HashSet**](https://beginnersbook.com/java-collections-tutorials/#4)
        - **[LinkedHashSet](https://beginnersbook.com/java-collections-tutorials/" \l "5)**
        - [**TreeSet**](https://beginnersbook.com/java-collections-tutorials/#6)
        - [**HashMap**](https://beginnersbook.com/java-collections-tutorials/#7)
        - **[TreeMap](https://beginnersbook.com/java-collections-tutorials/" \l "8)**
        - **[LinkedHashMap](https://beginnersbook.com/java-collections-tutorials/" \l "9)**
        - **[Hashtable](https://beginnersbook.com/java-collections-tutorials/" \l "10)**
        - [**Iterator and ListIterator**](https://beginnersbook.com/java-collections-tutorials/#11)
        - [**Comparable and Comparator**](https://beginnersbook.com/java-collections-tutorials/#12)
        - [**Java Collections Interview Questions**](https://beginnersbook.com/java-collections-tutorials/#13)

**Generics** mean **parameterized types**. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.   
An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

// To create an instance of generic class

BaseType <Type> obj = new BaseType <Type>()

**Note:** In Parameter type we can not use primitives like

'int','char' or 'double'.

1. List list = **new** ArrayList();
2. list.add(10);
3. list.add("10");
4. With Generics, it is required to specify the type of object we need to store.
5. List<Integer> list = **new** ArrayList<Integer>();
6. list.add(10);
7. list.add("10");// compile-time error
8. **import** java.util.\*;
9. **class** TestGenerics1{
10. **public** **static** **void** main(String args[]){
11. ArrayList<String> list=**new** ArrayList<String>();
12. list.add("rahul");
13. list.add("jai");
14. //list.add(32);//compile time error
16. String s=list.get(1);//type casting is not required
17. System.out.println("element is: "+s);
18. Iterator<String> itr=list.iterator();
19. **while**(itr.hasNext()){
20. System.out.println(itr.next());

ITERATOR

An Iterator is an object that can be used to loop through collections, like [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) and [HashSet](https://www.w3schools.com/java/java_hashset.asp). It is called an "iterator" because "iterating" is the technical term for looping.

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

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

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

// Make a collection

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

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// Get the iterator

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

// Print the first item

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

}

}

**LOOPING THROUGH COLLECTION**

while(it.hasNext()) {

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

}

**Testing**

**RTM(Requirement traceability matrix)**

* **Requirement Traceability Matrix (RTM)** is a document that maps and traces user requirement with test cases. It captures all requirements proposed by the client and requirement traceability in a single document, delivered at the conclusion of the Software development life cycle. The main purpose of Requirement Traceability Matrix is to validate that all requirements are checked via test cases such that no functionality is unchecked during Software testing.

**Test plan**

* A Test Plan refers to **a detailed document that catalogs the test strategy, objectives, schedule, estimations, deadlines**, and the resources required for completing that particular project. Think of it as a blueprint for running the tests needed to ensure the software is working properly – controlled by test managers

**Test Scenario**

* **Scenario Testing** in software testing is a method in which actual scenarios are used for testing the software application instead of test cases. The purpose of scenario testing is to test end to end scenarios for a specific complex problem of the software. Scenarios help in an easier way to test and evaluate end to end complicated problems.

**Test case**

* A test case is a document, which has a set of test data, preconditions, expected results and postconditions, developed for a particular test scenario in order to verify compliance against a specific requirement.

**Test Script**

* The test script is a set of instructions to test an application automatically.
* Test Scripts are a line-by-line description containing the information about the system transactions that should be performed to validate the application or system under test. Test script should list out each step that should be taken with the expected results.

**Test data**

* **Test Data in Software Testing** is the input given to a software program during test execution. It represents data that affects or affected by software execution while testing. Test data is used for both positive testing to verify that functions produce expected results for given inputs and for negative testing to test software ability to handle unusual, exceptional or unexpected inputs.

Test Straegy

* A test strategy is **a guideline to be followed to achieve the test objective and execution of test types mentioned in** the testing plan. ... It contains the scope and objective, business issues, testing approach, test deliverables, defect tracking approach, automation, and risks.