**Core Java:**

* **Introduction**
* **Java Language**
* **OOPS Concepts**
* **Packages**
* **Exception Handling**
* **Threads**
* **Java.lang package**
* **Java.util collections & thread executors package**
* **Generics**
* **Lambda Expressions**
* **Java.util stream (Discuss)**

**Advance Java:**

* **Servlets, JSP, JDBC**
* **Spring**
	+ **Spring IOC**
	+ **Spring MVC**
	+ **Spring ORM integration**
	+ **Spring Aspect**
	+ **Spring Boot**
	+ **Spring Cloud (discuss)**
* **Hibernate**
	+ **Spring Hibernate Integration**
	+ **HQL**
	+ **Criteria Query**
	+ **ORM Relations**
	+ **Cache Support**
* **No SQL (Mongo DB or Postgres)**
* **Amazon Web Services**

**Servers: Tomcat**

**Build Tool: Maven**

**CICD: Jenkins**

**Download:** Eclipse (Luna Service Release 2 (4.4.2)), GitHub, Postgres, Tomcat

Java: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> (download java SE Development Kit)

Eclipse: <https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunasr2> and select corresponding OS compatible

Tomcat download: [https://tomcat.apache.org/download-70.cgi#7.0.6](https://tomcat.apache.org/download-70.cgi#7.0.67) . After installation completed, please bring up the server and verify installation successful.

Maven download: [https://maven.apache.org/download.cgi](https://maven.apache.org/download.cgi%C2%A0%22%20%5Ct%20%22_blank) After installation, make sure, its properly configured and when you run. Maven version 3.3.9 or above

Postgres download: <https://www.postgresql.org/download/> click on windows->download the installer -> select 9.3.15 or higher and your OS

Create an account on github : **github.com**

Create an for aws :: <https://aws.amazon.com/>

<http://kaizen-koka.com/2015/12/13/download-install-setup/>

**Difference between C, C++ and JAVA:**

C, C++ compiler converts **.c** file in to **.obj** file which is nothing but machine code and linking will generate .exe file.

Where as in Java by compiling, .**java** will be converted in to .**class** file (**byte code**) then JVM executes into native code.

**Concepts:.**

1. Java is platform independent but JVM dependent. JVM is platform dependent. When you compile java file it is first converted in to **.class** file (byte code) then when executing it will become a native code.
2. JDK, JRE, JVM
3. path, classpath
	1. Classpath represents where the required class files are available. Classpath will be used by java compiler and JVM.
	2. Path is to represent where binary exe are available. Nothing but inside JDK\bin
4. Java memory allocation.

**Java language:**

Character Set:

Digits, alphabets, special symbols ( \_ $ )

Keywords:

public, protected, private, interface, class, abstract, implements, extends, int, byte, short, long,

float, double, char, boolean, new, import, package, try, catch, finally, final, throw, throws, if, else, switch, case, for, do, while, continue, static, void, transient, synchronized, instance, break, return,

volatile, this, default, final, super, goto, const

**Identifiers**: name for classes, variables, methods (first character must be a letter, \_, $)

**Datatypes**:

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Size(byte)** | **Initial /default value** |
| byte | 1 | 0 |
| short | 2 | 0 |
| int | 4 | 0 |
| long | 8 | 0 |
| float | 4 | 0.0f |
| double | 8 | 0.0 |
| char | 2 | ‘\u0000’ (nothing but null) |
| boolean | 1 bit (not really defined) | False |

**Variables**:

 An identifier which used to store some value.

**Constants**: are the ones which never change value after initialization. Represents through keyword **final**

**Literals**:

* Integer Literals
	+ Decimal integer literals
	+ Octal Integer literals
	+ Hexa Decimal Integer literals
* Floating Literals
	+ Decimal notation
	+ Exponentials notation
* Character Literals : enclosed by single quotes
* String Literals: enclosed by double quotes.
	+ Escape sequences nothing but character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.
	+ Example : \”, \\, \n, \\

**Operators**:

* **Athematic Operators : +, - , \*, /, %**
* Relational Operators :
* **Assignment Operators : =, +=**
* **Logical Operators: &&, ||, !**
* Unary Operators: ++,--
* Ternary operator: (?:)
* Bitwise Operators : >> << & | ^

**Control Statements**: if-then, if-then-else, switch, while, do-while, for, break, continue

Java comes with the following 4 OOPS (Object Oriented Programming System) concepts

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Object:

* Anything in this world is an object .
* No two objects are identical
* Objects shares two characteristics
1. State or Properties or Attributes
2. Behavior or Operations

In java – properties of an object will be represented by variables and operations will be by methods.

 Ex: 1) Car (object) - gear, brakes, seat (properties) – changing the gear, applying the break (operations).

 2) Pen (object) – cap, nib, ink (properties) – closing cap, writing on pen, throwing cap at someone (operations).

**Abstraction:** is nothing but providing required **properties** & **operations** of an object by hiding internal details.

**Encapsulation:** Writing properties and operations that are going to operate on properties in a single entity called encapsulation. In other words, in java – properties are called variables and operations are called as methods & entity is called as class. I.e. writing variables and methods which are going to use variables in to a class is called encapsulation.

In java – encapsulation can be achieved by private variables and public methods of a class.

**Inheritance:** Writing a new class by using the functionality of an existing class is called as inheritance. Existing class is known as base class or super class or parent class. New class is called as subclass or derived class or child class.

Inheritance used for reusability.

**Polymorphism:** is nothing but one operation behaving differently in different situation. I.e. one operation will have different implementations.

 Java comes with 2 types of polymorphisms.

1. Compile time polymorphism
2. Runtime polymorphism.

**Object & Class:** Class is a logical entity whereas object is a physical entity. Class is an entity which contains variables and methods. These two are called as members of the class.

class Class-name {

 datatype var1, var2;

 datatype var3, var4;

 returntype mtd-name(arg1){

 // body – valid java stmts

 }

}

Creating the object:

 Syntax: Classname object-name = new Classname();

 Ex: **BellInfo b** = new BellInfo ();

* Declaration: The code set in bold are all variable declarations that associate a variable name with an object type.
* Instantiation: The new keyword is a Java operator that creates the object.
* Initialization: The new operator is followed by a call to a constructor, which initializes the new object.

Sample Program:

Variables: Java comes with the following different type of variables.

* Primitive variables
* Reference variables
* Local variable
* Instance variable
* Static variables
* Final Variables

Primitive variable:

1. Variable declared with primitive datatype (int, long, double,…) is called primitive variables
2. Default value for primitive variables depends on primitive datatype used.
3. Memory size of primitive variable depends on primitive type what we use.

Reference variable:

1. Variable declared with class type is called reference variable.
2. Default value of reference variable is null for all classes.
3. Memory size of reference variable is 8 bytes (fixed).

Local variable:

1. Variable declared inside the method are called as local variable.
2. Scope of local variable is with in the method, where it is declared.
3. We have to initialize local variable explicitly, otherwise the following compile time error will come.

 Eg: “Variable xyz might not initialized”.

1. Local variable can be primitive variables and reference variables.
2. Memory will be allocated for local variable, when your invoking the method.
3. JVM allocates the memory for local variables in the stack frame where the definition is stored.

Instance Variable:

1. Variables declared inside the class without static keyword are called as instance variables.
2. Instance variables are no need to initialize, i.e. when your not initializing instance variables JVM will initializes them with default values.
3. Scope of instance variables are within the class where it is declared.
4. Instance variable can be primitive variables and reference variables.
5. JVM allocates memory for instance variables when our creating an object.
6. JVM allocates memory for instance variables in the HEAP.

Static Variable:

 Static is a modifier in java, which can be used for various methods and for classes (only for inner classes not for top level classes).

 Members with static keywords are called as Static members.

1. Static variable are declared with the static keyword for e.g.: static int a;
2. Only one copy of memory, will be allocated for static variable, for all the objects. i.e. all objects will share same memory location.
3. Memory will be allocation for static variable when JVM is loading the class into the memory.
4. Local variables cannot be the static variables.
5. Static variables belongs to class, so we can call static variable with the class name directly.

 i.e. static int a;

classname. Variable name

 Hello. a;

Static Methods:

1. Methods defined with static keyword are called as static methods.
2. Static methods belongs to class.
3. We can invoke static methods with classname or object of the class.
4. Inside a static methods we can use static variables and static methods.
5. We can’t use non static variables and non-static methods directly i.e. without objects, but we can use with an object.
6. Inside non-static methods we can use static members and non-static members.

Eg: class ABC

 { int a=10;

 Static int b=20;

 Void display1()

 { m1();

 m2();

 System.out.println(a);

 System.out.println(b);

}

Static void didplay2()

{ ABC x=new ABC();

 x.m1();

 System.out.println(x.a);

 m2()

 System.out.println(b);

}

Void m1()

{ System.out.println(“ I am m1()”);

}

Static vois m2()

{ System.out.println(“ I am m2()”);

} }

 Class demo1

{ public static void main( String arg[])

{ ABC.display2 ();

 ABC Obj=new ABC ()

 Obj.display1 ();

 Obj.display2 ();

} }

1. Non static variable “a” can’t be referenced from a static context.
2. Non static method display1 () can’t be referenced from a static context.
3. Static modifier is not allowed for we level classes.

Final variable:

1. Variable declared with final modifier are called final variable.
2. Final variable are also called as constants.
3. Final variable is allowed for instance variables, static variables and local vaiables.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Modifiers | **public**  | **protected** | **private** | **default** | **final**  | **static**  | **transient** | **volatile** | **synchronized** | **abstract** |
| **Instance Variables** | $$√$$$$√$$ | $$√$$ | $$√$$ | $$√$$ | $$√$$ | $$√$$ | $$√$$ | $$√$$ | $$×$$ | $$×$$$$√$$ |
| **Methods** | $$√$$ | $$√$$ | $$√$$$$√$$ | $$√$$$$√$$ | $$√$$ | $$√$$ | $$√$$ | $$√$$ |
| **Methods Variables** | $$×$$$$×$$ | $$×$$ | $$×$$ | $$×$$$$√$$$$√$$$$√$$ | $$√$$$$×$$$$√$$ | $$×$$ | $$×$$ | $$×$$ | $$×$$ | $$×$$$$×$$ |
| **Classes** | $$√$$ | $$×$$ | $$×$$ | $$×$$ | $$×$$ | $$×$$ | $$×$$ | $$√$$ |

* When you declare a class as final then that Class can’t be extended. You can define a class as final to avoid from modifying the class in scenarios like security. Built-in class String is declared as final class. So that no one modify the method. **Marking class final doesn't mark its fields as final and as such doesn't protect the object properties but the actual class structure instead.**
* When you declare a variable as static which means – it will be available across.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Same Package | Different Package |
|  |  | By Creating Obj | By Extending | By Creating Obj | By Extending  |
|  | Class A | Class B | Class C | Class D | Class E |
| Private | YES | NO | NO | NO | NO |
| default | YES | YES | YES | NO | NO |
| protected | YES | YES | YES | NO | YES |
| public | YES | YES | YES | YES | YES |

**Constructors**:

Super ()

Inheritance: //TODO scenarios

Polymorphism

|  |  |  |
| --- | --- | --- |
| **Property** | **Overloading** | **Overriding** |
| Method Name  | Same | same |
| Arugments type | different | same  |
| Method Signature(Metod anme+Argument type) | different | same |
| return type | Anything allowed | same or sub class with co-varient |
| private, static & final | can be overloaded | can't be overidden |
| Access Modifiers | No Restrictions | We can increase the scope at child class |
| throws (checked exceptions) | No restrictions | Parent should throw same checkedException or its parents |
| method resoultion | always takes care by compiler based on reference type | Always takes care by JVM based on runtime Object |

Accessing super class from SubClass not allowed.

**Simple Inheritance**:

Class A{

}

Calss B extends A{

}

**Multi Level Inheritance**

Class A{

}

Class B extends A{

}

Class C extends B{

}

**Hierarchical Inheritance**

Class A{

}

Class B extends A{

}

Class C extends A {

}

**Multiple Inheritance**

Class A {

}

Class B{

}

Class C extends A, B { **-- NOT allowed**

}

**Hybrid Inheritance**

Class A{

}

Class B extends A{

}

Class C extends A{

}

Class D extends B,C { **NOT allowed**

**}**

**Strings empty and null checks**

+-------+-----------+----------------------+

| s | s == null | s.isEmpty() |

+-------+-----------+----------------------+

| null | true | NullPointerException |

| "" | false | true |

| "foo" | false | false |

+-------+-----------+----------------------+

Java Virtual Machine (JVM):

**Heap Memory** Objects Instant Variables **Stack Memory** Local Variable

Display1()

{ (a)

}

Display2 ()

{ ( b)

}

OBJ 1

A B

20

10

A B

Obj2

31

x

 

66

55

A B

Obj 3

 Frames

Heap Memory:

* Memory allocated for reference variable.
* Memory allocated for object.

Stack Memory:

* Method definitions will be stacked in stack frame.
* For each new method in a class, there will be one frame will be allocated in stack.

**JVM:**

1. Monitor all the statements.
2. If an error/exception, then identifies corresponding Exception class.
3. Create the object for Exception class.
4. Throws the object.
5. Catch the object & terminate the Program
6. JVM displays info in that Object.

 Exception classes are in package java.lang.\* 

There are two type of problems.

* Exceptions which can be handled
* Error which can’t be handled
* All exceptions in Java are classes.
* All exception are subclass of java.lang.Exception
* In all exception classes – there will be only constructors. We don’t have any methods. All Exception subclasses are using superclass Throwable methods. Like getMessage(); printStackTrace();
* We handle the Exception with the following 5 keywords.
* try
* catch
* finally
* throw
* throws

 **try:** block is used toplace the statements need to be monitored in case of abnormal behavior.

 **catch:** block is used to catch the exceptions raised by try block.

* Catch should follow by try.
* Other statements are not allowed between try and catch.
* For one try – we can write more than one catch block.
* In case of more than one catch block – order of exceptions must be subclass to superclass.

 **finally:** block is used for the statements which need to be executed irrespective of exceptions occur in try block.

* Only one finally is allowed for each try block.
* When you have statements like System.exit(0) in try block then finally will not be executed.
* When the running thread interrupted while executing try or catch statements by kiiling the thread then finally won’t be executed.
* When try block goes in to infinite loop then finally won’t be executed.

try { try{ try{ try{

} } } }

catch () { catch(){ finally{ catch{

} } } }

finally { catch (){

} }

**throws:** key word is used to specify the method level exceptions. Statements inside a method may throw some exceptions. If you want to handle them – you can do that by surrounding those statements with try catch block. If you don’t want to handle the exceptions inside the method then throw them to calling method using throws at method level.

public void mtd2() throws ArrayIndexOutOfBoundsException, ArithmeticException {

//valid java statements

}

In the above method mtd2() – we are not specifying the exceptions but the we are communicating the caller method of mtd2() to handle those exception that I’m throwing.

**throw:** is used to throw the exceptions in ur own. JVM handles built in exception. JVM can’t handle application level exceptions or user defined exceptions.

**User Defined Exceptions:**

* Write your own exception class by extending java.land.Exception or java.lang.RuntimeException
* Write one or more constructors based on your requirement.
* Override toString() method.
* If required override equals and hascode method.

**Types of Exceptions:**

 ****

**Checked Exceptions examples:**

* IOException
* [SQLException](http://javarevisited.blogspot.com/2012/01/javasqlsqlexception-invalid-column.html)
* DataAccessException
* [ClassNotFoundException](http://javarevisited.blogspot.com/2011/08/classnotfoundexception-in-java-example.html)

**UnChecked Exceptions Examples:**

* NullPointerException
* ArrayIndexOutOfBound
* IllegalArgumentException
* IllegalStateException

**Thread Life Cycle**

Create and start thread

When a resources avaliable

Notify(), notifyall()

Run() CPU Scheduling alogorithm

Stop()

After time is over

LRE: Least Rcently entered

Stop()

Killing

LRE

Sleep(n)

Wait()

When resource is not allowed

**Errors:**

Errors are thrown by java runtime system and indicate some irrecoverable conditions that occur during the program execution. Program should need to be fixed or so, inorder to rerun the program.

## Eg: java.lang. OutOfMemoryError, java.lang.StackOverflowError

**Exception** : An exception is an abnormal condition in which the normal execution of code got hampered.

* Few Facts: Exception occur at runtime during program execution.

*public class java.lang.Throwable implements java.io.Serializable {*

 *public java.lang.Throwable();*

 *public java.lang.Throwable(java.lang.String);*

 *public java.lang.Throwable(java.lang.String, java.lang.Throwable);*

 *public java.lang.Throwable(java.lang.Throwable);*

 *public java.lang.String getMessage();*

 *public synchronized java.lang.Throwable getCause();*

 *public java.lang.String toString();*

 *public void printStackTrace();*

*}*

**Methods which are in thread class:**

isAlive();

yield();

join();

sleep();

start();

run();

stop();

**Method which are in Object class :**

Notify()

notifyAll()

wait()

isAlive() : Method is used to check whether the thread is alive or dead.

Yield(): When you called yield() method on the running thread. Running thread will give a chance to another thread which is in ready to run state. i.e, Running thread will be placed back to ready to run state. Some other thread which has the highest priority will be moved from ready-to-run state to running state but JVM doesn’t guarantee.

Join(): When you call join method on the threads then the join thread will be completed first and then parent thread from where the other threads are started. Example, from main method 2 child threads t1 and t2 are started. When I call join method on t1 and t2 then main thread wait until child thread finishes it task.

**Daemon Threads**: Are the service threads. Daemon threads live as long as dependent threads are running. You can make any thread as deamon by using the following methods.

Void setDaemon(boolean);

Boolean isDaemon(); We can check whether the thread is daemon thread or not.

Threads can created in 2 way :

1. By extending thread class
2. By implementing runnable interface.

 Implementing through runnable interface is best option. When you extend the thread class, we don’t have a chance to extend any other class, bcoz no multiple inheritance with java classes, where as when you implement runnable interface we can extend some other class also. And when you extend thread classes all the implemented methods in the thread class will be loaded in to the memory. Where are when you implement runnable interface we don’t get such problem.

 Scheduling Algorithm:

1. **Primitive Scheduling Alg :** Suppose 2 threads t1 and t2 enter into ready to run state with the priorities 4 and 6 respectively. Now T2 gets the CPU time, because of it has highest than T1. Now T2 is running and t1 is in ready to run state. Another thread called t3 with priority 9 and entered in to ready to run state. And t3 has highest priority than the running thread. And then t2 will be preempted and t3 enter into running state.
2. **Time slice or Round Robin Alg** : In this alg, each thread will be given fixed amount of time and CPU time will be allocated for each thread based on FCFS(First come First Serve) or Primitive Algorithm and thread will be running state for specified quantum time. After the time is over thread will be back to ready to run state and another thread will be entered in to running state and so on…

  **Dead Locks :**

 Example : Thread t1 which is holding resource r1 is waiting for resource r2 and thread t2 which is holding a resource R2 and waiting for resource R1. T1 releases the resource R1 after serving R2 and T2 releases Resource R2 after serving R1. They won’t release the resource and they won’t get the resource also. Simply T1 and t2 will be blocked. No chance to move from out of block state. This situation is called dead lock.

Thread Priorities: We can give an integer number ranging from 1 -10 as priority to the threads. They are 3 constraints defined inside the thread class.

1. MIN\_PRIORITY - 1
2. NORM\_PRIORITY -5
3. MAX\_PRIORITY - 10

 We have two methods to find the priority of the thread and to change the priority of the thead.

1)int getPriority()

2)void serPriorities(int)

 **Synchronization**: In multi-threading environment when all threads are accessing the object concurrently. Some time we can get inconsistent results. To avoid inconsistent results, I don’t want to allow all the threads to access the object concurrently. I want to allow one-by-one to access the object. This is known as Synchronization.

When you synchronize the object, object will be locked and will be monitored. Threads which would like to use locked objects will enter in to the monitor and uses the locked object. After using the object, come out of the monitor. The threads outside the monitor unable to access the locked objects. We can do the synchronization using the modifier called as synchronize modifier. We can use this modifier in 2 ways,

1. Method level synchronization
2. Block level synchronization

Synchronized modifier is not allowed for classes, interfaces and variables.

When you call any synchronized method with an object – that object will be locked. Then no other threads use that object.

When you are using block level synchronization, you should pass any object to the block. And that passed object will be synchronized or locked.

Note: In method level synchronization – Object of the class which contains synchronized method will be locked. Where as in block level synchronization, the objects which we are passing to synchronized block (any object ) will be locked.

**Wrapper Class**

**Primitive type Wrapper Class**

 boolean Boolean

char Character

Byte Byte

Short Short

int Integer

long Long

float Float

double Double

 We have 8 primitive datatypes, there are 8 wrapper classes corresponding to 8 primitive datatypes.

These 8 wrapper classes are useful for the following things.

1. We can add only objects to the collections classes. In this case, we need object representative of primitive variable.
2. We can do variety of conversions using the methods provided in the wrapper classes.

Primitive

1. String to String(xxx)
2. Wrapper XXX valueOf(xxx)
3. Xxx’=
4. Byte byteValue()

Shrt shortValue()

Int intValue()

Long longValue()

Float floatValue()

Double doubleValue()

1. Byte b1= new Byte(10);

Byte b2= new Byte(“99”);

Byte b3= new Byte(“A99”); ×

**Types of Conversions:**

1. Primitive type to wrapper onject.

Eg: class WDemo

{ PSVM(String As[])

{

// primitive to wrapper

Int i=99;

Interger ii1= new Interger(i);

Interger ii2= Interger. valueOf(i);

System.Out.Println(iii);

// String to Wrapper

String str= “123456”;

Long str= new Long(str);

Long i= Long. parseLong(str);

Long i2= new Long(I);

Long i3= Long. valueOf(1);

S.O.P(112);

// wrapper to primitive

Int x= ii1.intValue();

Byte b= ii1.byteValue();

S.O.P(x);

// primitive to String

Double d= 99.99;

String str= Double.toString(d);

S.O.P(Str);

// wrapper to string

Short ss= new Short(“54jk”);

String str1= ss.toString();

S.O.P(str1);

}

}

**Boolean and Character:**

1. Character ch= new Character(‘a’);

 Boolean b1= new Boolean(true);

 Boolean b2= new Boolean(“false”);

1. parseBoolean(“false”);

 parseCharacter(‘c’)

1. valueOf(boolean)

valueOf(char)

1. String toString(Boolean);

 String toString(char);

**Advantage of collections over Arrays**

|  |  |
| --- | --- |
| **Array**  | **Collections** |
| Arrays are not resizable. | Provides lots of useful datatypes and resizable dynamically |
| **Arrays doesn't allow generics** | **Collections allow generics** |
| Array will store both primitive and objects | Doesn't store primitive datatypes. Stores only objects. |
| size/length of Array must be provided | Collections will be created with default size |
| Performane - both will be same. | Performance both will be same. As ArrayList internally uses Arrays and each time when it reaches maximum. Its needs be created new array and copy data from old Array |
| You must provide length | ArrayList initial size is 0 and intial capacity is 10. When it reaches maximum , it increases the size with int newCapacity = (oldCapacity \* 3)/2 + 1; |

**AutoBoxing (Java 1.5 feature):** As any Java programmer knows, you can’t put an int (or other primitive value) into a collection. Collections can only hold object references, so you have to *box* primitive values into the appropriate wrapper class (which is [Integer](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Integer.html) in the case of int). When you take the object out of the collection, you get the Integer that you put in; if you need an int, you must *unbox* the Integer using the intValue method. All of this boxing and unboxing is a pain, and clutters up your code. The autoboxing and unboxing feature automates the process, eliminating the pain and the clutter.

**Generics:** Generics shifts error prone scenarios from runtime to compile time. Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

Code that uses generics has many benefits over non-generic code:

* Stronger type checks at compile time.
A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Elimination of casts.
The following code snippet without generics requires casting:

* List list = new ArrayList();
* list.add("hello");
* String s = **(String)** list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

**List vs Set**

|  |  |
| --- | --- |
| **List**  | **Set** |
| An ordered collection. User of this interface has precise control over where the element is inserted. | An unordered collection. It doesn't gurantee any order |
| Allows duplicates | Doesn't allow duplicates |
| Ordered based on index of the element inserted | inserted in to Random order |
| It allows as many null values as you want | allows only one null value. |
| Popular implementations of List are ArrayList, Vector, LinkedLIst | Popular implementations of Set are HashSet, LinkedHashSet, TreeSet |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **ArrayList** | **Linked List** |
|  | Synchronized | Not Synchronized | Not Syncronized |
|  | Ordered insertion based on index | Ordered insertion based on index | Ordered insertion based on index |
| **Search** | 3 | 1 | 2 |
| **Insert** | 3 | 2 | 1 |
| **Remove** | 3 | 2 | 1 |
|  |  | If you know how much size you want always define the size of an arraylist. Increasing the size of an arraylist costlier |  |
|  | Thread safe | Not thread safe | Not thread safe |
|  | Increases the size with double. If intial size is 10 then it will make it to 20 when it reaches maximum | Increases the size with half. If intial size is 10 then it will make it to 15 when it reaches maximum | LinkedList doesn't need to initialize its size as it doesn't require to capture the memory. It internally implements doubly linked list |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HashSet** | **TreeSet** | **LinkedHashSet** |
|  | HashSet is Implemented using a hash table | TreeSet is implemented using a tree structure | implemented as a hash table with a linked list  |
| **add** | O(1) | O(log (n)) | O(1) |
| **remove** | O(1) | O(log (n)) | O(1) |
| **contains** | O(1) | O(log (n)) | O(1) |
|  | Random | it will sort items in sorting order | insertion order |
|  |  | need to implement Comparable or Comparator Interface to execute sorting |  |
|  | not Thread safe | not Thread safe | not Thread Safe |
| **Speed** | 1 | 3 | 2 |
|  | allows only one null | not allowed , throws Nullpointer Exception | allows only null |
|  | HashSet is Implemented using a hash table |  |  |
|  | equals() method for comparision | compareTo() for ordering | equals() for comparision |

**java.lang.Comparable:**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by [Collections.sort](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List-)

Int compareTo(Object o);

**Java.util.Comparator interface:**

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as [Collections.sort](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List,%20java.util.Comparator)) or [Arrays.sort](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort(T[],%20java.util.Comparator))) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as [sorted sets](https://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html) or [sorted maps](https://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)), or to provide an ordering for collections of objects that don't have a [natural ordering](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html).

**Int** [**compare**](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html#compare(T,%20T))([**T**](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) o1, [**T**](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) o2)

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Comparable** | **Comparator** |
| Sorting logic | Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects | Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id,name etc. |
| Implementation | Class whose objects to be sorted must implement this interface.e.g Country class needs to implement comparable to collection of country object by id | Class whose objects to be sorted do not need to implement this interface.Some other class can implement this interface. E.g.-CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id |
| Sorting method | int compareTo(Object o1)This method compares this object with o1 object and returns a integer.Its value has following meaning1. positive – this object is greater than o12. zero – this object equals to o13. negative – this object is less than o1 | int compare(Object o1,Object o2)This method compares o1 and o2 objects. and returns a integer.Its value has following meaning.1. positive – o1 is greater than o22. zero – o1 equals to o23. negative – o1 is less than o1 |
| Calling method | Collections.sort(List)Here objects will be sorted on the basis of CompareTo method | Collections.sort(List, Comparator)Here objects will be sorted on the basis of Compare method in Comparator |
| Package | Java.lang.Comparable  | Java.util.Comparator |

// Scanner nextInt() -- Trouble shoot