# BASICS

## WHY JAVA ?

1) Java is a platform-independent language because it has runtime environment i.e JRE and API

2) Java codes are compiled into byte code or machine-independent code.

This byte code is run on JVM (Java Virtual Machine).

**Features:**

1) Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand.The confusing

and ambiguous concepts of C++ are either left out in Java or they have been re-implemented

in a cleaner way.

Eg : Pointers and Operator Overloading are not there in java but were an important part of C++.

2) Object Oriented

3) Robust

Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time

error checking and runtime checking.

But the main areas which Java improved were Memory Management and mishandled

Exceptions by introducing automatic Garbage Collector and Exception Handling

4) Platform Independent

Unlike other programming languages such as C, C++ etc which are compiled into platform

specific machines. Java is guaranteed to be write-once, run-anywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent

and can be run on any machine, plus this bytecode

format also provide security. Any machine with Java Runtime Environment can run Java Programs.

5) Secure

6) Multi Threading

8) Portable

Java Byte code can be carried to any platform. No implementation dependent features.

Everything related to storage is predefined, example: size of primitive data types

9) High Performance

Java is an interpreted language, so it will never be as fast as a compiled language like C or C++.

But, Java enables high performance with the use of just-in-time compiler.

=====================================================================================

Java Editions

1. Java Standard Edition

Java Standard edition is a computing platform which is used for development

and deployment of portable code that is used in desktop and server environments.

Java Standard Edition is also known as Java 2 Platform, Standard Edition (J2SE).

Java Standard Edition has a wide range of APIs such as Java Class Library etc.

the best implementation of Java SE is Oracle Corporation’s Java Development Kit (JDK)

2. Java Micro Edition

Java Micro Edition is a computing platform which is used for the development and

deployment of portable codes for the embedded and mobile devices.

Java Micro Edition is also known as Java 2 Platform Micro Edition (J2ME).

The Java Micro Edition was designed by Sun Microsystems and then later on Oracle

corporation acquired it in 2010.

Example: micro-controllers, sensors, gateways, mobile phones, printers etc

3. Java Enterprise Edition

Java Enterprise Edition is a set of specifications and extending Java SE 8 with features such as

distributed computing and web services. The applications of Java Enterprise Edition run on

reference runtimes. This reference runtime handle transactions, security, scalability,

concurrency and the management of components to be deployed. Java Enterprise Edition is also

known as Java 2 Platform Enterprise Edition (J2EE), and currently,

it has been rebranded as Jakarta EE.

Example: e-commerce, accounting, banking information systems.

4. JavaFX

JavaFX is used for creating desktop applications and also rich internet applications(RIAs)

which can be run on a wide variety of devices. JavaFX has almost replaced Swing as the

standard GUI library for Java Standard Edition.

JavaFX support for desktop computers and web browsers.

=====================================================================================

## Java variables

static variables:

A static variable is common to all the instances (or objects) of the class because

it is a class level variable. In other words you can say that only a single copy of4

static variable is created and shared among all the instances of the class.

Memory allocation for such variables only happens once when the class is loaded in the memory.

Example 2: Static Variable can be accessed directly in a static method

class JavaExample{

static int age;

static String name;

//This is a Static Method

static void disp(){

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

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

}

// This is also a static method

public static void main(String args[])

{

age = 30;

name = "Steve";

disp();

}

}

Output:

Age is: 30

Name is: Steve

======================================================================================================

Static variable initialization

Static variables are initialized when class is loaded.

Static variables are initialized before any object of that class is created.

Static variables are initialized before any static method of the class executes.

Default values for static and non-static variables are same.

primitive integers(long, short etc): 0

primitive floating points(float, double): 0.0

boolean: false

object references: null

==========================================================================================================

Static final variables

The static final variables are constants.

public class MyClass{

public static final int MY\_VAR=27;

}

1) The above code will execute as soon as the class MyClass is loaded,

before static method is called and even before any static variable can be accessed.

2) The variable MY\_VAR is public which means any class can use it.

It is a static variable so you won’t need any object of class in order to access it.

It’s final so the value of this variable can never be changed in the current or in any class.

Key points:

final variable always needs initialization, if you don’t initialize it would throw a compilation error. have a look at below example-

public class MyClass{

public static final int MY\_VAR;

}

Error: variable MY\_VAR might not have been initialize

## Access Modifiers

An access modifier restricts the access of a class, constructor, data member and method in another class.

Java language has four access modifier to control access level for classes and its members.

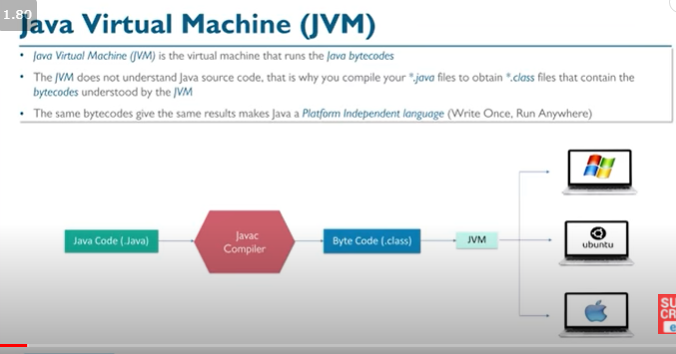
**Default: Default has scope only inside the same package**

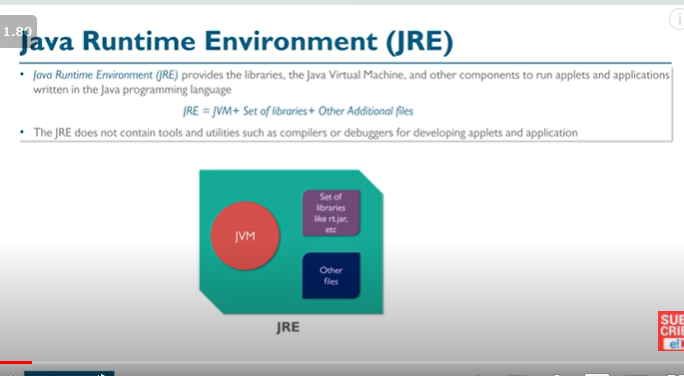
**Public: Public has scope that is visible everywhere**

**Protected: Protected has scope within the package and all sub classes**

**Private: Private has scope only within the classes**

## Jvm vs jdk vs jre





JVM:

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment

to execute java byte code. The JVM doesn't understand Java typo, that's why you

compile your \*.java files to obtain \*.class files that contain the bytecodes

understandable by the JVM.

JVM control execution of every Java program. It enables features such as automated exception

handling, Garbage-collected heap.

JRE :

The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine,

and other components to run applets and applications written in the Java programming language.

JRE does not contain tools and utilities such as compilers or debuggers for developing applets

and applications

JRE = JVM + libraries [ NO compilers and debuggers]

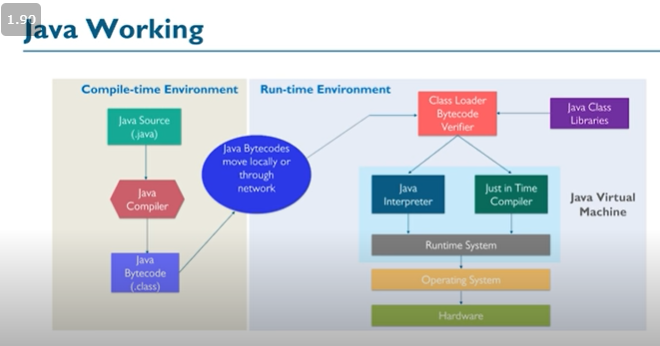
JDK :

The JDK also called Java Development Kit is a superset of the JRE,

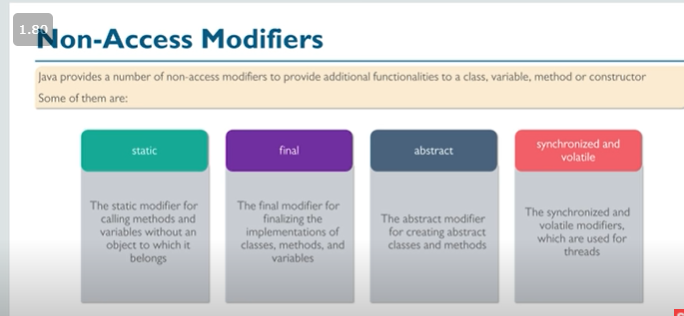
and contains everything that is in the JRE, plus tools

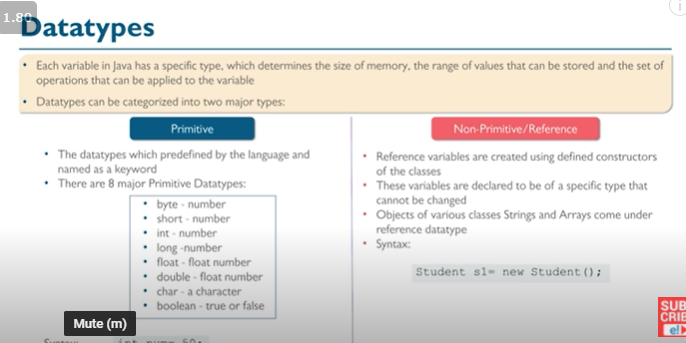
such as the compilers and debuggers necessary for developing applets and applications.

JDK = JRE + compilers + debuggers

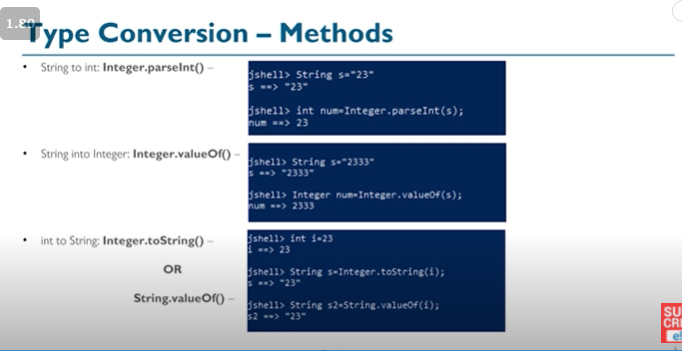


## NON ACCESS MODIFIERS

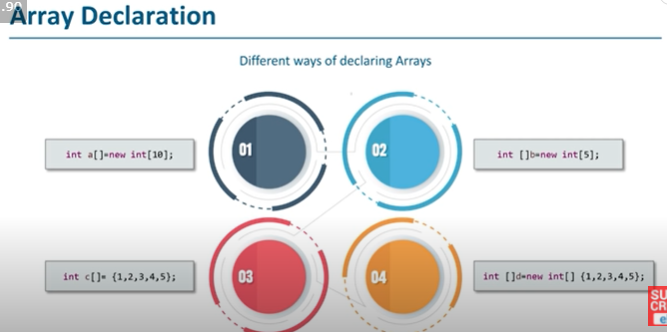




## TYPE CONVERSIONS



## ARRAY DECLARATION



# garbage collection

Java garbage collection is the process of releasing unused memory occupied by unused objects.

This process is done by the JVM automatically because it is essential for memory management.

When a Java programs run on the JVM, objects are created on the heap, which is a portion of

memory dedicated to the program. Eventually, some objects will no longer be needed.

When there is no reference to an object, then that object is assumed to be no longer needed

and the memory occupied by the object are released. This technique is called Garbage Collection.

Can the Garbage Collection be forced explicitly ?

No, the Garbage Collection can not be forced explicitly.

We may request JVM for garbage collection by calling System.gc() method.

But This does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection

Programmer doesn't need to worry about dereferencing an object.

It is done automatically by JVM.

Increases memory efficiency and decreases the chances for memory leak

finalize() method

Sometime an object will need to perform some specific task before

it is destroyed such as closing an open connection or releasing any resources held.

To handle such situation finalize() method is used.

The finalize() method is called by garbage collection thread before collecting object.

Its the last chance for any object to perform cleanup utility

# Generics

<https://howtodoinjava.com/java/generics/complete-java-generics-tutorial/>

# JAVA IO

Scanner myObj = new Scanner(System.in); // Create a Scanner object

nextBoolean() boolean value

nextByte() byte value

nextDouble() double value

nextFloat() float value

nextInt() int value

nextLine() String value

nextLong() long value

nextShort() short value

next() Method: The next() method in java is present in the Scanner class and is used to get

the input from the user. In order to use this method, a Scanner object needs to be created.

This method can read the input only until a space(” “) is encountered. In other words,

it finds and returns the next complete token from the scanner.

nextLine() Method:

The nextLine() method in java is present in the Scanner class and

is used to get the input from the user. In order to use this method, a Scanner object needs

to be created. This method can read the input till the end of line. In other words, it can take

input until the line change or new line and ends input of getting ‘\n’ or press enter.

# Java MULTITHREADING

To create a thread, Java provides a class Thread and an interface Runnable both are

located into java.lang package.

Life cycle of a Thread

New : A thread begins its life cycle in the new state.

It remains in this state until the start() method is called on it.

Runnable : After invocation of start() method on new thread, the thread becomes runnable.

Running : A thread is in running state if the thread scheduler has selected it.

Waiting : A thread is in waiting state if it waits for another thread to perform a task.

In this stage the thread is still alive.

Terminated : A thread enter the terminated state when it complete its task

Creating a thread in Java

To implement multithreading, Java defines two ways by which a thread can be created.

By implementing the Runnable interface.

By extending the Thread class.

class MyThread implements Runnable

{

public void run()

{

System.out.println("concurrent thread started running..");

}

}

class MyThreadDemo

{

public static void main(String args[])

{

MyThread mt = new MyThread();

Thread t = new Thread(mt);

t.start();

}

}

Extending Thread class

class MyThread extends Thread

{

public void run()

{

System.out.println("concurrent thread started running..");

}

}

classMyThreadDemo

{

public static void main(String args[])

{

MyThread mt = new MyThread();

mt.start();

}

}

## Java Synchronization

Synchronization is a process of handling resource accessibility by multiple thread requests.

The main purpose of synchronization is to avoid thread interference. At times when more than

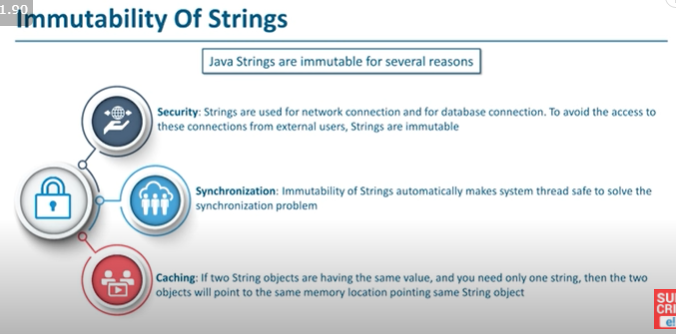
one thread try to access a shared resource, we need to ensure that resource will be used by

only one thread at a time. The process by which this is achieved is called synchronization.

The synchronization keyword in java creates a block of code referred to as critical section.

# JAVA STRINGS

## STRINGS



char charAt(int index) returns char value for the particular index

int length() returns string length

String substring(int beginIndex) returns substring for given begin index.

boolean contains(CharSequence s) returns true or false after matching the sequence of char value.

boolean equals(Object another) checks the equality of string with the given object.

boolean isEmpty() checks if string is empty.

String concat(String str) concatenates the specified string

String replace(char old, char new) replaces all occurrences of the specified char value.

String replace(CharSequence old, CharSequence new) replaces all occurrences of the specified CharSequence

String[] split(String regex) returns a split string matching regex.

int indexOf(int ch) returns the specified char value index.

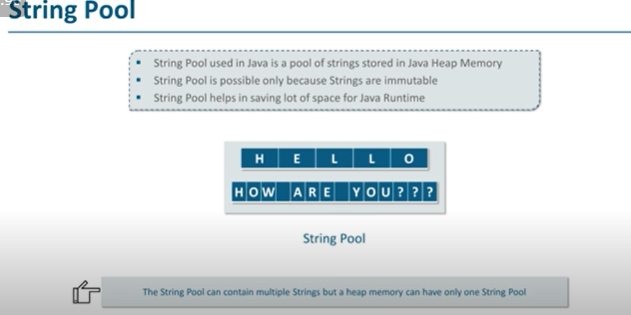
int indexOf(int ch, int fromIndex) returns the specified char value index starting with given index.

int indexOf(String substring) returns the specified substring index.

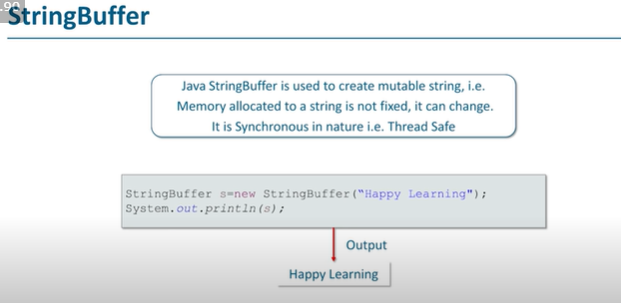
int indexOf(String substring, int fromIndex) returns the specified substring index starting with given index.

String trim() removes beginning and ending spaces of this string.

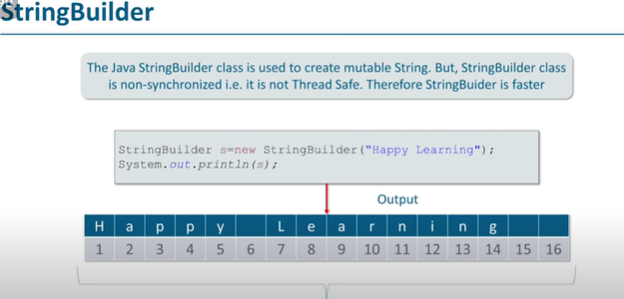
static String valueOf(int value) converts given type into string. It is an overloaded method.



## STRING BUFFER



## STRING BUILDER



Each time we create a String literal, the JVM checks the string pool first.

If the string literal already exists in the pool, a reference to the pool instance is

returned. If string does not exist in the pool, a new string object is created, and is

placed in the pool. String objects are stored in a special memory area known as string

pool inside the heap memory.

String Comparison

Using equals() method

Using == operator

By CompareTo() method

Using == operator

The double equal (==) operator compares two object references to check whether they

refer to same instance.

This also, will return true on successful match else returns false.

public class Demo{

public static void main(String[] args) {

String s1 = "Java";

String s2 = "Java";

String s3 = new String ("Java");

boolean b = (s1 == s2); //true

System.out.println(b);

b = (s1 == s3); //false

System.out.println(b);

}

}

By compareTo() method

String compareTo() method compares values and returns an integer value which tells if

the string compared is less than, equal to or greater than the other string.

It compares the String based on natural ordering i.e alphabetically. Its general syntax is.

Java String class functions

charAt() method

String charAt() function returns the character located at the specified index.

indexOf() method

String indexOf() method returns the index of first occurrence of a substring or

a character

length() method

String length() function returns the number of characters in a String.

substring() method

String substring() method returns a part of the string. substring() method has two override methods.

1. public String substring(int begin);

2. public String substring(int begin, int end);

contains()Method

String contains() method is used to check the sequence of characters in the

given string.

It returns true if a sequence of string is found else it returns false.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

StringBuffer class in Java

StringBuffer class is used to create a mutable string object.

class Test {

public static void main(String args[])

{

String str = "study";

str.concat("tonight");

System.out.println(str); // Output: study

StringBuffer strB = new StringBuffer("study");

strB.append("tonight");

System.out.println(strB); // Output: studytonight

}

}

insert()

This method inserts one string into another. Here are few forms of insert() method.

StringBuffer insert(int index, String str)

StringBuffer insert(int index, int num)

StringBuffer insert(int index, Object obj)

reverse()

This method reverses the characters within a StringBuffer object.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java StringBuilder class

StringBuilder is identical to StringBuffer except for one important difference that

it is not synchronized, which means it is not thread safe.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java String tokenizer

2. StringTokenizer(String str, String delim)

3. StringTokenizer(String str, String delim, booleanreturnValue)

Following are the methods in string tokenizer

1. booleanhasMoreTokens()

2. String nextToken()

3. String nextToken(String delim)

4. booleanhasMoreElements()

5. Object nextElement()

6. intcountTokens()

import java.util.\*;

public class TokenDemo2{

public static void main(String args[])

{

String a= " : ";

String b= "Welcome : to : studytonight : . : How : are : You : ?";

StringTokenizer c = new StringTokenizer(b, a);

int count1 = c.countTokens();

for (inti = 0; i<count1; i++)

System.out.println("token [" + i + "] : "

+ c.nextToken());

StringTokenizer d= null;

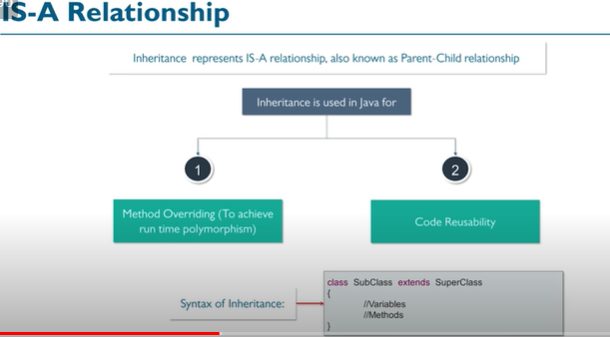
while (c.hasMoreTokens())

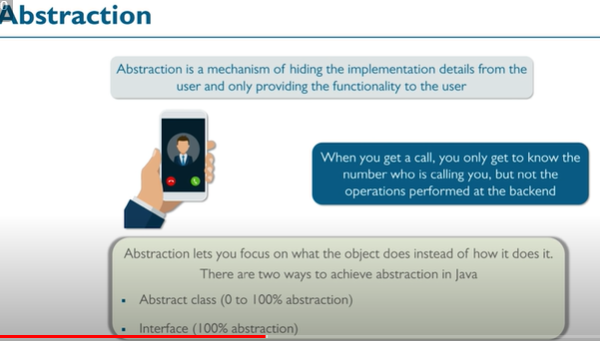
System.out.println(d.nextToken());

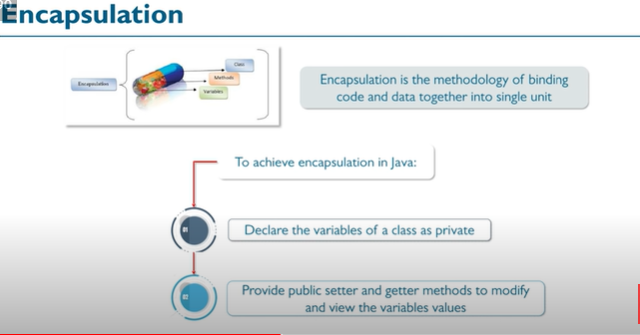
}

}

# OOPS







What is OOPS?

OOPS is a programming approach which provides

solution to real life problems with the help of algorithms based on real world.

Main Features of OOPS

Inheritence

Polymorphism

Encapsulation

Abstraction

Inheritance provided mechanism that allowed a class to inherit property of another class

Purpose of Inheritance

It promotes the code reusabilty i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.

It promotes polymorphism by allowing method overriding.

Types of Inheritance

Java mainly supports only three types of inheritance that are listed below.

1) Single Inheritance

2) Multilevel Inheritance

3) Heirarchical Inheritance

NOTE: Multiple inheritance is not supported in java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Why multiple inheritance is not supported in Java?

To remove ambiguity.

To provide more maintainable and clear design

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Aggregation:

In Java, aggregation represents HAS-A relationship, which means when a class contains

reference of another class known to have aggregation

Q. When to use Inheritance and Aggregation?

When you want to use some property or behaviour of any class without the requirement of

modifying it or adding more functionality to it, in such case Aggregation is a better option

because in case of Aggregation we are just using any external class inside our class as a

variable.

Whereas when you want to use and modify some property or behaviour of any external class or

may be want to add more function on top of it, its best to use Inheritance.

Composition in Java

Composition is a more restricted form of Aggregation. Composition can be described

as when one class which includes another class, is dependent on it in such a way

that it cannot functionally exist without the class which is included.

For example a class Car cannot exist without Engine, as it won't be functional anymore.

Hence the word Composition which means the items something is made of and if we change the composition of things they change, similarly in Java classes, one class including another class is called a composition if the class included provides core functional meaning to the outer class.

Q)Can we Override static method? Explain with reasons?

No, we cannot override static method. Because static method is bound to class whereas

method overriding is associated with object i.e at runtime.

Upcasting in Java

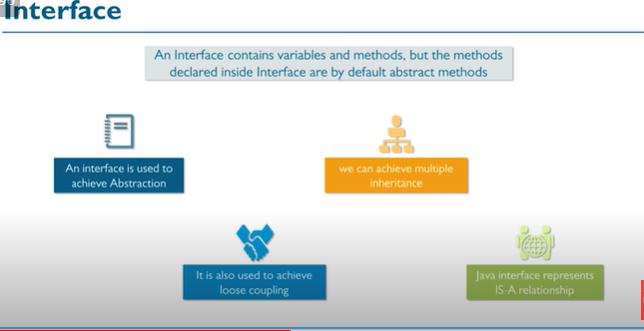
When Parent class reference variable refers to Child class object, it is known as Upcasting.

In Java this can be done and is helpful in scenarios where multiple child classes extends one

parent class.

In those cases we can create a parent class reference and assign child class objects to it.

# Interfaces



It can have only abstract methods and static fields.

However, from Java 8, interface can have default and static methods and from Java 9,

it can have private methods as well.

Interface is a blueprint of an class and used to achieve abstraction in Java.

Interface contains abstract methods and default, private methods. We can not create

object of the interface.

Interface can be used to implement multiple inheritance in Java.

Advantages of Interface

It Support multiple inheritance

It helps to achieve abstraction

It can be used to achieve loose coupling.

Interface Key Points

Methods inside interface must not be static, final, native or strictfp.

All variables declared inside interface are implicitly public, static and final.

All methods declared inside interfaces are implicitly public and abstract,

even if you don't use public or abstract keyword.

Interface can extend one or more other interface.

Interface cannot implement a class.

Interface can be nested inside another interface.

Static methods in Interface – Java 8

From Java 8, Java allows to declare static methods into interface.

The purpose of static method is to add utility methods into the interface

**Private methods – Java 9**

In Java 9 version, a new feature is added that allows us to declare private methods inside the interface. The purpose of private method is just to share some task between the non-abstract methods

of the interface.

In this example, we created an interface Abc that has a default method and a

private method as well. Since private methods are not accessible outside

to interface. So, we called it from the default method.

interface Abc{

// Default method

default void msg(){

greet();

}

// Private method

private void greet() {

System.out.println("This is private method");

}

}

public class Demo implements Abc{

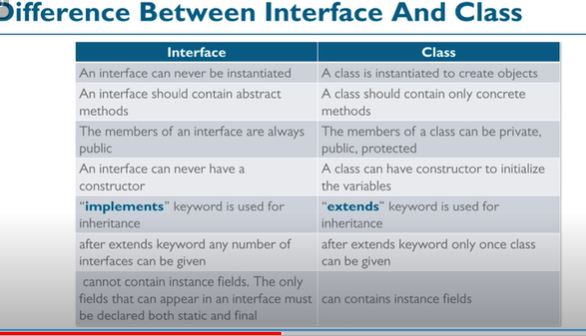
public static void main(String[] args) {

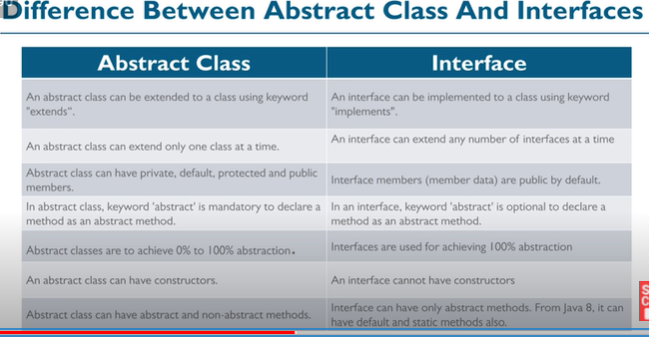
Demo d = new Demo();

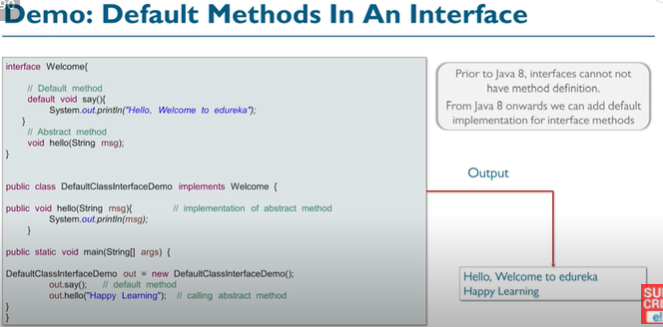
d.msg(); // calling default method

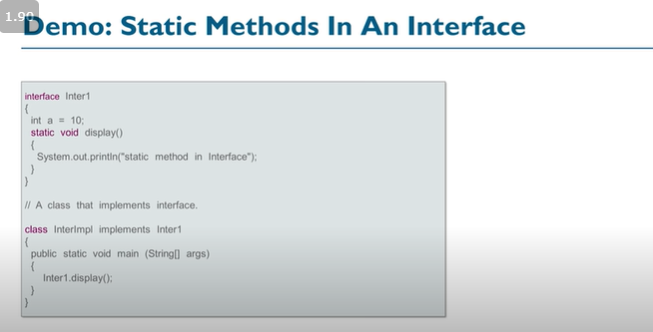
}

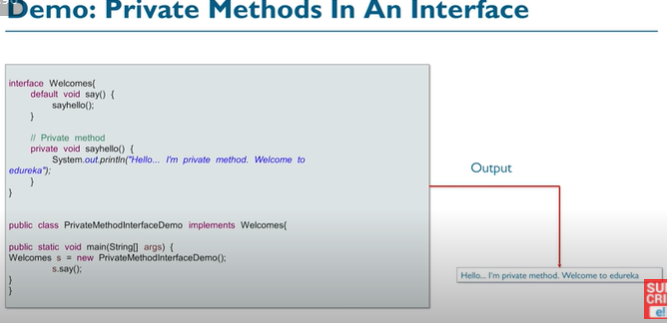
}

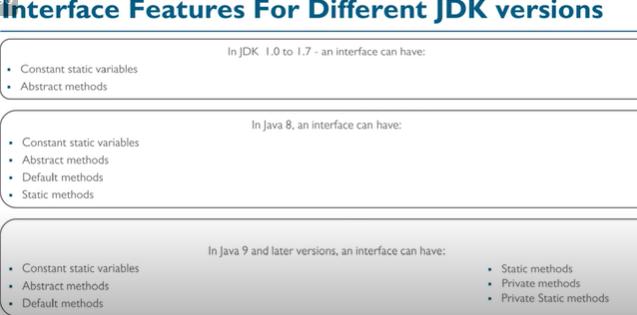


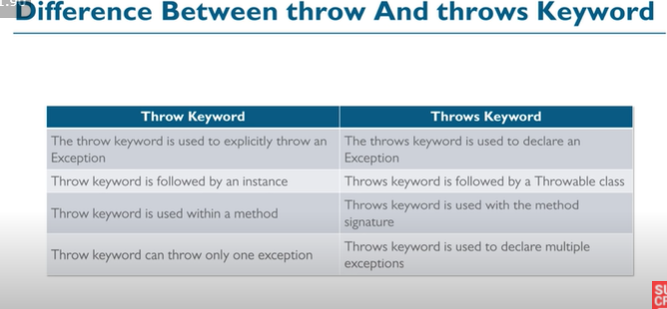










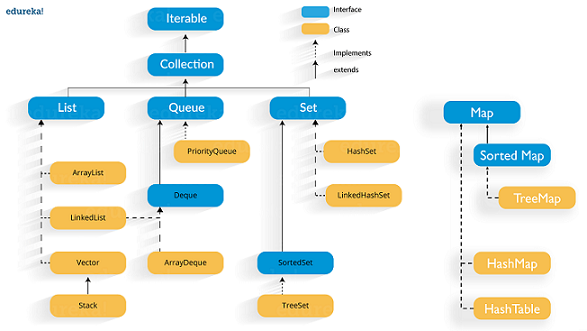


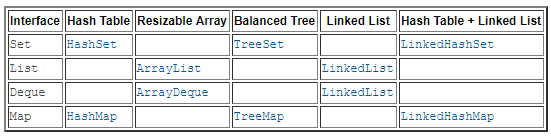
# JAVA PACKAGES

Package is a collection of related classes.

Java uses package to group related classes, interfaces and sub-packages in any Java project.

# COLLECTIONS Framework





**Interfaces**

**List**

**Queue**

**Set**

**==================No Map==========================================**

### METHODS IN COLLECTION INTERFACE

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object element)

boolean removeAll(Collection<?> c)

boolean removeIf(Predicate<? super E> filter)

boolean retainAll(Collection<?> c)

int size() -----------------------------------------------------

void clear() --------------------------------------------------------

boolean contains(Object element) -------------------------------

boolean containsAll(Collection<?> c)

Iterator iterator()

Object[] toArray() It converts collection into array.

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.

boolean isEmpty() It checks if collection is empty.

default Stream<E> stream() It returns a sequential Stream with the collection as its source.

public boolean equals(Object element) It matches two collections.

=====================================================================================

## Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No. Method Description

1 public boolean hasNext() It returns true if the iterator has more elements otherwise it returns false.

2 public Object next() It returns the element and moves the cursor pointer to the next element.

3 public void remove() It removes the last elements returned by the iterator. It is less used.

=====================================================

## List Interface

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

List <data-type> list1= new ArrayList();

List <data-type> list2 = new LinkedList();

List <data-type> list3 = new Vector();

List <data-type> list4 = new Stack();

ArrayList

E get(int index) fetch the element from the particular position of the list.

boolean contains(Object o) It returns true if the list contains the specified element

E remove(int index) remove the element present at the specified position in the list.

boolean remove(Object o) remove the first occurrence of the specified element.

void sort(Comparator<? super E> c) sort the elements of the list on the basis of specified comparator.

E set(int index, E element) replace the specified element in the list, present at the specified position.

=====================================================================================

## Queue Interface

Queue interface can be instantiated as:

Queue<String> q1 = new PriorityQueue();

Queue<String> q2 = new ArrayDeque();

Object poll()

retrieves and removes the head of this queue, or returns null if this queue is empty.

Object element() retrieves, but does not remove, the head of this queue.

Object peek() retrieves,but does not remove, the head of this queue, or returns null if this queue is empty.

Object remove() retrieves and removes the head of this queue

===================

## Java Deque Interface

ArrayDeque class

=========================================================================

## Set Interface

Set<data-type> s1 = new HashSet<data-type>();

Set<data-type> s2 = new LinkedHashSet<data-type>();

Set<data-type> s3 = new TreeSet<data-type>();

============================================================================

## Java HashSet

HashSet contains unique elements only.

HashSet allows null value.

HashSet class is non synchronized.

HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.

HashSet is the best approach for search operations

constructor has collection parameter bro...

HashSet(Collection<? extends E> c) Init using the elements of the collection c.

=====================================================================

## Java TreeSet class

Java TreeSet class contains unique elements only like HashSet.

Java TreeSet class access and retrieval times are quiet fast.

Java TreeSet class doesn't allow null element.

Java TreeSet class is non synchronized.

Java TreeSet class maintains ascending order.

TreeSet(Collection<? extends E> c) The elements of the collection c.

TreeSet(Comparator<? super E> comparator) construct an empty tree set that will be sorted according to given comparator.

TreeSet(SortedSet<E> s) TreeSet that contains the elements of the given SortedSet.

E higher(E e)

It returns the closest greatest element of the specified element from the set,

or null there is no such element.

Iterator iterator() iterate the elements in ascending order.

E lower(E e) It returns the closest least element of the specified element from the set,

or null there is no such element.

E first() It returns the first (lowest) element currently in this sorted set.

E last() It returns the last (highest) element currently in this sorted set.

Iterator descendingIterator() It is used iterate the elements in descending order.

E floor(E e)

It returns the equal or closest least element of the specified element

from the set, or null there is no such element.

E ceiling(E e)

It returns the equal or closest greatest element of the specified element from the set,

or null there is no such element.

SortedSet tailSet(E fromElement)

It returns a set of elements that are greater than or equal to the specified element.

SortedSet headSet(E toElement)

It returns the group of elements that are less than the specified element.

NavigableSet descendingSet() It returns the elements in reverse order.

SortedSet subSet(E fromElement, E toElement))

It returns a set of elements that lie between the given range which

includes fromElement and excludes toElement.

=====================================================================================

## Map interface

There are two interfaces for implementing Map in java:

Map and SortedMap,

and three classes:

HashMap, LinkedHashMap, and TreeMap.

V put(Object key, Object value) insert an entry in the map.

void putAll(Map map) 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) 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.

boolean containsValue(Object value) This method returns true if some value equal to the value exists within the map, else return false.

boolean containsKey(Object key)

This method returns true if some key equal to the

key exists within the map, else return false.

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.

boolean isEmpty() This method returns true if the map is empty; returns false if it contains at least one key.

V replace(K key, V value) It replaces the specified value for a specified key.

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.

K getKey() It is used to obtain a key.

V getValue() It is used to obtain value.

V setValue(V value) I

t is used to replace the value corresponding to this entry with the specified value.

/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());

}

}

}

====================================================================

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());

}

}

}

**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);

}

}

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);

}

}

===========================================================================

## Java Stack

Method Modifier and Type Method Description

empty() boolean The method checks the stack is empty or not.

push(E item) E The method pushes (insert) an element onto the top of the stack.

pop() E The method removes an element from the top of the stack and returns the same element as the value of that function.

peek() E The method looks at the top element of the stack without removing it.

search(Object o) int The method searches the specified object and returns the position of the object.

# JAVA ADDITIONAL INFO (programming)

## FINAL KEYWORD

final variable -----------------------To create constant values

final methods------------------preventing method overriding

final class ---------------- prevent inheritance

===========================================================================================

Initializing a final variable

Once you declare a final variable,

it is a must to initialize it.

You can initialize the final instance variable −

At the time of declaration as.

public final String name = "Raju";

public final int age = 20;

Within an instance (non-static) block.

{

this.name = "Raju";

this.age = 20;

}

Within a default constructor.

public final String name;

public final int age;

public Student(){

this.name = "Raju";

this.age = 20;

}

=========================================

// a final variable

final int THRESHOLD = 5;

// a blank final variable

final int THRESHOLD;

// a final static variable PI

static final double PI = 3.141592653589793;

// a blank final static variable

static final double PI;

Initializing a final variable :

We must initialize a final variable, otherwise compiler will throw compile-time error.

A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable :

You can initialize a final variable when it is declared.This approach is the most common.

A final variable is called blank final variable,if it is not initialized while declaration.

Below are the two ways to initialize a blank final variable.

A blank final variable can be initialized inside instance-initializer block

or inside constructor. If you have more than one constructor in your class then

it must be initialized in all of them, otherwise compile time error will be thrown.

A blank final static variable can be initialized inside static block

================================================================================================

final variable

The final keyword, when applied to fields of a Java class, has nothing to do with inheritance.

Instead, it indicates that outside of the constructor, that field cannot be reassigned

=======================================================================================

final method:

When a method is declared as final then it cannot be overridden by subclasses.

The Object class does this—a number of its methods are final.

The following fragment illustrates final keyword with a method:

class A

{

final void m1()

{

System.out.println("This is a final method.");

}

}

class B extends A

{

void m1()

{

// ERROR! Can't override.

System.out.println("Illegal!");

}

}

====================================================================

final class:

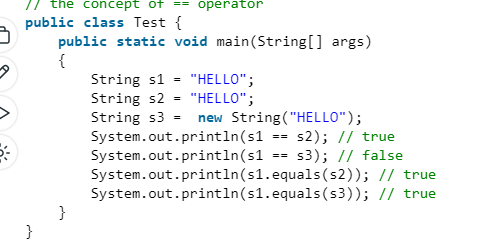
When a class is declared as final then it cannot be subclassed i.e.

no any other class can extend it. This is particularly useful,

for example, when creating an immutable class like the predefined String class.

## JAVA == vs equals

1. The main difference between the .equals() method and == operator is that one is a method and the other is the operator.
2. We can use == operators for reference comparison (**address comparison**) and .equals() method for **content comparison**. In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.



## STATIC BLOCK IN JAVA

A static block in a program is a set of statements which are executed by the JVM

(Java Virtual Machine) before the main method. At the time of class loading,

if we want to perform any task we can define that task inside the static block,

this task will be executed at the time of class loading. In a class, any number of a

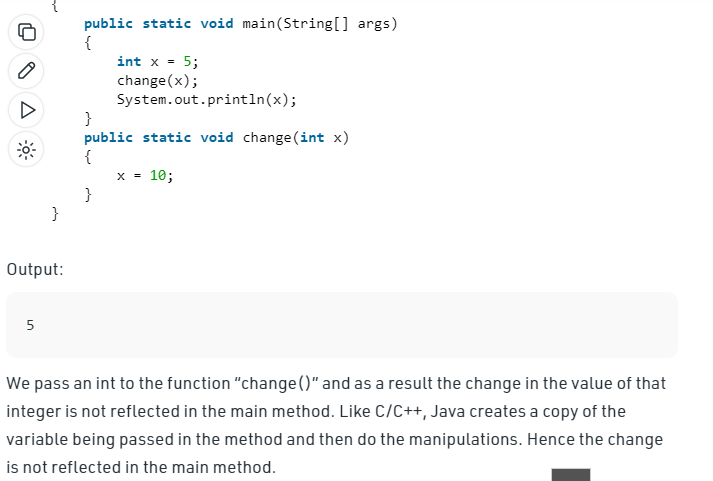
static block can be defined, and this static blocks will be executed from top to bottom

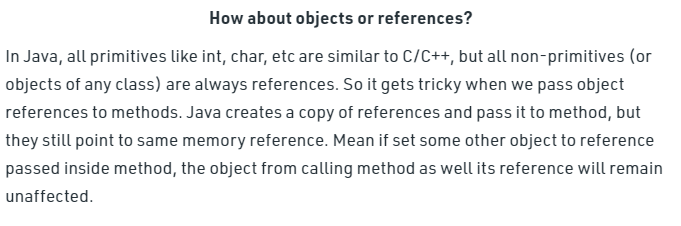
**static and initializer block**

We can have both static and initializer blocks in a Java program.

But static block will execute first even before initializer block.

## JAVA IS STRICTLY PASS BY VALUE





**The changes are not reflected back if we change the object itself to refer some other location or object.**

**Changes are reflected back if we do not assign reference to a new location or object:**  
If we do not change the reference to refer some other object (or memory location), we can make changes to the members and these changes are reflected back.

# Java8

## ANONYMOUS CLS

## ANONYMOUS CLASS

Without Lambda Expression (ANANYMOUS CLASS)

==========================================================

interface Drawable{

public void draw();

}

public class LambdaExpressionExample {

public static void main(String[] args) {

int width=10;

//without lambda, Drawable implementation using anonymous class

Drawable d=new Drawable(){

public void draw(){System.out.println("Drawing "+width);}

};

d.draw();

}

}

@FunctionalInterface //It is optional

interface Drawable{

public void draw();

}

public class LambdaExpressionExample2 {

public static void main(String[] args) {

int width=10;

//with lambda

Drawable d2=()->{

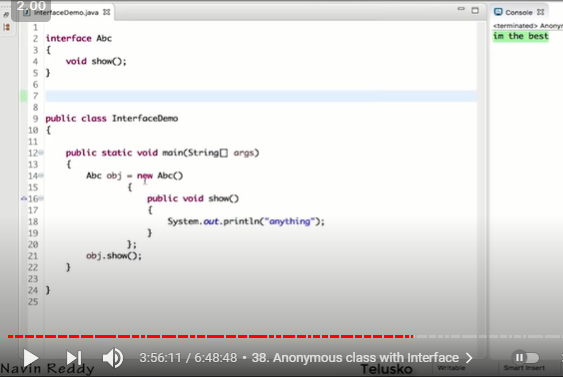
System.out.println("Drawing "+width);

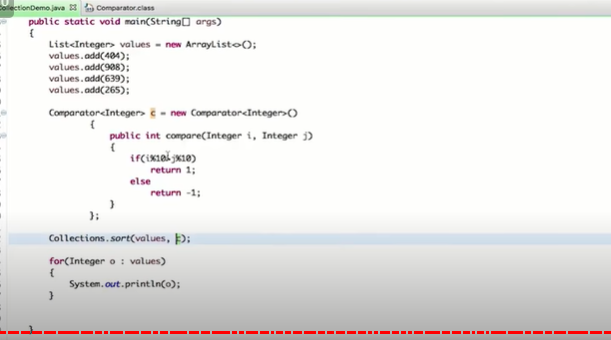
};

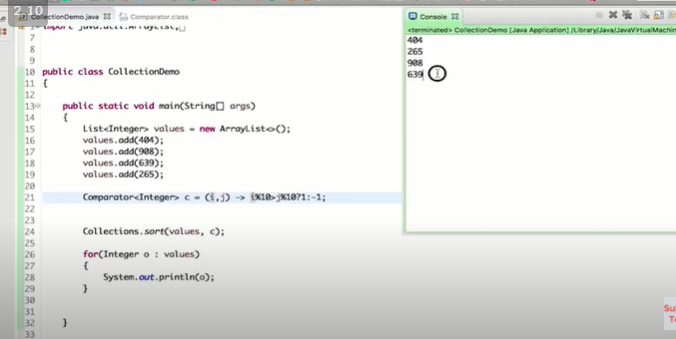
d2.draw(); // calling draw method

}

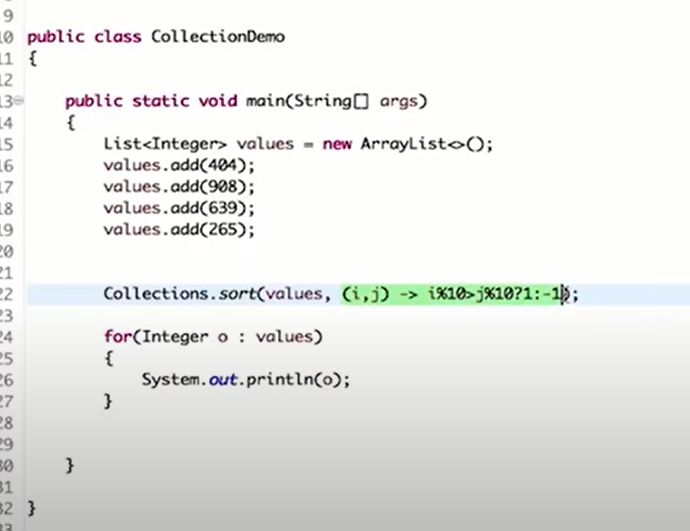
}

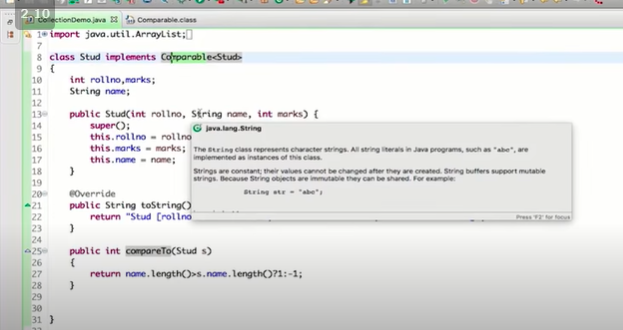






## LAMBDA Expression





The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the

method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Why use Lambda Expression

1)To provide the implementation of Functional interface.

2)Less coding.

Java Lambda Expression Example: Foreach Loop

import java.util.\*;

public class LambdaExpressionExample7{

public static void main(String[] args) {

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

list.add("ankit");

list.add("mayank");

list.add("irfan");

list.add("jai");

list.forEach(

(n)->System.out.println(n)

);

}

}

### SORTING using lambda

public class LambdaExpressionExample10{

public static void main(String[] args) {

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

list.add(new Product(2,"Dell Mouse",150f));

System.out.println("Sorting on the basis of name...");

// implementing lambda expression

Collections.sort(list,(p1,p2)->{

return p1.name.compareTo(p2.name);

});

for(Product p:list){

System.out.println(p.id+" "+p.name+" "+p.price);

}

}

}

## Functional Interface

Lambda expression provides implementation of functional interface. An interface which has only

one abstract method is called functional interface. Java provides an anotation

@FunctionalInterface, which is used to declare an interface as functional interface.

## COMPARABLE INTERFACE

As the name suggests, Comparable is an interface defining a strategy of comparing an object with

other objects of the same type. This is called the class's “natural ordering”.

Accordingly, in order to be able to sort – we must define our Player object as comparable by

* **implementing the Comparable interface:**

public class Player implements Comparable<Player> {

@Override

public int compareTo(Player otherPlayer) {

return Integer.compare(getRanking(), otherPlayer.getRanking());

}

}

## COMPARATOR INTERFACE

The Comparator interface defines a compare(arg1, arg2) method with two arguments that represent

compared objects and works similarly to the Comparable.compareTo() method

## Comparator vs Comparable

=============================

The Comparable interface is a good choice when used for defining the default ordering or, in other words, if it's the main way of comparing objects.

Then, we must ask ourselves why use a Comparator if we already have Comparable?

There are several reasons why:

Sometimes, we can't modify the source code of the class whose objects we want to ort, thus making the use of Comparable impossible

Using Comparators allows us to avoid adding additional code to our domain classes

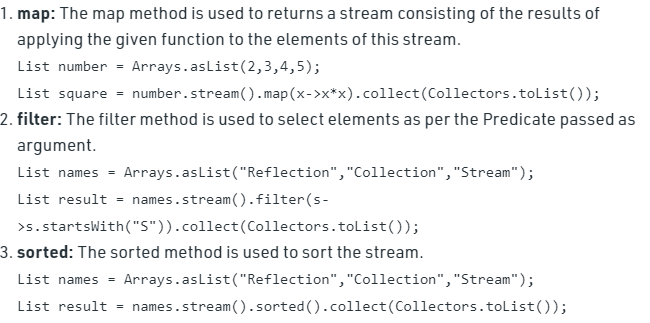
We can define multiple different comparison strategies which isn't possible when using Comparable

To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas

if you sorting needs to be done on attributes of different objects, then use Comparator in Java.

## JAVA STREAMS

A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result



A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.\

min:

List<Integer> list = Arrays.asList(-9, -18, 0, 25, 4);

// Using stream.min() to get minimum

// element according to provided Integer Comparator

Integer var = list.stream().min(Integer::compare).get();

Integer minnum = Stream.of(10, 13, 4, 9, 2, 100)

.min( new Comparator<Integer>() {

public int compare(Integer i1,Integer i2)

{

return i1.compareTo(i2);

}

}

)

.get();

Integer minnum = Stream.of(10, 13, 4, 9, 2, 100)

.min((i1, i2)->

{

return i1.compareTo(i2);

}

)

.get();

String maxFirstChar = Stream.of("Aryan", "Tom", "Harry", "Steve")

.max(Comparator.comparing(String::valueOf))

.get();

String minFirstChar = Stream.of("Aryan", "Tom", "Harry", "Steve")

.min((a,b) ->{

return a.compareTo(b);

})

.get();

distinct()

Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.

peek:

Returns a stream consisting of the elements of this stream, additionally performing

the provided action on each element as elements are consumed from the resulting stream

Stream.of("one", "two", "three", "four")

.filter(e -> e.length() > 3)

.peek(e -> System.out.println("Filtered value: " + e))

.map(String::toUpperCase)

.peek(e -> System.out.println("Mapped value: " + e))

.collect(Collectors.toList());

collect

-----------------------------------------------------------------------------

### Terminal Operations:

------------------------------------------------------------------------------

collect:

The collect method is used to return the result of the intermediate operations performed on the stream.

List number = Arrays.asList(2,3,4,5,3);

Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());

forEach:

The forEach method is used to iterate through every element of the stream.

List number = Arrays.asList(2,3,4,5);

number.stream().map(x->x\*x).forEach(y->System.out.println(y));

reduce:

Parameters:

identity - the identity value for the accumulating function

accumulator - an associative, non-interfering, stateless function for combining two values

The reduce method is used to reduce the elements of a stream to a single value.

The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

count()

Long empCount = empList.stream()

.filter(e -> e.getSalary() > 200000)

.count();

sum()

Summing a stream of numbers can be expressed as:

Integer sum = integers.reduce(0, (a, b) -> a+b)

(or)

Integer sum = integers.reduce(0, Integer::sum);