**JAVA**

* **New keyword -**   
  1. Send request to class for creating object.  
  2. Once object is created it will store object address into reference variable.  
  3. Mandatory to call constructor.
* **Variables -**   
  **1. Local Variable** – This variable created inside method and it can be accessible only in that method only.  
  - Need to initialize local variable else it will be giving error while accessing it.  
    
  **2. Non – static variable** – Created Outside method and inside class without static keyword.  
  - No need to initialize it will get by default value depends on data type.  
  - Mandatorily need to create object to access non-static variables then only it will accessible.   
  - It is also called as instance variables. (Copy of non-static members loaded into object reference variable)  
    
  **3. Static variable** -   
  - created Outside method inside class.  
  - access globally anywhere inside program without creating object.  
  - class having its memory in which static members are getting store hence no need to create object. Hence called as class members.  
  - Three ways to access static members   
  1. System.out.println(x); // direct variable name  
  2. System.out.println(A.x); // Reference to class Name  
  A a1 = new A();  
  3. System.out.println(a1.x); // creating obj and then access but during compilation it will convert A.x   
    
    
  **4. Reference variable -**   
  - A a1 = new A(); here a1 is reference variable that holds object address of respective class.  
  - reference variable can also hold null value. By default, reference variable having null value.  
  - static A a1 ; // we can access this reference variable anywhere in entire program.
* **Heap memory & stack memory**  
  - Heap memory contains all the objects   
  - stack memory maintains program execution flow. stack memory contains class members and references.  
  - flow always starts from main() method.
* **Methods**   
  **-** Non void that returns values like String, int, float, Boolean etc.  
  - void not returning any value  
  - need to call method else it will not run anymore.  
  - String a1 = method();  
  - public String method(){  
    
  return "hello world";  
  }  
  - method return value automatically store inside reference without calling a method just need to print object.
* **Return and Return value**  
  - return can be used into void method and it is optional to use. It just controls back to calling statement.  
  - return value is mandatory to use in non-void method it is returning value and control back to calling statement where it is getting called.
* **Constructor**   
  - Same name as class name, do not create with data type and void because treated as method  
  - As it is always void can use return keyword inside it.  
  - When object created new keyword call constructor automatically.  
  - Two types  
  1. **Default constructor** – created automatically when we compile java program.

2. **Parameter constructor** – created when explicitly provide value to object.

- **Constructor Overloading**   
 - Same name but different number of argument and same variable name but different data type this is called constructor overloading  
  
- **Constructor Chaning**  
- calling constructor from another constructor is called constructor changing. When we calling from another constructor it should be first statement while calling constructor  
- done by this() keyword

* **This keyword**  
  - This is special type of reference variable in java.  
  - object address is store into two places one is object reference variable and another is into this keyword.  
  - we can access non-static and also static variable most of non-static using this keyword.  
  - cannot use this keyword into static context.  
    
  **this()** - used to call constructor in constructor chaining.  
  - it should be first statement while calling another constructor.  
  - ensure that calling is done by another constructor only.
* **OOPS (Object Oriented Programming)**  
  **\* super most class in java is Object. Every class subclass of Object class.**  
    
  1. **Inheritance**   
  - Inheriting members of members of parent class to child class. Non-static probably.  
  - class to class – extends   
  - interface to class – implements  
  - types
* single (class B extends A)
* multilevel (class C extends B extends A )
* hybrid (class D extends B extends A as well as class D extends C extends A)
* hierarchical (class B extends class A and class C extends A)
* Multiple inheritance not support in java at inheritance level but support at interface level (class A, B extends C where C is child having both members of parent A and B)

- class A is parent class also called as Super class, parent class and B and C are the child class also called as subclass and child class.  
  
\* We can never inherit static members in java it just gives feel of inheritance but during compilation B.x gets converted to A.x.

* **Packages**  
  - Nothing but folders to store java files in organize manner.  
  - name should be in lower case.  
  - cannot contains reserve keyword.  
  - cannot write any code above package keyword.  
  - we can also create folder within folder like p1.p2.p3  
    
  - Referring class present inside different package then import becomes mandatory   
  package p1 contains class A and p2 contains class B and I need to do inheritance then class B extends A. Here I am getting error to A because A is in package p1 so need to import it import p1.A;   
    
  \* We can add same name class multiple classes in same project but need to keep in different package and while creating object explicitly do p1.A a1 = new A(); or P2.A a2= new A();  
    
  \* Import all the classes from particular package then use regular expression \*  
    
    
  \* There are two types sub class and non-subclass  
  **subclass:** means **inheritance** in happening class B extends A and creating object of child class B b1 = new B(); and accessing members of class A. This is called as subclass.  
  **non – sub class:** means without inheritance. extends A. In child class we are creating object of parent class directly class B { A a1 = new A(); } and we are directly access members of parent class this is called as non-sub class.

**Access specifiers / Access modifiers on members**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | private | default | protected | public |
| Same class | yes | yes | yes | yes |
| Same package sub class | no | yes | yes | yes |
| Same package non-sub class | no | yes | yes | yes |
| Different package sub-class | no | no | yes | yes |
| Different package non-sub class | no | no | no | yes |

**Private** members can be accessible only in same class

**default** members can be accessible in same class and same package with or without inheritance.

**protected** members access within same class same package and outside package only via inheritance.

**public** can be access in same class, same package, different package via inheritance and without we can access.

**Access specifier on class**

class cannot be private or protected it just default and public

public class Access anywhere in any package with inheritance only.

default can be access via inheritance inside same package only outside package we cannot access.

* **Access specifier on constructor**   
  - **private** – can create object inside same class only  
  - **default** and **protected** can create object in same class and same package only.  
  - **public** can be created in same and different package only.
* **Polymorphism**  
  - Develop a feature such that it will take more than one form depending upon situation.  
  - Two type of polymorphism   
  1. **Overriding (Runtime)** - Inherit members of parent to child and modify logic in child class.  
  - same name and same signature need to be there.  
  - check overriding happens or not keep @Override annotation on the top of inherited method.  
  - @Override feature introduce in java version 5.  
  - Ex: Gold Account and Silver Account some methods and features are same but change in logic.  
    
  \* During Overriding we can we can Increase the scope of access specifier , but we cannot reduce it   
  private – less scope  
  default  
  protected  
  public – most scope  
    
  \* Private method cannot get inherit and override.  
    
    
  2. **Overloading (compile time)** -  
  - same method name in same class having different number of argument and different type of argument.   
  - If you want to calculate area of triangle, circle, square you cannot create different methods for it it's better to create a method Area and pass multiple type of argument into it.
* **Encapsulation**  
  - Wrapping data inside method and access that data using publicly defined getters and setters and let methods operate that data called as Encapsulation.  
  - Using non-static members access via methods which are public defined.  
  - Making members private and access using getters and setters.  
  - This class also called POJO (plain old java object).  
  - Here we are directly avoiding access of variable by making it private.  
  - private int id;  
  - public void setId(int id){  
   this.id = id;  
  }  
  - public int getId(){  
  return id;  
  }
* **Interface**  
  - contains incomplete methods. Also called as abstract methods  
  - used to hiding implementations details. Or to achieve good design.  
  - interface – interface –extends  
  - class to class – extends   
  - Interface to class - implements  
  - inheritance support multiple inheritance.
* **IIB (Instance Initialization block)**  
  - Used to initialize non-static and static variables.  
  - when object is created before constructor IIB will execute.  
  - class A{  
  int i;  
  {  
  i = 10;  
  syso(i);  
  }   
    
  main(){  
  new A();  
  }  
    
  \* Can crate object inside IIB but program halt.
* **SIB (Static Initialization Block)**  
  - just add static word before block.  
  - used to initialize static variables.  
  - call SIB before main method.  
  - we can create object inside SIB.  
  - cannot call non-static variables into SIB.  
  - static {  
  }

- Without creating obj SIB will call

* **Super keyword**  
  - For that Inheritance in mandatory.  
  - used to call parent class members. Static and non-static both.  
  - cannot used into static context.  
  - using super keyword, we can call parent class constructor but for that we need to keep super keyword into child class constructor. And it should be first statement else it will give you error.  
  - when no argument parent class constructor is there in child class constructor automatically no argument constructor and super keyword place automatically during compilation.  
  - Need to create explicitly super keyword into child class constructor for parameterized constructor in parent class because compiler will not place it automatically.
* **Data types**  
  - Integer (Byte, Short, Int, Long) - Memory size in byte 1248 – default value 0  
  - Decimal (float, Double) Memory size in byte 48 – default value – 0.0  
  char – 2 – default value empty space  
  Boolean – NA - false  
  String – Na – null  
    
  \* var data type   
  - Intro java –10  
  - store any kind of value.  
  - used as local variable.

**Collection framework**  
Collection store group of objects in it.  
  
  
  
  
**ArrayList -**   
- Dynamic array.  
- Initial size of Array List is 10. And increase by 1.5 times automatically when exceeds.  
- Maintain Insertion Order. retrieve also in same order  
- Best performance while reading element.  
- we can store objects into Array List.  
- we can store heterogeneous type of data in array list.  
  
cons :  
- contains duplicate element.  
- not synchronized means it is not thread safe. (Multiple threads allow access on object  
- cannot store primitive types such as int, char, float etc. required to use Wrapper classes such as Integer, Float, Character.  
- Insertion of data in between arraylist will result poor performance.  
- can store multiple null values.  
- does not follows sorting order.  
  
ArrayList arr = new ArrayList(); // store heterogeneous data  
ArrayList<Type> arr = new ArrayList<Type>(); // Type - Wrapper classes can store that kind of data only  
also called generics :  
Generics helps us to create variables to store particular type of data in it . Like data type.  
Generics applied on non-static variables.  
  
package generics;

public class A<T> {

T val;

A(T x){

this.val = x;

System.out.println(val);

}

public static void main(String[] args) {

A<String> a1 = new A("Yash");

A<Integer> a2 = new A();

A<Character> a3 = new A('A');

}

}

Yash

123

A

arr.add(10);

arr.add(new Integer(30)); // Boxing – Wrapping data inside object is called Boxing.  
  
- Adding value to perticular index number   
arr.add(1, 500);  
- arr.addAll(2, y); here y is second arrayList contains value are 500,600 It will add this value at position second.  
- arr.contains(value) // return true or false.  
- arr.remove(index); remove value based on index number.  
- arr.get(index); // get value from particular index.  
- Read arrayList using iterator  
Iterator iterator = arr.iiterator();  
while(iterator.hasNextInt()){  
syso(iterator.next());  
}  
  
Array – refers to continuous memory.  
  
  
Linked List (Join particular memory and we can store collection of data)  
- Two types   
- Singly and doubly (singly one dimensional, doubly two dimensional)  
- by default, linked list internally used doubly linked list.  
- Singly – Traveling data from head to last node in one direction.  
- contains head and tail .  
-   
- Next is an address of next node  
- last node called tail and its address section is null;  
  
  
- Doubly linked list   
- Data transverse trough bidirectional mapping.   
- node contains 3 part - previous node address - value - next node address  
- head previous node address is null  
  
- Upcasting - Object is created and its value is store in Parent Reference variable.  
List<Integer> x = new LinkedList<Integer>();  
class upcasting = A a1 = new B(); // storing child class object into parent class reference.  
class down casting = B b1 = (B) new A(); // storin parent class obj into child class reference.  
  
  
- ArrayList and LinkedList can consist of duplicate record.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Collections** - Group of elements, objects ,items etc . It can be homegenious or   
hetrogenous also.

collecction Interface having - Adding , removing , deleting and searching object methods are there   
collection interface extends Iterable interface having only one method itrator.

collection is group of elements

**List** collection is group of elements with indexing. They are in order and based on indexing we can add , remove , delete , alter element based on indexing.

This part is missing in **Collection** so we will get extra benifit in list.

List can have duplicate also but they are ordered

List - classes(ArrayList, LinkedList, Vector , stack)

Vector and stack are in java version 1.1 . They are used in order program so called as legacy classes.  
  
  
ArrayList<Integer> al1 = **new** ArrayList<>();

ArrayList<Integer> al2 = **new** ArrayList<>(List.*of*(10,20,30,40,50));  
  
lambda  
al2.forEach(n->System.***out***.println(n));

**set (collection)**

- duplicates are not allowed.

- unordeder collection not having indexing but unique elements

- sorttedSet (Interface) - class Treeset implements SortedSet extends Set extends Collections extends Iterable

- Set clasees (hashSet and linkedHashSet)

**HashSet - dont allow duplicate**- we can set size of hashtable HashSet<Integer> hs = new HashSet<>(20);  
- 20 elements we can store into this hashset and also we can mentioned loading factor also   
- HashSet<Interger> hs = new HashSet<>(20,0.25f);  
i.e only 5 elements filled in above hashset  
  
- In hashset insertion order is not maintained.  
  
**TreeSet** - Insertion orde is not maintained but it is sorted set that doesn't allow duplicates.

**HashSet**: Unordered, allows null. // only single duplicates not allowed

**TreeSet**: Sorted, does not allow null. // NullpointerException

**LinkedHashSet**: Maintains insertion order, allows null. // only single duplicates not allowed

Non Synchronised all HashSet, TreeSet and LinkedHashSet are not thread safe

Set Type Iterator Enhanced for-loop forEach method Index Access

HashSet Yes Yes Yes No (Convert to List for indexing)

TreeSet Yes Yes Yes No (Convert to List for indexing)

LinkedHashSet Yes Yes Yes No (Convert to List for indexing)

Comparable Interface

class Point implements Comparable

this interface contain method compareTo that will compare two points(x,y) and arrange them into sorted order.

**Queue - FIFO**[1,2,3,4,5,6,7,8]  
If the work of 1 is over it will leave first and those who wants add it will add at end like 7 and 8

**Queue** -   
class - PriorityQueue

Interface - Deque - class ArrayDeque implements Deque extends queue extends collection - Iterable

.add(E e); // adding at the end

.poll() // remove first object | not found return null

.remove // did same thing as poll | return NoSuchElementFoundException handle using try catch

.peek() // just know what is first element

.element // same as peek and not found throws NoSuchElementFoundException

class Dequeue - Double ended queue  
means we can insert and delete element from the first and last of index

- used as array basic DS

stack - Insert and delete element from rare side only (back) or front side (LIFO)

queue - Insert from rare and delete from front then it is (FIFO) queue.

**package** practice;

**import** java.util.ArrayDeque;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Collection {

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

ArrayDeque<Integer> dq= **new** ArrayDeque<>(List.*of*(10,20,30,40,50,60));

dq.offerFirst(1);

dq.offerFirst(2);

dq.offerFirst(3);

dq.offerLast(70);

dq.forEach(x->System.***out***.println(x));

}

}

PriorityQueue - Uses DS heap

- Inserting and deleting on priority   
how to decide priority   
value is low then priority is high  
and value is high then priority is low

10,20,15,26,30,5

5 - high priority

30 is having low priority

always deleting element which are at high priority

heap is implementing using binary tree data stucture so we are looking how to data is inserting and deleting using binary tree

cannot left null values in priority queue

A Priority Queue is ordered based on priority.

A Deque is a more flexible queue that allows operations on both ends.

**Both are not allowed null values it will through Nullpointer Excpetion.**

**Map** - Implementation of hashing technique

- storing key and value pair

class - HashMap, LinkedHashMap, hashtable(Legacy class old class)

Interface - sortedMap - class - Treemap

TreeMap<Integer, String> tm = new TreeMap<>(Map.of(0,"A", 1,"B", 2, "C"));

System.out.println(tm);

tm.put();

syso(tm.celingEntry(5).getValue);

celingEntry - it will give same value if present orelse give neareast big value

TreeMap and hashMap are same in some cases

time to execution is diffrent

treemap sorted map

hashmap is not sorted

loading factor is there in hashmap

tree map log(n) time is required in hashmap constant time is required



**HashMap**: Best when you don’t need to maintain any specific order of elements, and you need fast operations (average O(1) time complexity).

LinkedHashMap: Best when you need to maintain the order of insertion or access, with a slight performance trade-off for this ordering.

TreeMap: Best when you need the keys to be stored in a sorted order (natural or custom). However, it has a slower performance for basic operations due to the need to maintain the sorting.

DSA  
  
array :   
- contains homogeneous data  
- arr.length

finding sum of all element

int sum=0;  
for(int x : arr){  
 sum+=x;

}  
  
return sum;  
  
Arrays.sort(arr); // sorting  
  
2. Searching and Element

for(int x : arr){  
iff(x==key){  
syso("found");

System.exit(0); // prpgram will stop here  
//break;  
}

}

syso("Not found");

3. Finding Maximum Element through an array  
int max = arr[0];  
fot(int x :arr){

if(x>max){

max = x;

}

}

syso(max);  
  
**package** collection;

**public** **class** Array {

**public** **static** **int** findingMax(**int**[] arr) {

**int** max = arr[0];

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

**if**(i>max) {

max = i;

}

}

**return** max;

}

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

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

**int** findingMax = *findingMax*(arr);

System.***out***.println(findingMax);

}

}  
  
  
find Max and second max element from an array  
**package** collection;

**public** **class** Array {

**public** **static** **int** findingMax(**int**[] arr) {

**int** max1 = arr[0];

**int** max2 = arr[0];

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

**if**(i>max1) {

max2 = max1;

max1 = i;

}**else** **if**(i>max2) {

max2 = i;

}

}

**return** max2;

}

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

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

**int** findingMax = *findingMax*(arr);

System.***out***.println(findingMax);

}

}  
  
  
**package** collection;

**public** **class** Array {

**public** **static** **int** findingMax(**int**[] arr) {

**int** max1, max2;

max1=max2 = arr[0];

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

**if**(i>max1) {

max2 = max1;

max1 = i;

}**else** **if**(i>max2) {

max2 = i;

}

}

**return** max2;

}

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

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

**int** findingMax = *findingMax*(arr);

System.***out***.println(findingMax);

}

}

Left Rotation of an aaray

Being on perticular index copy element to previous index

**public** **static** **int**[] leftRotation(**int**[] arr) {

**int** temp;

temp = arr[0];

**for**(**int** i = 1;i<arr.length;i++) {

arr[i-1] = arr[i];

}

arr[arr.length-1] = temp;

**return** arr;

}

{1,2,3,4,5,6,7};

2,3,4,5,6,7,1 // left shift

right shift

**public** **static** **int**[] rightRotation(**int**[] arr) {

**int** temp = arr[arr.length-1];

//System.out.println(temp);

**for**(**int** i = arr.length-1; i>0;i--) {

arr[i] = arr[i-1];

}

arr[0] = temp;

**return** arr;

}

LinkedHashMap

key value next prev

- Maintain Order of Insertion

- h(x) = x%10;

- should not fill 100% for LinkedHashMap

- Limit Size 10 delete first Inserted Key

- Used in chache least Rescently used obj is deleted from LinkedHashMap

- reading is slow becuase prev and next

LinkedHashMap as a Chache Memory

used for ordering based on aacesss : 5,.75f,true

LinkedHashMap<Integer, String> arr = new LinkedHashMap<>(5,.75f,true);

arr.put(1,"a");

arr.put(2,"b");

arr.put(3,"c");

arr.put(4,"d");

arr.put(5,"e");

String s = arr.get(4);

s = arr.get(1);

RUO - > 2 3 5 It is uses access and return result based on we are keeping true in LinkedHashMap

follow order of innsertion : first inserted in eldest

and follow Access RUO is removed

LinkedHashMap<Integer, String> arr = new LinkedHashMap<>(5,.75f,true){

protected boolean removeEldestEntry(Map.Entry e){

return size()>5;

}

};

arr.put(1,"a");

arr.put(2,"b");

arr.put(3,"c");

arr.put(4,"d");

arr.put(5,"e");

String s = arr.get(4);

s = arr.get(1);

// 2 will remove

LinkedHashMap<Integer, String> arr = new LinkedHashMap<>(5);

arr.put(1,"a");

arr.put(2,"b");

arr.put(3,"c");

arr.put(4,"d");

arr.put(5,"e");

// here 1 is removed

// you can add multiple elements here

// The method will call removeEldestEntry When you are trying to exceed limit

HashSet : Sort Element

- Does not maintain insertion order

LinkedHashSet<String>

- getting same order which we have inserted

Hashtable ht = new Hashtable();

ht.put(1,"A");

String s = (String)ht.get(1);

Enumeration e = ht.getElements(); // values

Enumeration e = ht.keys(); // keys

while(e.hasMoreElements){

syso(e.nextElement());

}

Property class inherting from class hashtable

- we can set prop in key value pair

- store in .txt as well as xml

- can read both of them

Revision

Class: A template for creating objects, defining the properties and behaviors.

Object: An instance of a class, which contains actual data and can perform actions defined in the class.

Exception

throw:

\* Purpose: The throw keyword is used to explicitly throw an exception from a method or block of code.

\* Usage: You use throw when you want to trigger an exception, either a built-in exception or a custom exception, under specific conditions.

// Define a custom checked exception

public class InvalidAgeException extends Exception {

// Constructor that accepts a custom message

public InvalidAgeException(String message) {

super(message);

}

}

public class TestCustomException {

public static void validateAge(int age) throws InvalidAgeException {

if (age < 0 || age > 150) {

throw new InvalidAgeException("Age must be between 0 and 150.");

}

}

public static void main(String[] args) {

try {

validateAge(-5); // This will throw the InvalidAgeException

} catch (InvalidAgeException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

public class Example {

// Method that declares it may throw a checked exception

public static void checkNumber(int num) throws Exception {

if (num < 0) {

// Using throw to throw an exception

throw new Exception("Number cannot be negative");

}

}

public static void main(String[] args) {

try {

checkNumber(-1); // This will throw an exception

} catch (Exception e) {

System.out.println("Caught exception: " + e.getMessage());

}

}

}

Conclusion:

\* throw: Used to throw an exception.

\* throws: Used to declare that a method might throw one or more exceptions.

Spring boot :

@ControllerAdvice: Global exception handling for all controllers.

@ExceptionHandler: Can be used at both the controller and global levels to handle specific exceptions.

String :   
String s1 = new String("yash"); // heap area

String s1 = "yash"; //String constant pool

s1+"yash"; // create new string object

s1.concat("yash");

String StringBuffer StringBuilder

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

storage | heap / SCP heap area heap area

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

objects | imutable mutable mutable

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

memory | If we modify str consume less memory occupy less   
 it will create new obj memory  
 and allocate more memory

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Thread safe | not thread safe synchronised safe not thread

safe

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Performance | slow fast as compare to string fast String  
 buffer

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Use | frequently data is changing frequently data is changing frequently

String buffer vs builder

In Java, both StringBuffer and StringBuilder are used to create mutable sequences of characters. These classes allow you to modify strings (e.g., append, insert, delete characters) without creating new objects each time. However, there are key differences between the two in terms of thread safety and performance.

1. Thread Safety

\* StringBuffer: It is synchronized, meaning that it is thread-safe. Multiple threads can safely use a StringBufferwithout causing data inconsistency or corruption.

\* StringBuilder: It is not synchronized, meaning that it is not thread-safe. If multiple threads access a StringBuilder concurrently, external synchronization must be applied to ensure thread safety.

Conclusion:

\* Use StringBuffer when thread safety is required.

\* Use StringBuilder when thread safety is not a concern, and you need better performance. In most single-threaded applications, StringBuilder is preferred.

Stream

import java.util.\*;

import java.util.stream.\*;

public class MaxExample {

public static void main(String[] args) {

// Create an ArrayList of integers

List<Integer> numbers = Arrays.asList(1, 5, 3, 9, 7, 2);

// Find the maximum value using Stream API with expanded syntax

Optional<Integer> max = numbers.stream()

.max((a, b) -> Integer.compare(a, b));

// Print the result

max.ifPresent(value -> System.out.println("Max value: " + value)); // Output: Max value: 9

}

}

System.out.println() is used to print output to the console or terminal.

System is a class containing static members for system-level functions.

out is a PrintStream object that represents the standard output stream (console).

println() prints the provided value and adds a new line after it.



Key Points

Lists are ideal when you need to access elements by position and allow duplicates.

Sets are best for collections where you want unique elements without duplicates.

Queues are useful for processing elements in a specific order, like a task queue.

Maps are essential for storing key-value pairs and are frequently used for fast lookups by key.

Index :

1. **Variables**

- **Variable Types**: Local Variable, Instance Variable, Static Variable, Parameter

- **Primitive Types**: byte, short, int, long, float, double, char, boolean

In Java, primitive types are the basic data types that are **not objects** and hold simple values directly.

- store values directly in the **memory allocated** to them

- Since they're not objects, they do not have methods but can be wrapped in classes (e.g., Integer for int, Double for double) when needed.

**ex :**

int primitiveInt = 5; // primitive int

Integer wrappedInt = **Integer.valueOf**(primitiveInt); // Wrapping

int unwrappedInt = wrappedInt.**intValue**(); // Unwrapping

**Simplies java**

int primitiveInt = 5; // primitive int

Integer wrappedInt = primitiveInt; // Autoboxing

int unwrappedInt = wrappedInt; // Unboxing

Benefits of Using Wrapper Classes

Collections Compatibility: Only objects can be stored in collections, so wrapping primitives is necessary.

Utility Methods: Wrapper classes offer methods for conversion (e.g., Integer.parseInt()).

Nullability: Wrapper classes allow null values, while primitives cannot be null.

List<Integer> intList = new ArrayList<>();

intList.add(5); // Autoboxing

intList.add(10);

int sum = intList.get(0) + intList.get(1); // Unboxing

System.out.println("Sum: " + sum); // Output: Sum: 15

- Reference Types: Arrays, Classes, Interfaces

In Java, reference types are types that refer to objects rather than directly storing the data as with primitive types.

we are using refernce to store and access values or methods inside it

String , Enum , Arrays, Classes , Interface

2. Data Types

- Primitive Types: Integer, Floating-Point, Character, Boolean

- Non-Primitive Types: Arrays, Strings, Classes, Interfaces

3. Operators

- Arithmetic: +, -, \*, /, %

- Assignment: =, +=, -=, \*=, /=, %=

- Comparison: ==, !=, >, <, >=, <=

- Logical: &&, ||, !

- Bitwise: &, |, ^, ~, <<, >>, >>>

- Ternary: condition ? trueValue : falseValue

4. Control Statements

- Conditional: if, else, else if, switch

switch :

int day = 3;

String dayName;

switch (day) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day";

break;

}

System.out.println("Day: " + dayName); // Output: Day: Wednesday

java 14 // switch syntax

int day = 3;

String dayName = switch (day) {

case 1 -> "Monday";

case 2 -> "Tuesday";

case 3 -> "Wednesday";

case 4 -> "Thursday";

case 5 -> "Friday";

case 6 -> "Saturday";

case 7 -> "Sunday";

default -> "Invalid day";

};

System.out.println("Day: " + dayName); // Output: Day: Wednesday

- Looping: for, while, do-while, for-each

- Branching: break, continue, return

5. Keywords

Access modifiers control the visibility and accessibility of classes, methods, and variables. They define who can access these elements.

- Access Modifiers: public, protected, private

Non-access modifiers provide additional functionality and properties to classes, methods, and variables, but they don’t change accessibility.

- Non-Access Modifiers: static, final, abstract, synchronized, volatile, transient

synchronized: Used with methods and blocks to restrict access to a single thread at a time, commonly used in multithreading.

volatile: Ensures visibility of changes to variables across threads.

transient: Prevents a variable from being serialized.

- Class Related: class, extends, implements, interface

- Object and Class: new, this, super, instanceof

- Exception Handling: try, catch, finally, throw, throws

- Control Flow: if, else, switch, case, default

- Looping: for, while, do, break, continue

- Others: package, import, return, void

6. Collections Framework

- List: ArrayList, LinkedList, Vector, Stack

- Set: HashSet, LinkedHashSet, TreeSet

- Map: HashMap, LinkedHashMap, TreeMap, Hashtable, WeakHashMap, IdentityHashMap

- Queue: PriorityQueue, ArrayDeque, LinkedList

- Other Utilities: Collections, Arrays, Comparator, Comparable

7. Object-Oriented Programming Concepts

- Class: Definition, Fields, Methods, Constructors, Static Members

- Object: Instantiation, new keyword

- Inheritance: extends, super

- Polymorphism: Method Overloading, Method Overriding

- Abstraction: abstract classes, interface

- Encapsulation: Getters and Setters, Access Modifiers

- Composition: Using instances of other classes as fields

8. Exception Handling

- Keywords: try, catch, finally, throw, throws

- Classes: Exception, RuntimeException, IOException, NullPointerException, ClassNotFoundException, etc.

9. Input and Output (I/O)

- File I/O: File, FileInputStream, FileOutputStream, FileReader, FileWriter

- Buffered I/O: BufferedReader, BufferedWriter

- Data Streams: DataInputStream, DataOutputStream

- Object Streams: ObjectInputStream, ObjectOutputStream

- Character Streams: Reader, Writer

- NIO Package: Paths, Files, Path, FileChannel, ByteBuffer

10. Multithreading and Concurrency

- Thread: Thread, Runnable, Callable, Future

- Synchronization: synchronized block, synchronized method, wait, notify, notifyAll

- Executors Framework: ExecutorService, ScheduledExecutorService, Executors

- Concurrency Utilities: CountDownLatch, CyclicBarrier, Semaphore, Lock, ReentrantLock, Atomic classes

11. Java 8+ Features

- Lambda Expressions: Inline functional programming, (params) -> expression

- Stream API: Stream, Collectors, filter, map, reduce

- Optional Class: Optional, Optional.empty(), Optional.of()

- Functional Interfaces: Predicate, Consumer, Supplier, Function

- Date and Time API: LocalDate, LocalTime, LocalDateTime, ZoneId, Period, Duration

12. Annotations

- Built-in: @Override, @Deprecated, @SuppressWarnings, @FunctionalInterface

- Custom Annotations: User-defined annotations for custom use

- Meta-annotations: @Retention, @Target, @Inherited, @Documented

Spring Boot

@SpringBootApplication

This is the main annotation used to mark a Spring Boot application. It is a combination of three other annotations:

@Configuration: Marks the class as a source of bean definitions for the application context.

we can configure beans over here

@Configuration

public class AppConfig {

@Bean

public MyBean myBean() {

return new MyBean();

}

}

@EnableAutoConfiguration: Tells Spring Boot to automatically configure your application based on the dependencies on the classpath.

- AutoConfig application on dependenciees in classPath

@ComponentScan: Tells Spring to scan the current package and its sub-packages for annotated components (like @Component, @Service, etc.).

Used to tell Spring where to search for components, services, and other beans. By default, @SpringBootApplication already includes this.

@ComponentScan("com.example.services")

public class AppConfig {

}

@SpringBootApplication

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

@Bean

Used within @Configuration classes to define beans that will be managed by the Spring container.

@Bean

public MyBean myBean() {

return new MyBean();

}

@EnableAutoConfiguration

This annotation tells Spring Boot to automatically configure your application based on the dependencies in the classpath.

@EnableAutoConfiguration

This annotation tells Spring Boot to automatically configure your application based on the dependencies in the classpath.

@Component

Marks a class as a Spring-managed bean. This is the base annotation for components like services, repositories, etc.

@Component

public class MyComponent {

}

@RestController

A convenience annotation that combines @Controller and @ResponseBody. It marks the class as a RESTful web service controller, where the methods return data rather than views

@RestController

@RequestMapping("/api")

public class MyController {

@GetMapping("/hello")

public String sayHello() {

return "Hello, world!";

}

}

@RequestParam

Used to extract query parameters from an HTTP request.

@PathVariable

Used to bind a method parameter to a URI template variable.

@GetMapping("/user/{id}")

public String getUser(@PathVariable String id) {

return "User ID: " + id;

}

@RequestBody

Used to bind the body of an HTTP request to a method parameter, typically for handling JSON or XML data.

@RequestBody

Used to bind the body of an HTTP request to a method parameter, typically for handling JSON or XML data.

@ResponseBody

Used to indicate that the return value of a method should be bound to the web response body (for REST APIs).

@ResponseBody

@GetMapping("/message")

public String getMessage() {

return "This is a message";

}

@Autowired

Used for automatic dependency injection. It can be used on fields, constructors, or methods.

@Inject

This is a standard Java annotation (from javax.inject) for dependency injection, similar to

@Qualifier

Used to specify which bean to inject when there are multiple candidates of the same type.

@Autowired

@Qualifier("myBean")

private MyBean myBean;

@Value

Used to inject values into fields from application properties, environment variables, or other sources.

@Value("${app.name}")

private String appName;

@Entity

Marks a class as a JPA entity (used with Spring Data JPA).

@Repository

Marks a class as a repository, typically for data access logic. It also enables exception translation in Spring Data access layers.

@Service

Marks a class as a service provider. It is used in the service layer of the application.