**MY JAVA REMINDER NOTES**

* Java is an Object Oriented Programming Language
* **Primitives** contain just values, **non-primitives** contain values and methods inside the container.
* **Primitives** start with lower case, **non-primitives** start with upper case.
* **Primitives** are created by Java, we cannot create primitive data types.
* **Non-primitives** can be created by programmer, java created some as String.
* The size of **primitive** types depend on the data type, **non-primitives** types have all the same size.
* Java don’t convert large data automatically small data because of that we insist on

**int** n**=** 20**; ==> byte** b**=(byte)** n**;**

* If you use char as data type for char's java gives you the character in java

**char** ch**=**'m'**;**  **==>** 'm'

* If you use int as data type for char's java gives you the ASCII value of the character

**int** dh**=**'m'**; ==>** 109

* Sometimes we may need a different string container even the values are same.

String parentName = **new String**("Tom Hanks");

**METHODS AT STRİNGS AND CHAR**

* **.replace();** method change given 2 text or character in the string

**String** name= “Java is Java”;

**System.out.println(**name.replace(“ava”,”et”)**); // Jet is Jet**

* **.replaceFirst();** method change the given character to another just for the first matching

character in the string

* **.replaceAll()**; method change given whole characters to others

**String** name= “Java is Java”;

**System.out.println(**name.replaceAll(“A-Za-z”,”\*”)**); // \*\*\*\* \*\* \*\*\*\***

* **.toLowerCase();** method make all characters lowercase

**String** name= “Techproeducation”;

**System.out.println(**name.toLowerCase()**); // techproeducation**

* **.toUpperCase();** method make all characters uppercase

**String** name= “Techproeducation”;

**System.out.println(**name.toUpperCase()**); // TECHPROEDUCATİON**

* **.length();** method calculate how many character string has
* **.equals();** method check if 2 given string equals
* **.equalsIgnoreCase();** method checks regardless of upper-lowercase character
* **.concat();** method get together 2 string

**String** str1**= "Java";**

**String** str2**= "Good";**

**System.out.println(**str1.concat(str2)**); // JavaGood**

* **.indexOf();** method tell us the given character’s first index

**String** name= “Java is Java”;

**System.out.println(**name.indexOf(“v”)**); // 2**

* **.lastIndexOf();** method tell us the given character’s last index

**String** name= “Java is Java”;

**System.out.println(**name.indexOf(“v”)**); // 10**

* **.charAt();** method give us the given digit’s character

**String** name= “Techproeducation”;

**System.out.println(**name.charAt(3)**); // h**

* **.substring();** method give us a part of string which is between given digits

**String** name= “Techproeducation”;

**System.out.println(**name.substring(3)**); // hproeducation**

**System.out.println(**name.substring(4,11)**); // proeduc (includin 4 but not 11)**

* **.startsWith();** method check the string if it is start with the given characters

**String** name= “Techproeducation”;

**System.out.println(**name.startsWith(“T”)**); // true**

* **.endsWith();** method check the string if it is end with the given characters

**String** name= “Techproeducation”;

**System.out.println(**name.endsWith(“n”)**); // true**

* **.isBlank();** method checks if the String has any character different from space
* **.isEmpty();** method checks the string if it contains any character or not even if space

**String** name= “”;

**System.out.println(**name.isEmpty()**); // true**

* **.repeat();** method repeat given string at given times

**String** str1**= "Java";**

**System.out.println(**str1**.repeat(**3**)); // JavaJavaJava**

* **.trim();** method erase the space character where it is just begining and ending of string

**String** name= “ Learning Java is so good ”;

**System.out.println(**name.trim()**); //Learning Java is so good//**

* **Math.abs();** It is a method in Math class. Get absolute value of given value

**int** num= -25;

**System.out.println(**Math.abs(num)**); // 25**

* **Math.min();** checks given 2 value to find which one is min

**int** num1= -25;

**int** num2= 12;

**System.out.println(**Math.min(num1,num2)**); // -25**

* **Math.max();** checks given 2 value to find which one is max

**METHODS AT SCANNER CLASS**

* **.next();** method stop when see a space character

// input ==> “Learn Java Earn Money”

**System.out.println(**input.next()**); // Learn**

* **.nextLine();** method get the whole input if its contain a space, also it have a issue, it take you the end of the code because of that we write the code before the other

// input ==> “Learn Java Earn Money”

**System.out.println(**input.next()**); // Learn Java Earn Money**

**METHODS AT WRAPPER CLASS**

* **.valueOf();** method unbox the value from non-primitives to Primitives ( “4” 🡺 4 etc.)

**String** bookPrice**= "$ 35.99";**

**double** bP**= Double.valueOf(**bookPrice**);**

**System.out.println(**bP**); // 35.99**

* If you put char into value of method, it is give you ASCII value
* i=i+1; **==>** i++;
* i=i+3; **==>** i+=3;
* i=i-4; **==>** i-=4;
* j=j\*4; **==>**  j\*=4;
* i=i/6; **==>**  i/=6;
* Nested if-else are time-consuming (time-complexity)
* Switch parenthesis may contain

**i)**int **ii)**byte **iii)**short **iv)**char **v)**String

* Switch parenthesis may not contain

**i)**long **ii)**float **iii)**boolean **iv)**double

* **Boundry Value Analysis (BVA):** To test code with all critical value, one more value before all critical value and one more value after all critical value
* **Nested Structures** use too much time in execution, it makes the application slow. It is time consuming
* **Advice:** If there are more than 2 condition, you should use switch

If same condition here but ternary syntax is easier-shorter than ıf-else, you should choose ternary

* Nested loops are time-consuming (time-complexity). If you have 2 loops, the time complexity is the square of 2 ==>4 (if 3 ==> 9 stc.)
* **do-while loop:** Zero execution is impossible, loop body will be executed at least once

**ARRAYS**

* **Arrays.toString();** is show us all data in the array

**İnt []** set **= {3,26,2,14,48};**

**System.out.println(**Arrays.toString(set)**); //** **[3, 26, 2, 14, 48]**

* **Arrays.deepToString();** is same with toString() but its for multidimensional arrays
* **Arrays.sort();** get the ascending operation in the array. If its string the method puts them in order to initial letter

**İnt []** set **= {3,26,2,14,48};**

**System.out.println(**Arrays.toString(set)**); //** **[2, 3, 14, 26, 48]**

* **Arrays.binarySearch();** checks the array if it is contain specific value. But you must use **.sort()** before using this method

**System.out.println(**Arrays.binarySearch(set,14)**); // 2 ==>**  digit number of 14

* **.length;** gives us number of element in the array. You shouldn’t use () end of the method

**System.out.println(**set.length**); // 5**

* **[i].length();** gives us length of the array’s requested element in the array.

**İnt [][]** cluster**= {{14,48,5,7,0},{3,26,2,14}};**

**System.out.println(**cluster[1].length**); // 4**

* **.toArray(new String[0]);** is convert List to Array

// listConv ==> [hawk, robin, fox]

**String []** arr **=** listConv**.toArray(new String[0]);**

**System.out.println(**Arrays.toString(arr)**); // [hawk, robin, fox]**

* **.equals();** checks if 2 array exact same

**System.out.println(**cluster.equals(set)**); // false**

* **.split();** is starts to read the given string at the begining, split it when its find the key word and puts the each part to requested array. If we use with [], we can take a single part

**String** str**= "No-pain-no-gain";**

**String []** arr **=** str**.split("-"); // [No, pain, no, gain]**

**String** s1 **=** str**.split("-")[0]; // No**

* **For-each loop:** run for every data in the array. It is avaliable for just "Arrays" and "Collections"

**ARRAY LİST**

* Arrays are so fast and use less memory but to create an array we should assign a size. But in array list you don’t need to, its flexible
* ArrayList can store just non-primitive data but Arrays can store primitive data and reference
* **.add();** method puts the element to the end

// ages ==> [13, 9]

ages**.add(**21**); // [13,9,21]**

* **.addAll()**; adds a list elements end of the another list

// prices ==> [330, 550, 770]

ages.**addAll(**prices**); // [13, 44, 9, 21,** 330, 550, 770**]**

* **.addAll(2, prices);** adds prices (etc.) list elements into the "ages" list at index 2

ages**.addAll(**2, prices**); // // [13, 44,** 330, 550, 770**, 9, 21, 330, 550, 770]**

* **.clear();** method removes all elements in a list
* **.contains();** method checks if a specific element exist in a list

**System.out.println(**ages.contains(44)**); // true**

* **.containsAll();** checks if a list's all element exist in an another list

**System.out.println(**ages.containsAll(prices)**); // true**

* **.equals();** checks 2 array list if the lists have same elements in the same indexes

**boolean** result= names1**.equals(**names2**); // true** or **false**

* **.size();** gives us number of element in the list

names1**.size();**

* **.set();** method changes one of the elements of the ArrayList without changing the size. It’ s mean the digit has already a value this method overwrite on it.

// names1 ==> [Spark, Tony, Chirs]

names1**.set(**1,"Jone"**); // [Spark, Jone, Chris]**

* **.isEmpty();** method looks at the ArrayList whether if it is empty or not
* .**remove();** without index remove a first matching value. Returns type is boolean with index; it returns the removed value

// names ==> [Tom, Christian, Tommy, Adriana, Tom]

**System.out.println(**names**.remove**("Tom")**); [Christian, Tommy, Adriana, Tom]**

**System.out.println(**names**.remove(**2**)); // Adriana**

**System.out.println(**names**); // [Christian, Tommy, Tom]**

* **.get();** get the given digit’s value from the array list

// yourList ==> [3, 2, 5]

yourList**.get(**1**); // 2**

* **.indexOf();** method tell us the given character’s first index
* **List<String> list = Arrays.asList();** isconvert Array to List. But fixed size and stored same container. If change one of them, the others change too. .remove() method cannot usable.

**String []** student**= {"Tom", "Grace", "Jimmy"};**

**List<String>** listOfSt **= Arrays.asList(**student**);**

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

listOfSt**.set(**1,"jone"**);**

student **[0] =** "tim"**;**

**System.out.println(**listOfSt**); // [tim, jone, Jimmy]**

**System.out.println(**Arrays.toString(student)**); //** **[tim, jone, Jimmy]**

**DATE CLASS**

* **new Date();** is gives us the date of present

**Date** currentDate**= new Date();**

**System.out.println(**currentDate**); // Thu Jan 26 20:44:23 TRT 2023**

* **.getTime();** is gives us a numeric value, that is millisecond from 1.1.1970 to now

**System.out.println(**currentDate**.getTime()); // 1674755188515**

**LOCAL DATE CLASS**

* **LocalDate.now();** is gives us the date of present in local area

**LocalDate** myDate**= LocalDate.now();**

**System.out.println(**myDate**); // 2023-01-26** ==> year:month:day

* **.getMonthValue();** is gives us month’s number
* **.getMonth();** is gives us month’s name
* **.getDayOfWeek();** is gives us day’s name
* **.getDayOfYear**(); is gives us day’s number
* **.plusYears();** is gives us future date with adding our value

**System.out.println(**myDate**.plusYears(**3**).plusMonths(**4**).plusDays(**12**)); // 2026-06-07**

* **LocalDate.of();** is provide us to create a date value

**LocalDate firstDate= LocalDate.of(2005, 8, 23);**

**LocalDate secondDate= LocalDate.of(2011, 3, 11);**

* **.isAfter();** compares to date and returns boolean

**System.out.println(**firstDate**.isAfter(**secondDate**)); // false**

**LOCAL TIME CLASS**

* **LocalTime.now();** is gives us the time of present in local area.

**LocalTime** myTime**= LocalTime.now();**

**System.out.println(**myTime**); // 20:50:17.461825800** ==> hh:min:sec.ms

* **.minusHours();** is gives us future date with adding our value

**System.out.println(**myTime**.minusHours(**3**).minusMinutes(**20**)); // 17:48:34.461871100**

**LOCAL DATE TIME CLASS**

* **LocalDateTime.now();** is gives us the date-time of present in local area. If we use with continent/city in the ZoneId.of(), it will be the time-date of the city

**LocalDateTime** myDateAndTime**= LocalDateTime.now();**

**System.out.println(**myDateAndTime**); // 2023-01-26T20:54:32.382856500**

**LocalDateTime.now(ZoneId.of("**Asia/Tokyo"**))); // 2023-01-27T03:02:29.044717**

**DATE TİME FORMATTER CLASS**

* **.ofPattern();** is allows us to create a format as we want

**DateTimeFormatter** dtf**= DateTimeFormatter.ofPattern(**"dd:MMMM:yyyy"**);**

* **.format();** is allows us to organize the information we give as we want

// we create myDate value with LocalDate class

**System.out.println(**dtf**.format(**myDate**)); // 28:January:2023**

//d --> 1, 2, 3, ...........31

//dd --> 01, 02, 03......31

//M --> 1, 2, 3, ..........., 10, 11, 12

// MM --> 01, 02, 03 ......, 10, 11, 12

// MMM --> Jan, Feb, Mar, .....Dec,

// MMMM --> January, February, ......

// yy --> 23 (just two last digits of the year value)

// yyyy--> 2023 (whole year value)

* **For time value;**

// we create myTate value with LocalTate class

**DateTimeFormatter** dtf2**= DateTimeFormatter.ofPattern(**"hh:mm:ss a"**);**

**System.out.println(**dtf2**.format(**myTime**)); // 18:24 PM**

//HH --> 24 hours system

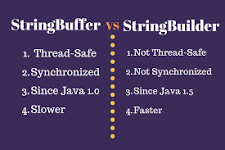
// hh --< 12 hours system

// After time pattern leave a space put "a" it put AM or PM next to the time value

**VARARGS**

* (int... v, String ...s etc.)
* If you want to create a method works with any number of parameters, use "Varargs"
* when you use VarargS you may think you are using Arrays
* It can’t use more than 1 Varargs in a method
* It can’t use another parameter after Varargs in a method. But before the Varargs a different variable can be created.

**STRING BUILDER, BUFFER CLASSES**



|  |  |
| --- | --- |
| Thread-safe means being able to do multiple threads(task) at the same time | Synchronized means being able to select appropriate order for multi thread |

* Both of them are mutable.
* If you need mutable String and do not need multi thread and synchronization use StringBuilder Class to create a String
* If you need mutable String and multi thread and synchronization use StringBuffer Class to create a String
* **StringBuilder** sb1 **= new StringBuilder(**“Java”**);** is create a String as “Java” etc.
* **.append();** is add a value to string as a .concat() method

sb1**.append(**"!"**) // Java!**

* **.insert();** add a data after skiping as many character as the entered value

sb1**.insert(**4," is love"**); // Java is love!**

sb1**.insert(**2, "Money", 1, 4**); // Jaoneva is love!** (if you want to enter some of it)

* **.replace();** change the value which is between the entered digits

sb1**.replace(**2, 5, "1"**); // Ja1va is love!** (First digit inclusive, secont digit exclusive)

* **.deleteCharAt();** remove the given digit’s value

sb1**.deleteCharAt(**2**); // Java is love!**

* **.delete();** delete values between entered digits

sb1**.delete(**8, 12**); // Java is !**

* **.reverse();** reverse the string

sb1**.reverse(); // ! si avaJ**

Most common interview question. If you want tor everse a string you must be mention about StringBuilder’s reverse method

* **.toString();** assign the value to string

**String** newS= sb1**.toString(); // newS= “! si avaJ”**

* **compareTo();** compare two StringBuilder object and return the difference between first characters ASCII value

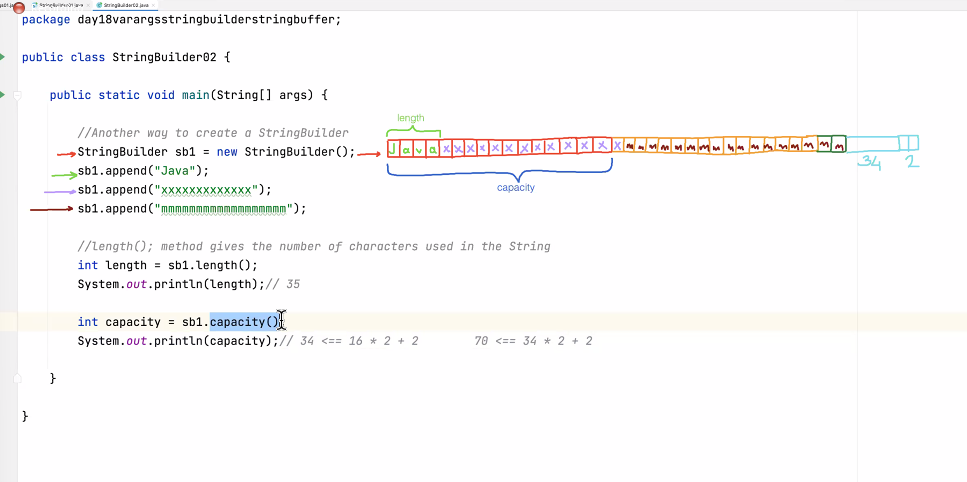
//sb2= Lava sb3=Java

**int** r = sb2**.compareTo(**sb3**);**

**System.out.println(**r**); // -2**

* **.capacity();** gives us the reserved capacity fort he string
* **StringBuilder** sb1 **= new StringBuilder(**4**);** java create 4 capacity fort he string.
* If you dont put a value in parantheses Java create **16** **digit default**
* If the entered value exeed the capacity Java change the capacity like that;

**Existing capacity\*2 +2**

****

**CONSTRUCTORS**

* It is necessary for creating an object
* Java automatically create default constructor when is created a class.
* When we create a constructor, Java automatically delete default constructor
* Method’s have return type but Constructors do not have
* Method’s are for some actions, Constructor’s are for object creation

**public class** Students **{**

**public** Students**(){ }**



**ACCESS MODIFIER**

* **Public:** Open to access for whole project
* **Protected:** Open to access for same package and child class for another package
* **Default:** Open to access for same package
* **Private:** Open to access for just same class
* Access modifier of class can be public or default

**STATIC**

* It makes a variable belong to a class not a object

|  |  |
| --- | --- |
| **STATIC VARIABLE**  Class variable  It can use with class name (also with an object)  Every update can seen by all object  It can be used only in a static method | **NON STATIC VARIABLE**  Instance variable = object variable  To use, an object must be created  If its update with an object, the value change just for the object |

**OOP**

**INHERITANCE**

* Useful for avoiding from repetitions.
* It is easy to make changing-updating on codes
* Child classes can use methods and variables from parent classes but parent classes cannot use methods and variables from child classes
* A child cannot have more than one parent
* For "public" and "protected" access modifiers, there is no issue in "inheritance".
* For "default" access modifier they must be in the same package
* For "private" access modifier, inheritance is impossible
* Static Methods or variables do not take part in inheritance
* **Multilevel Inheritance;** means is a class extends a childclass (child’s child)
* **IS-A ==>** from child to parents
* **HAS-A ==>** from parents to child
* **super()** keyword is used to call first parent class’s constructor, **this()** keyword is used to call in class constructor.
* **this();** runs other constructor in same the class which has same variable type in constructor parantesis

**public** Civic**() {**

**this(**true**); ==> public** Civic**(boolean** a**) {**

**public** Civic**(int** a**) {**

**this(); ==> public** Civic**() {**

* **super();** is already exist in all constructor as default but it is invisible. Itruns parent class’s constructor. If we entered in it a variable, it will goes to run to parent class’s constructor which has same variable type in constructor parantesis

**public** Civic**() {**

**super(**true**);** ==> **public** Honda**(boolean** a**) {**

**public** Civic**() {**

**super(**“Earn”**);** ==> **public** Honda**(String** a**) {**

* **this.** is for calling object from the class itself. **==>** **this.**age**; //** Looks same class
* **super.** is for calling object from the first level parent class. **==> super.**age**; //** Looks parent class
* Java executes Constructors from top to bottom
* If a variable exists in a Class Java uses that otherwise it looks parent class
* OOP (Object Oriented Programing) Concept has 4 principles;

1) Inheritance

2) PolyMorphism --> MethodOverloading() - MethodOverriding()

3) Encapsulation

4) Abstraction --> Abstract Classes and Interfaces

* When we create an object Java start to run relevant constructors from outermost parent class to object’s class **(!)**
* When we create an object, we can select data type of the object from the class itself or from the parent classes.
* When we call a variable Java looks Data type of object at first.

**Animal** cat1 = **new Cat();**

**System.out.println(**cat1**.age); // 4 ==> comes from Animal Class**

* When we call a method Java focus on that method's Class at first.

**Animal** mammal2**= new Mammal();**

mammal2**.eat(); // Mammals eat... ==> comes from Mammal Class**

* **Method Signature:** method name and the parameter list
* **Argument:** A variable in a class
* **Parameter:** A variable in a method parentheses

**METHOD OVERLOADING**

* To create a methods with same name but with different parameters
* Methods any access modifier can be overloaded
* Static and non-static methods can be overloaded
* Method Overloading = Compile Time (Static) polymorphism

**METHOD OVERRIDING**

* To specify, to change method body
* We can apply specific implementation to an inherited method without even modifying the parent class code.
* Modified method in Child class ==> Overriding Method
* Modified method in Parent class ==> Overridden Method
* Rules;
* Method Signature is intouchable
* Overriding Method’s access modifier must be **same or wider** than Overridden Method’s
* If you want to change the return type, there must be **IS-A** relationship.
* Overridden Method’s return type is "**top return type**" of the Overriding Method
* If there is IS-A relationship between return types, those called **Covariant Return Types**
* **private** methods cannot be overridden
* **static** method cannot be overridden
* **final** keyword;
* For variables; you have to initialize, after assigning a value you cannot update the value
* For methods; the method cannot be overridden.
* For classes; the class cannot have "Child Class"
* **@Override** : Override Annotation. Checks the overriding rules
* If you delete @override, InteliJ didn’t assist us. If we do any mistake we cannot override the method, we create another method which is completely different.
* Method Overloading = Compile Time (Static) polymorphism
* Method Overriding = Run Time (dynamic) polymorphism

|  |  |
| --- | --- |
| **OVERLOADING**  To be apply even a single class  Private Method can be overload  Method Parameters are changed  Static method can be overload  Other name => Static Polymorphism  Other name => Compile Time Polymorphism | **OVERRIDING**  Needs Inheritance  Private Method cannot be overridden  Method signature cannot be changed  Method body is changed  Static method cannot be override  Other name => Dynamic Polymorphism  Other name => Run Time Polymorphism |



**ENCAPTULATION**

* Hiding data. Access modifier must be private

**private String** psychologicalIssues**=** **"**Depressions**";**

* If we need to read, we must create getter method in that class;
* Access Modifier is **public**
* Return Type must be the same with the Data Type of the variable
* Method name must be= get+VariableName

**public String get**PsychologicalIssues**() {**

**return** psychologicalIssues**; }**

* If its boolean= is+VariableName

**public boolean is**Success**() {**

**return** success**; }**

* If we need to update, we must use setter method in that class;
* Access Modifier is **public**
* Return Type must be **void**
* Method Name= set + Variable Name
* Parameter in prantesis will be the same with the variable

**public void set**Success**(boolean** success**){**

**this.**success**=**success**; }**

* For Example: If you want to create a "hidden", "updatable" but "not readable" variable:
* Make access modifier private
* Create set method for the variable
* Do not create get method for the variable (So they cannot read from another class)

**ABSTRACTION**

* A method without body
* The other method’s name are; regular method= concrete method
* If we create abstrac method, all concrete child classes have to use it (abstrac child classes not have to)
* To make a method abstract
* **Remove** method body and use “;” not use ”{}” in parent class
* Use **abstract** keyword between "access modifier" and "return type"
* Use **abstrac** keyword between "access modifier" and "class" keyword
* After creating an abstract method, create implementation in all "concrete" child classes.

**public abstract class** Animal **{**

**public abstract void** eat**();**

**} }**

**public class** Cat **extends** Animal**{**

**@Override**

**public void** eat**() {**

**System.out.println**("Cats eat.."**);**

**}**

* Concrete method can be use in abstrac class

**WARNING:** Abstract method cannot be “final”. Because final keyword make it constant

**WARNING:** Abstract class cannot be “final”. Because final class have not child class

**WARNING:** Abstrac class have default constructor too. But Java does not let you create an object from an abstract class. Because it will be defect object.

**WARNING:** Abstract method cannot be private. Because child class have to achieve it

**WARNING:** Abstract method cannot be static

* Same rule in Overriding are valid for implementing abstract methods. (For return type, access modifier, IS-A relation etc.)

**INTERFACE**

* Pros;
* Unlike classes, multiple inheritance is allowed for interfaces
* Huge structure Works slow. Thats why we create a lot of interfaces as parent.
* To avoid all structure broken, we can use a lot of interfaces
* Allows us not to forget mandatory tasks, because all method in interface is abstract and they must be applied all concrete child classes. ( interface = To do list )
* from class to interface ==> implement
* from interface to interface ==> extend
* To make a "class" parent of an "interface" is impossible
* All method in inheritance are “public” and “abstract”. Thats why we don’t have to use those keyword
* Interface methods must be implemented as “public”
* If we want to create a concrete method in interface, we have to use “default” or “static” keyword before return type
* Interfaces are not class so they don’t have constructor that is why we cannot create object from an interface
* Abstract method can’t be static because static methods cannot be overridden
* If we create methods with the same name in different parent interfaces, we have to make “parameters” and "return types" same as well.
* All variables in interfaces are public, static and final.
* An interface may have "public" or "default" access modifier same as classes
* If we create a variable with unique name in interface we can call it from child class with using just variable names.

**Tight Coupling:** Classes which dependent on each other completely. In this structure when a class is broken the others will be broken too (not good).

**Loose Coupling:** Independed classes. This structure provide us to avoid collective collapse.



**COLLECTIONS**

**LINKEDLIST**

* LinkedList does not use indexes, so removing and adding opperation faster than ArrayList.

**LinkedList<String>** students **= new LinkedList<>();**

|  |  |
| --- | --- |
|  |  |

* LinkedList is reccomended for adding and removing operations.
* But for searching operation ArrayList is better.
* **.add();** add a element to the end of the list. If it is used with digits, it add the element to that digits

names**.addLast(**"Brad"**);** **// [Brad]**

names**.add(**1, "Jim"**);**  **// [Brad, Jim]**

* **.addLast();**
* **.addFirst();**
* **.remove();** retrieves and removes the first element of the list. (with digits; is same)
* **.remove();** if it is used with a specific element; it removes the first occurrence of the element and returns true, if it doesn’t exist in the list returns false

names**.remove(**"Brad"**);** **// true**

**System.out.println(**names**);**  **// [Jim**]

* **.removeLast();** removes and returns the last element from this list.
* **.removeLastOccurrence**(); removes the first occurrence of the element and returns boolean

names**.removeLastOccurrence(**"Brad"**);** **// false**

* **.pop();** removes and returns the first element of this list. If list is empty gives an error.

**[Angie, Ali, Brad, Jim, Brad, Brad, Ali]**

students**.pop();** **// [Ali, Brad, Jim, Brad, Brad, Ali]**

* **.poll();** same with pop(), but if the list is empty it doesn’t give an error
* **.element();** retrieves, but does not remove, the first element of this list or null

students**.element();** **// Ali**

* **.getFirst();** same with .element(), but if the list is empty, it gives an error
* **.getFirst();** with digits gives the element of the digits.
* **.set();** update the given digit’s element.

students**.set(**1, "John"**);** **// [Ali, John, Jim, Brad, Brad, Ali]**



**SET**

* Sets are for storing unique data.
* **Sets ==> HashSet, LinkedHashSet, TreeSet**
* Uses hashing technique which is a technique to create unique data(code)
* HashSet does not put the elements in any order, that is why HashSet is **so fast**
* HashSet allows you to store just one "null" as value

NOTE: When we add same element repeatedly, Java does not give error. It puts the repeated element just once into the set.

* LinkedHashSet puts the elements in "insertion order". That is why it is slower than HashSet
* **.retainAll();** removes different elements from the first collection. Returns boolean.

// lhs= [12, 3, 14, 5, 32, 1, 45, 19, null]

// myLhs= [12, 31, 14, 51, 32, null]

lhs**.retainAll(**myLhs**);**  // lhs= [12, 14, 32, null]

* TreeSet puts elements in natural order. That is why TreeSet is so slow.

Interview Question: What do you use to store unique elements in natural order?

Answer: I use TreeSet

But as you know TreeSet is so slow, how can you prevent your code works slowly?

Answer: Create HashSet, add elements into HashSet then convert it to TreeSet.



**QUEUE, DEQUE**

* Queue are First In First Out (FIFO) rule
* Deque stands for Double Ended Queue, it uses FIFO and LIFO (Last In First Out)
* Queue, Deque are interface. We have to use child classes constructer. (LinkedList, PriorityQueue)
* **LinkedList** constructor use insertion order when adding in a list
* **PriorityQueue** constructor use Java Logic when adding in a list, only cares about the first element is the least
* Deque focus on first and last elements.

**ITERATORS**

* Loops have infinite loop risk but Iterators do not have
* There are not performance difference between loops and Iterators.
* Iterators are more successful in removing and updating elements in collections
* **Iterator:** It can just "remove" the elements. It is not possible to update collection. It works just from left to right. // .next() method
* **ListIterator:** It can remove, update and add elements. It works both from left to right and right to left. // .next() method and previous() method
* We can only operate with the element that the pointer is on.
* **.iterator();** convert a list to a iterator.

**Iterator<String>** yourItr**=** yourList**.iterator();**

* **.hasNext();** checks if next element exist and returns boolean

herItr**.hasNext(); // True**

* **.next();** moves the pointer to the next element and gives the next element

// [Tom, Ajda, Brad, Jim, Aeron, Cindy]

yourItr**.next(); // Tom**

* **.remove();** removes the element which is pointer on. Before using this method, we have to move the pointer at least once

yourItr**.remove();** // yourltr ==> [Ajda, Brad, Jim, Aeron, Cindy]



**ENUM**

* A storage to store constant data your application needs.
* We should use “enum” keyword instead of “class” keyword
* We can use only “private” or “default” access modifier for constructors

**EXCEPTION**

* We can avoid from exception by using try-catch block. If we don’t, our application break on that line and the other lines not runs
* Using if-else statement not recommended for avoiding the exception. Because we cannot think every situation
* To handle those, we can use “throws Exception” after method name or we can use try-catch block
* **Run Time Exceptions (Un-Checked Exceptions):**
* **ArithmeticException:** Indicates mathematical errors, especially division by zero.
* **ArrayIndexOutOfBoundsException:** Indicates that an element has been called outside the bounds of the Array
* **StringIndexOutOfBoundsException:** Indicates that the string has been exceeded
* **NullPointerException:** Indicates that any action was attempted on the null reference (Strings can be assign as null)
* **NumberFormatException:** Indicates that the given reference include some characters different from number
* **IllegalArgumentException:** Indicate that a method has been passed an illegal or inappropriate argument.
* **ClassCastException:** Thrown when code tries to cast inappropriate data types

**Object** obj **=** 70**;** ==> **String** s **= (String)** obj**;**

* **IllegalStateException:** IllegalStateException signals that a method's been invoked at an illegal or inappropriate time.

**Iterator<Integer>** intListIterator **= new ArrayList<>().iterator();**

intListIterator**.remove(); // IllegalStateException** ,because pointer on the –1

* **Compile Time Exceptions (Checked Exceptions):**
* **FileNotFoundException:** is related with "file path" and "file existence"
* **IOException:** is related with all Input and Output Exceptions. It is parent of FileNotFoundException
* **.getMessage();** shows the "Technical message" about that exception

// NumberFormatException e

e**.getMessage();** // For input string: "123a"

* **.printStackTrace();** When java completed to run all lines, it adds the end of console detailed error message about that exception

e**.printStackTrace();** // java.lang.NumberFormatException: For input string: "123a"

* **.read();** reads and returns by one by character's ASCII values from a document.
* We can use only “Exception” in try-catch block or multiple “catch” block can be used
* If we want to use multiple catch block in a “try”, child exception catch blok must be used before parent’ block

**}catch (**ArithmeticException e**){**

**System.out.println(**"Most probably divisor is zero - " + e**.getMessage());**

**}catch (**Exception e**){**

**System.out.println(**"An Exception occurred - "**+** e**.getMessage());**

* **finally:** is used to execute some codes under every condition**.**
* “finally” can be used alone
* “try” can not be used alone. It can be used with catch block and/or finally block

|  |  |
| --- | --- |
| **throw**  is used inside the method body  can be used multiple times in a method body  can be used just one exception in a line  can be used just to throw RunTimeException | **throws**  is used in the method signature line  can be used just once in the line  can be used multiple exception in the line  can be used to throw all exceptions |

**void** Demo**() throws ArithmeticException, NullPointerException {**

**throw new ArithmeticException( );**

* When we call a method inside a main method, If calling method has a “throws Exception”, main method must has too.
* To create Custom Run Time Exception, you must extend to RuntimeExceptionClass
* To create Custom Compile Time Exception, you must extend to Exception Class

**ERRORS**

* Errors are the conditions which cannot get recovered by any handling techniques.
* It surely cause termination of the program abnormally.
* Errors belong to unchecked type and mostly occur at runtime.
* Some of the examples of errors are OutOfMemoryError, SystemCrashError, StackOverFlowError etc.

**MAPS**

**HASHMAP**

* HashMap is not "thread-safe" and is not "Synchronized"
* HashMap occur 16 bucket (0-15). If 75% of the existing buckets is full Java creates a new 16 buckets
* Every bucket can contain more than one element.
* Searchin operation is like that ==> Bucket --> Hashcode --> Key ==> The returns value
* Keys should be unique, values can be repeated
* Map elements are called as "Entry", all map elements are called as "Entry Set"
* Entries are in random order, this makes HashMaps so fast
* Key part can have just one null, value part can have more than one null value
* If we use same key repeatedly in a Map, final value will be accepted in the Map (this like updating, but using put() for updating is not recommended)
* **.put();** adds element in a HashMap

// {null=45000000, Democratic Kongo=null, Nicaragua=null}

**HashMap<String, Integer>** cp **= new HashMap<>();**

cp**.put(**"USA", 400000000**);** // {null=45000000, Democratic Kongo=null, Nicaragua=null, USA=400000000}

* **.keySet();** returns all keys in a Set

**Set<String>** keys **=** cp**.keySet();** // [null, Democratic Kongo, Nicaragua, USA]

* **.values();** returns all values in a Collection

**Collection<Integer>** values **=** cp**.values();** // [45000000, null, null, 400000000]

* **.get();** returns the value which belongs a specific key

**Integer** populationOfUsa **=** cp**.get(**"USA"**);** // 400000000

* **.replace();** updates the value which belongs a specific key (with/without checking)

cp**.replace(**"USA", 200000000**);** // {null=45000000, Democratic Kongo=null, Nicaragua=null, USA=200000000}

cp**.replace(**"USA"**,** 200000000**,** 390000000**);** // {null=45000000, Democratic Kongo=null, Nicaragua=null, USA=390000000}

cp**.replace(**"USA"**,** 450000000**,** 390000000**);** // {null=45000000, Democratic Kongo=null, Nicaragua=null, USA=390000000}

* **.size();** returns num of entries

**int** numOfEntries **=** cp**.size();** // 4

* **.getOrDefault();** checks specific key. If it exists, returns value of the entry, otherwise returns our giving default value

**Integer** r1 **=** cp**.getOrDefault(**"USA", 0**);** // 390000000

* **.putIfAbsent();** puts the entry into the map if the key is absent

cp**.putIfAbsent(**"USA", 500000000**);** // {null=45000000, Democratic Kongo=null, Nicaragua=null, USA=390000000}

cp**.putIfAbsent(**"UK", 500000000**);**  // {null=45000000, Democratic Kongo=null, Nicaragua=null, USA=390000000, UK=500000000}

* **.containsKey();** checks if giving key exist

**boolean** r2 **=** cp**.containsKey(**"USA"**);** // true

* **.containsValue();** checks if any element has giving value

**boolean** r3 **=** cp**.containsValue(**45000000**);**  // true

* **.clear();** removes all entries’s keys and values // { }
* **.isEmpty();** checks if map is empty and returns boolean
* **.entrySet();** returns all entries in a Set using Set<Map.Entry<K,V>> data type

**Set<Map.Entry<String, Integer>>** allEntries **=** cp**.entrySet();**

// [null=45000000, Democratic Kongo=null, Nicaragua=null, USA=390000000, UK=500000000]

* For loop;

**for(HashMap.Entry** w**:** cp**.entrySet()){**

**System.out.println(**w**.getKey());**

**}**

**HASHTABLE**

* HashTable is "threadsafe" and "synchronized"
* Entries are in random order
* HashTable are slower than HashMaps
* We cannot use “null” in HashTable

**TREEMAP**

* TreeMaps are not "Thread-Safe" and "Synchronized"
* TreeMap puts the entries in "natural order" by using "keys". That is why too slow
* We can use “null” for values in HashTable

**STATIC BLOCK**

* We can initialize variables with using static block
* Static is run one time when java runs the class
* Static block is run before every other things even if main method

**static {**

**System.out.println(**"This is Static Block is called just once in Class Loading"**);**

**}**



**INSTANCE BLOCK**

* When we want to execute some code for all constructor in the class, we can use Instance Block
* Instance Block is run every times when we create an objeckt from the class.

**{**

**System.out.println(**"I am instance block"**);**

**}**

**FOR INTERVİEW**

* Interview Question: Can you explain "final", "finally", and "finalize" keywords?
* "finalize" is a process Java does before throwing unused things from memory to trash.
* What are the differences between "throws" and "throw" keywords?