**JAVA**

1. **What is object class?**

The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know.

what are the methods in object class?

Totally 9 methods

1.equals()

Object class equals method does reference comparison ( == operator).

It is reflexive: for any non-null reference value x, x.equals(x) should return true.

It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

For any non-null reference value x, x.equals(null) should return false.

For any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

String str = "guru";

String str2= "guru";

String str4= new String("guru");

String str5 = new String("guru");

System.out.println(str.equals(str2)); true

System.out.println(str4.equals(str5)); true

System.out.println(str4 == str5 ); false

System.out.println(str == str2 ); true

System.out.println(str == str5 ); false.. to understand this it is necessary to know how objects created in heap and scp

2.public int hashCode()

Returns a hash code value for the object.

The general contract of hashCode is:

Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.

An object hash code value can change in multiple executions of the same application.

String str = "guru";

String str2= "guru";

String str4= new String("guru");

String str5 = new String("guru");

String str6 = new String("gurumurthy");

System.out.println(str.hashCode());

System.out.println(str.hashCode());

System.out.println(str2.hashCode());

System.out.println(str2.hashCode());

System.out.println(str4.hashCode());

System.out.println(str5.hashCode());..............all above returns same hashcode

String str6 = new String("gurumurthy");........ this one produces a different hashcode

Note:In above case hashcodes are produced based on string value

public String2(String original) {

this.value = original.value;

this.hash = original.hash;

}

If two objects are equal according to equals() method of object class, their hash code must be same as they will have same reference.. ie. in above example str and str2 should return same hashcode...

however, it does not mean if two objects are unequal, then hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables. ie. In above example str4 and str5 are not equals ,, but the hashcode are same which is need not to be.

whereas str5 and str6 hashcodes are not same

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects.

for example str4 and str5 are different objects but produces same hashcode.

Distinct hashcode is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.

3.toString();

4.getClass();

5.clone();

6.finalize()

7.wait();

8.notify();

9.notifyAll()

1. **what is the contract between equals and hashcode method?**

Note that it is generally necessary to override the hashCode method whenever equals method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

1. **what happens when equal method is overridden and not the hashcode method and vice versa?**

Note that your program will not throw any exceptions if the equals() and hashCode() contract is violated, if you are not planning to use the class as Hash table key, then it will not create any problem.

https://www.youtube.com/watch?v=WB1Jia3sXh0

Let’s consider a class

public class person {

private string name;

private int age;

constructor (int age, string name){

this.age = age;

this.name =name;

}

public class TestPerson{

public static void main (string[] arg){

Person p1 = new Person("A",20);

System.out.println(p1); or System.out.println(p1.tostring);

both are same as p1 calls internally toString of object class

System.out.println(p1.hashcode());

}

}

this produces output demo.person@456da3 where 456da3 is a hashcode in hexadecimal value and 156783839938 is a hashcode integer

Now let’s override hashcode method by introducing the this method in person class

@override

public int hashcode(){

return this.age;

}

this produces output demo.person@13 where 14 is a hashcode for age 20 in hexadecimal value and 20 is a hashcode integer

Now let me compare two objects

Person p1 = new Person("A",20);

Person p2 = new Person("A",20);

System.out.println(p1 == p2); this compares address of first object to address(reference) of second object and returns false

System.out.println(p1.equals(p2));equals does the content comparison ,so this should return true but in actual case it returns false. Why? Because note that p1 is an object, so equals is an object class method that cannot compare content of person class & returns false. Remember super class cannot access data of subclass ie. Here equals method is of object class and object class(in general super class) cannot compare data of person class(child class). So subclass needs to override the method to use its own data in method.

so let’s override the equal method so we can compare objects based on data

@override

public boolean equals(Object obj){

if (obj == this)

return true;

if (!(obj instanceof return Person))

return false;

Person person = (Person) obj;

return person.getAge() == this.getAge()

&& person.getName() == this.getName();

}

when we call p1.equals(p2),,p1 calls method equals, so "this" refers to object p1 and argument obj is p1.. hence we return true when obj == this

@override

public string toString(object obj){

return this.age + "," + this.name;

}

tostring is to get some meaningful name when object is printed,, instead of garbage and hashcode in hexadecimal value

now System.out.println(p1.tostring) or even System.out.println(p1) would return A,20 instead of demo.person@456da3

at this moment , everything is working fine ,, equals method works fine,, so there is no scope of overriding hashcode method...

lets wait,,,,,,,,

public static void main (string[] arg){

Set set = new LinkedHashSet<>();

set.add(new Person("A",20));

set.add(new Person("B",30));

set.add(new Person("A",20));

set.add(new Person("C",20));

for(Person p: set)

System.out.println(p);

}

}

ideally above should print only 3 values but it prints 4,,

why? Since we have not overridden hashcode method yet.. we have an

object with value A and 20,, assume whose hashcode is 200

object with value B and 30,, assume whose hashcode is 300

object with value A and 20,, assume whose hashcode is 400

Now first object goes to a bucket say 2 with formula bucket= hashcode/100; where 100 is no of buckets,,, second to 3rd bucket and third object

to 4th bucket allowing a duplicate value in set

To eliminate this, we have to make sure the 3rd object goes to bucket 2..where 1st and 3rd object are compared and inserted only if not same.

so accepting hashcode provided by java caused issue, hence we should override in the above overrided method, we returned age as hashcode because of contract, when object are equal hashcode must be equal..

so ideally if objects are equal then there ages must also be equal,, hence we returned age.

so with this 3rd object is directed to first bucket based on age value 20 and it is compared based on age and name and then inserted.

so if we bring another 4th object say, B and 20,, then it would direct us to 2nd bucket ,, but inserts value as A 20 is different from B 20.

If only equals method is overridden, we won’t be able to retrieve the value because HashMap use hash code to find the bucket to look for the entry and it might look at wrong bucket as while inserting key produced a different hashcode than what it is now. and one another factor is allows the duplicate key as it generates a different hashcode for the same key.

If we only override hashCode(),all the time same hashcode will be returned and we go to the right bucket but value may not be retrieved because equals() method might return false as it does reference comparison.

1. **why there is notify and other multithread related methods in object class?**

Important point to note here is that monitor is assigned to an object not to a particular thread. That's one reason why these methods are in Object class. To reiterate threads wait on an Object's monitor (lock) and notify() is also called on an object to wake up a thread waiting on the Object's monitor

1. **What is the difference between protected and default modifier?**

* **Protected**fields or methods cannot be used for classes and Interfaces
* We can declare class as protected but there is no point in declaring a class as protected. Protected class members have the same visibility as of private except that it also can be accessed from its inherited classes
* The Fields, methods, and constructors declared as **protected**in a superclass can be accessed only by subclasses in other packages.
* The **Default**will act as public within the same package and acts as private outside the package.

<https://www.tutorialspoint.com/what-are-the-differences-between-protected-and-default-access-specifiers-in-java>

<https://stackoverflow.com/questions/20404977/accessing-a-protected-variable-from-a-subclass-outside-package>

1. **String initialization and no of objects creation**

case 1:

String s = new String("guru"); 2 objects will be created.

using the new keyword guarantees that a new String object will be created and a new memory location will be allocated in the Heap memory, so compulsory one object "guru" will be created in heap area with s as reference and for every String literals and constant string expressions a new object will be created in string constant pool, that is why the name constant.

note that these are cached at compile time.. ie.when application loads. The compiler puts them in the String Literal Pool to prevent duplicates and improve memory consumption implicitly reference variable is maintained by JVM for scp objects. some part of Heap area is reserved for SCP. Objects in SCP is not eligeble for garbage collector , it is maintained by JVM

String s = "guru"; 1 object will be created.

that is in scp area. If the object is not present with the content only then the object will be created.

so for String s = new String("guru") and String s = "guru"; totally only 2 objects will be created.

case2:

String s1 = new String("guru"); 2 objects, since new keyword is used one in heap area with reference s1 and other in scp as content guru is not present in scp

String s2 = new String("guru"); 1 object,since new keyword is used one in heap area with reference s2 and already content is present in scp so not created

String s3 = "guru";0 object,since there is no new keyword no heap area object and already content is present in scp so s3 reference is pointed to scp

String s4 = "guru";s4 refernce is pointed to content "guru" in scp area(future purpose: this is scp is used)

totally 3,ie. 2 objects in heap area and 1 object in scp

case3:

String s = new String("guru");2, one in heap and other in scp

s.concat("murthy"); 1 in scp with content murthy as it is literal,

during runtime operation if objects are created, it is created only in heap area not in scp. so 1 object with content gurumurthy in heap, but it does not have any reference in heap memory, hence it is available for garbage collector.

s = s.concat("solutions"); 1 in scp with solutions as it is literal, due to run time operation 1 object gurumurthysolutions in heap area referencing s

since s now reference gurumurthysolutions , guru will be available for GC

totally 6 objects, guru, gurumurthy,gurumurthy solutions in heap area with guru availabe for gc and guru,murthy,solutions in scp

1. **Static modifier**

* We can declare variables and methods as static.
* We can also declare inner classes as static but not top class
* If static variable is not initialized, JVM does it for you
* In case of instance variable for every object a separate copy will be created, but in case of static a single copy is created at class level for all objects.
* Instance variables cannot be directly accessed in static methods
* Inheritance applicable for static methods including the main method
* Static method cannot call non-static methods, to do so you need to create the instance of that static method
* If you are not using any instance variable compulsorily you should mark method as static

Ie. simple utility method, let’s say a method that accepts 2 integers and returns sum should be a static method.

1. **Static control flow**
2. Identification of static members.
3. Execution of static variable assignment and static block from top to bottom
4. Execution of main method which is the last thing to get executed
5. **Method signatures/Polymorphism**

Method signatures consist of method names followed by argument types.

Compiler used method signature to resolve method calls

Within a same class two methods with same signature cannot exist

Overloading:

* When method has same name but different argument types
* Resolution always taken care by compiler based on reference type, hence overloading is also called as compile time polymorphism
* Only overloading method in java is concat

A = guru, B=10, C=20, D=30

A+B+C+D =GURU102030

B+C+D+A= 60GURU

B+C+A+D=30GURU30

* Automatic promotions of datatype allowed in overloading

1. You can pass integer to a method with argument float.
2. If you try to pass double instead, it is considered as overloading as double > float
3. Methods with object and string as argument type, if null is passed it will have priority to the child method. But if null is passed for string and stringbuffer it gives compilation error.

Overriding:

1. Child class inheriting the method of parent class but can have different implementation
2. Method name and method argument remains same but return type can vary. Ex: If parent class return type is object, then child class can return its object class child like string, stringbuffer etc. But cannot do the vice versa
3. Private or final methods of parent class is not available for overriding
4. If parent method is public then overriding method should also be public. Modifier cannot be overridden
5. We can override non abstract or final methods as abstract/final
6. In overriding the resolution is based on the run time, hence called runtime polymorphism

Ie. Parent class P has a method that is overridden in child class C.

When you create an object with parent as reference holder

P p = new C(); and call method

p.method(); compiler checks if the method exist in parent class

but when executing it executes child class as parent object cannot call child class, which is determined during runtime.

1. **Data hiding**

Outside person cannot access our internal data directly or our internally data should not go outside directly is nothing but data hiding.

This can be achieved by making a variable private and have some validation in getter method.

1. **Encapsulation**

Process of binding data(variables) and corresponding methods into a single unit class is nothing but encapsulation.

It increases data security but length of the code increases thus slowing down execution

1. **Inheritance [is-a-relationship]**

* Inheritance is achieved by extending the class.
* Reusability of code is the main advantage
* Parent class can hold child class reference but cannot call method of child class. Can be done only by typecasting it back to child class
* Child class cannot hold reference of parent class at all
* Java can’t extend more than one class at a time, thus multiple inheritance is not allowed. This is why we have interfaces

1. **Composition and Aggregation [has-a-relationship]**

* Composition: Without the existence of container object child object can’t exist. Ex: No classroom without school
* Aggregation: Child class can have their own lifetime. Ex: Marks and students
* If all the methods of parent class is required go for is-a-relationship(abstraction) else go for has-a-relationship(interface)

1. **Abstraction**

* Abstraction is a modifier applicable for classes and method but not to variables.
* When we know about the requirement but not details then we can go for abstract methods and child class is responsible for implementation
* Every child class that extends abstract class must implement all abstract methods ,partial implementation will make child class also as abstract
* Abstract class cannot be final, native, synchronized, private or strictfp. These are illegal combination
* Objects cannot be created for an abstract class
* Can have constructors
* We cannot create object for abstract class but we can have constructors, because constructor is executed whenever we are creating child class object to perform initialization of child class object.

1. **Interface**

* Every method is public and abstract. Other modifiers are not allowed although with latest versions of java default methods with implementations are available.
* Every variable is public, static and final. Hence variable declaration is compulsory
* No static and instance blocks are allowed to declare
* Cannot declare constructor

1. **Exception Handling**

* Exception is an unexpected event that disturbs the normal flow of execution
* It is recommended to handle exception for graceful termination of program
* For every thread JVM will create a run time stack. Each and every method call performed by that thread will be stored in corresponding stack. Each entry in stack is called stack frame. After Completing every method the corresponding entry from the stack will be removed. After completing method calls stack will become empty and will be destroyed by JVM before terminating the thread.
* Throwable class has two child class Exception and Error
* VM error, Memory error, Stackoverflow error
* Checked exceptions by compiler such as filenotfoundexception . All checked exceptions uses throws keyword in method signature
* Unchecked exceptions are those not handled by the compiler such as airthmeticException, these occur only during runtime

1. **Serialization**

* Process of writing a state of an object to a file is called serialization. Strictly speaking it is process of converting a java object into a network supported form or file supported form ie. Usually byte stream
* It is possible to serialize multiple objects but while deserializing it should be done in same order as that of serialization
* Object object = oos.readObject(); where oos is objectoutputstream object gives the order of serialization.
* FileOutputStream fos = new FileOutputStream(“abc.txt”);

ObjectOutputStream oos = new ObjectOutputStream(fos);

Oos.writeobject();

* After serialization the transient object will be lost, to recover it custom serialization can be developed. Write custom read and write object methods
* Child class inherits serialization of parents class by default even if not extended in child class
* In serialization everything is taken care by JVM and we do not have control over what properties are serialized, to achieve it we go for externalization.
* SerialVersionUUId is required by JVM to ensure sender and receiver have loaded classed for the objects that are compatible with respect to serialization

<https://www.simplilearn.com/tutorials/java-tutorial/serialization-in-java#:~:text=Serialization%20in%20Java%20is%20the,then%20de%2Dserialize%20it%20there>.

1. **Why serialization required when data can be transferred over rest api?**

* <https://stackoverflow.com/questions/2475448/what-is-the-need-of-serialization-of-objects-in-java>

1. **String, StringBuffer and StringBuilder**

* String is immutable
* String builder is mutable
* String buffer is mutable and thread safe

1. **Deep copy and Shallow copy**

Deep Copy:

* When we do a copy of some entity to create two or more than two entities such that changes in one entity are not reflected in the other entities, then we can say we have done a deep copy.
* In the deep copy, a new memory allocation happens for the other entities, and reference is not copied to the other entities. Each entity has its own independent reference. The following example demonstrates the same.
* In order to make the clone() method support the deep copy, one has to override the clone() method.

Shallow Copy:

* When we do a copy of some entity to create two or more than two entities such that changes in one entity are reflected in the other entities as well, then we can say we have done a shallow copy.
* In shallow copy, new memory allocation never happens for the other entities, and the only reference is copied to the other entities.
* The default version of the clone() method supports shallow copy.

<https://www.javatpoint.com/shallow-copy-vs-deep-copy-in-java>

Note that in cases such as string and primitive data types deep or shallow copy does not make any sense.

1. **JVM Memory model**
2. Heap area

* One heap area per JVM
* Created during JVM start
* Stores objects, instance variables
* Not threadsafe
* Need not to be contiguous.

1. Method area

* This is where class is loaded. It contains constant pool, fields, method code, constructor code etc that are used in class and initialization of objects
* Shared by all JVM threads
* Created during JVM start up
* Generally, part of heap area, it could be fixed or vary in size
* JVM implementation can give control to programmer over Method area creation, its sizing etc. If method area memory is not sufficient to satisfy an allocation request then JVM throws OutOfMemoryError.

1. Stack area

* Where method is loaded and its execution takesplace
* A stack is created for every thread

1. Native method stack

* Similar to stack but dedicated to native methods only

1. PC register

* Keeps track of current execution instructions at any given moment

1. **JVM memory model changes in java8**

* JVM part of JRE which in turn part of JDK.
* <https://connect2grp.medium.com/evolution-of-java-memory-model-af24d5365581>
* <https://www.geeksforgeeks.org/metaspace-in-java-8-with-examples/>

1. **JVM Architecture**

* JVM part of JRE which in turn part of JDK.
* Class loader responsible for

1. Loading

.class file is stored in path, reads and stores in JVM memory area with fully qualified name, parent , methods , constructor etc information. Then JVM will create class.class object to represent the .class file in Heap area

1. Linking

It consists of verifying, preparing and resolving activities.

Component bytecode verifier verifies if .class file is correctly formed structurally, if not throws verifyError

Prepare allocates memory to static variables ,assign default values and execute static methods

All the symbols stored in constant pool, throws symbol cannot find error

1. Initialization

Original values are assigned in place of default values assigned by linking and static blocks are executed in order

* + - * 1. Bootstrap class loader: Loads all java API present in rt.jar file in JDK🡪JRE🡪LIB🡪rt.jar. This is implemented in c++
        2. Extension class loader: Child class of BCL is implemented in java and loads path JDK🡪JRE🡪LIB🡪EXT🡪\*.java
        3. Application class loader : Loads class path environment variables
* Memory area
* Execution engine: Central component of JVM that contain

1. Interpreter: reads byte code and converts to machine code and executes line by line. Problem with this is its redo work even if same method executed again and again
2. JIT compiler: To improve performance of interpreter JIT came in. Multi invoked methods are converted into machine code and provided by JIT.
3. **How do you choose different garbage collector in java.**

* java -XX:+UseSerialGC -jar Application.java
* <https://sematext.com/blog/java-garbage-collection-tuning/#:~:text=The%20number%20of%20threads%20that,%2DXX%3AParallelGCThreads%3D4>.

1. **Ways to define a thread**
2. By extending thread class and overriding run method.

* Lifecycle begins with creation of thread object and ends with calling start() method
* Start() registers the thread with thread scheduler, perform all mandatory activities and invoke run()
* if we call run() directly instead of start(), instead of two threads only one will be created and it executes the run method

1. By implementing runnable interface

* Advantage is that , multiple inheritance issue is resolved

1. **Thread priorities**

* Every thread in java has some priority, either default or customized.
* Priority varies from 1 to 10, minimum priority being 1 and first 10 is executed
* Thread scheduler allows thread with highest priority
* If priority greater than 10, illegal argument exception is thrown when setting priority
* Default priority for thread will be inherited from parent
* Main thread has priority of 5

1. **Ways to prevent thread execution**
2. Yield()

* Whenever thread.yield() is called, current executing thread gives chance to other threads of same priority which are waiting.
* If no other threads or of lower priority, then it continues execution
* If multiple threads are waiting with same priority, thread scheduler decides which thread get chance

1. Join()

* If thread t1 wants to wait for thread completion of t2, then t1 calls t2.join()
* Meanwhile t1 goes to wait status till t2 is completed

1. Sleep()

* When thread do not want to perform operation for specified time
* Throws interrupted execption

1. **Executor Service**

* When you create a thread and call start method its single thread run
* When there are hundred jobs and 10 thread you want them to run asynchronously that picks a job, completes and again picks. To achieve this we have executor service

ExecutorService s = new ExecutorService.newFixedThreadPool(10);

For(int i = 0; i< 100; i++){

s.execute(new Task());

}

Where class Task implements Runnable interface

* Internally used blocking queue since all threads act on queue to take task concurrently. To be threadsafe blocking queue is implemented

1. **9 Key interfaces in collections**
2. Collection
3. List
4. Set
5. Sorted Set
6. Navigable Set
7. Queue
8. Map
9. Sorted Map
10. Navigable Map
11. **Collection**

* If we want to represent group of individual heterogenous objects as single entity then we should go for collection.
* Considered as root interface of collections framework
* There is no concrete class that implements collection, only other interfaces
* List , Set and Queue are child interfaces of collection. Not Map

1. **List**

* List is when we want to represent group of individual objects as entity where duplicates are allowed, null is allowed and insertion order is preserved.
* Insertion order is preserved via index and we can differentiate duplicate objects via index

Arraylist

* Underlying data structure is array
* Random access allowed
* Initial capacity is 16
* Shift operation while adding objects, hence preferred for retrieval operation than insertion
* Elements are stored in consecutive memory location
* Every method is non synchronized unlike vector
* Not threadsafe unlike vector
* Performance high compared to vector
* Synchronized version of arraylist can be obtained Collections.synchronizedList(al); where al is arraylist

LinkedList

* Underlying data structure is double linked list
* No random access but implements serializable and cloneable interface
* Best choice for frequent insertion operation
* Elements not stored in fixed fashion
* We can use Linkedlist to develop stacks and queues. Hence it has specific methods such as addfirst,addlast,removefirst,removelast,getfirst,getlast

Vector

* Null insertion, serializable, cloneable, random access and threadsafe
* Default capacity is 10
* addElement,removeElement,add,add(index,object) are few methods
* Designed for FIFO(First in First out)

Stack

* Child class of vector
* Designed for LIFO(Last in First out)
* There is only one constructor
* Push,Pop,Peek,Empty,Search are the methods

1. **Set**

* Set is when duplicates should not be allowed

1. HashSet

* It is a class that implements Set interface and extends AbstractSet class
* Underlying date structure is hashtable.
* Insertion order is not preserved
* All objects inserted based on hashcode.
* Only once null is allowed
* Except treeset heterogeneous objects are allowed
* Serializable, Cloneable and No random access
* Best for search operation based on hashcode
* Initial capacity is 16 with default fill ration of 0.75
* Non synchronized
* Requires less memory

1. Linked HashSet

* It is a class that implements Set interface and extends Hashset class
* Underlying data structure is LinkedList + hashtable
* Insertion order is preserved
* All objects inserted based on hashcode.
* Only once null is allowed
* Non synchronized
* Requires more memory than hash set
* Used in developing cache based applications

1. Tree Set
2. Sorted Set
3. It is an interface that extends Set interface
4. Insertion order is sorted in some order
5. First, last, head, tail, subset are some specific methods
6. Navigable Set
7. It is an interface that extends Sorted Set interface
8. Along with Sorted set interface methods it provides methods such as lower, higher, ceiling , floor, pollfirst, pollLast, descendingSet etc

* It is a class that implements Sorted Set or Navigable Set interface
* Underlying data structure is Balanced Tree
* Insertion order is not preserved but sorted in some order
* Heterogenous elements are not allowed
* Null is not allowed
* Sorting depends upon the supplied comparator
* Uses compare and compare toe method for comparison
* Performance slower compare to hashset

1. **Map**

* Object is combination of key and value
* Map is considered as collection of Entry objects where Entry is an interface withing Map Interface with entry specific methods such as getKey, getValue, setValue
* Map return type is object

1. HashMap

* It is a class that implements Map interface
* Underlying date structure is hashtable.
* Insertion order is not preserved
* All objects inserted based on hashcode of keys.
* One null key and any number of null values allowed
* Except treeMap heterogeneous objects are allowed
* Serializable, Cloneable and No random access
* Best for search operation based on hashcode
* Non synchronized, not threadsafe unlike hashtable
* Performance is high as not threadsafe

1. Linked HashMap

* It is a class that inherits(extends) HashMap and implements HashMap
* Underlying data structure is LinkedList + hashtable
* Insertion order is preserved
* All objects inserted based on hashcode.
* Only once null is allowed
* Non synchronized
* Default capacity is 16 and loading factor is 0.75

1. Tree Map
2. Sorted Map
3. It is an interface that extends Map interface
4. Insertion order of key is sorted in some order
5. Firstkey, lastkey, headmap, tailmap, submap are some specific methods
6. Navigable Set
7. It is an interface that extends Sorted Map interface
8. Along with Sorted Map interface methods it provides methods such as ceilingKey, ceilingEntry, pollLast, descendingKeySet etc

* It is a class that implements Sorted Map or Navigable Map interface
* Underlying data structure is Red-Black Tree
* Insertion order is not preserved but sorted in some order
* Heterogenous elements are not allowed
* Null is not allowed
* Sorting depends upon the supplied comparator

1. **Queue**

* Interface that provides functionality of the queue data structure
* [BlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html), [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html" \o "interface in java.util.concurrent), [Deque](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html), [TransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html" \o "interface in java.util.concurrent) implements Queue
* [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html), [ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [ArrayDeque](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html), [ConcurrentLinkedDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html), [ConcurrentLinkedQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html), [DelayQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html), [LinkedBlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html), [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html), [LinkedTransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html), [PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html), [PriorityQueue](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html), [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html) extends Queue
* PriorityQueue priorityQueue = new PriorityQueue();  
  ConcurrentLinkedQueue concurrentLinkedQueue = new ConcurrentLinkedQueue();  
  //above two class directly implements Queue interface  
    
  TransferQueue transferQueue = new LinkedTransferQueue();  
  //This LinkedTransferQueue class implements TransferQueue interface that extends Queue interface  
    
  BlockingDeque blockingDeque = new LinkedBlockingDeque();  
  //This LinkedBlockingDeque class implements BlockingDeque interface that extends Queue interface  
    
  BlockingQueue blockingQueue3 = new DelayQueue();  
  BlockingQueue blockingQueue4 = new PriorityBlockingQueue();  
  BlockingQueue blockingQueue5 = new SynchronousQueue();  
  BlockingQueue blockingQueue = new LinkedBlockingQueue();  
  //Above 4 classes implements BlockingQueue interface that extends Queue interface  
    
    
  Deque deque = new ConcurrentLinkedDeque();  
  Deque deque2 = new LinkedBlockingDeque();  
  Deque deque3 = new ArrayDeque();  
  Deque deque4 = new LinkedList();
* AbstractQueue is an abstract class. Some of the above classes instead of directly implementing queue , they extend AbstractQueue class which inturn implement Queue which means these classes have more methods to implement or job to do than those classes that directly implemented Queue
* Some method that implement other interfaces that extends Queue because those class may not require all the methods available in Queue or some other specific implementation would be required which is defined in interfaces they implemented which extends main interface Queue

Methods of Queue are add, offer, element, peek, remove, poll

* Being an interface, the queue requires, for the declaration, a concrete class, and the most common classes are the LinkedList and PriorityQueue in Java. Implementations done by these classes are not thread safe. If it is required to have a thread safe implementation, PriorityBlockingQueue is an available option.
* Queue used to keep the elements that are processed in the First In First Out (FIFO) manner.
* add() that come from collection will throw an IllegalStateException if no space is currently available in the Queue, otherwise add method will return true. offer() that comes from queue method will return false if the element cannot be inserted due to capacity restrictions
* peek() and element() methods returns the first element . If element does not exist peek() returns null where are lateral throws NoSuchElementException
* poll() removes first element where as remove method takes own given element.

1. PriorityQueue

* It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.
* Add and offer methods are synonymous in priorityqueue, no difference at all because Capacity is unbounded and auto grows internally.
* The method can throw two types of exceptions:

NullPointerException: If the element to be inserted is NULL.

ClassCastException: If an element to be inserted is of a different type that cannot be compared to the existing elements of the Queue.

* Actually, internal data structure of PriorityQueue is not ordered, it is a binary [heap](http://en.wikipedia.org/wiki/Heap_(data_structure)). PriorityQueue doesn't need to be ordered, instead, it focuses on head of data. Insertion is in *O(log n)* time. Sorting wastes time and useless for a queue. Like a tree structure..

1

| \

3 2

|

4

* PriorityQueue allows only those elements which implements Comparable or you need to provide Custom Comparator. Integer and String are allowed as they implement comparable. If you are storing an employee class object you should provide implement comparable or provide your custom comparator.

1. BlockingQueue

* It is a class that implements Map interface
* Underlying date structure is hashtable.
* Designed for producer & consumer queues with all public operations are protected with a single lock
* BlockingQueue allows multiple threads to insert the data and remove the data
* It is a bound queue which means size cannot grow, so if memory if full queue waits to add element into queue till some memory is freed by removing element. Similarly if there are no elements in queue, it blocks threads from retrieval operation.

1. PriorityBlockingQueue

It is an interface that extends Map interface

It’s DS is binary heap tree like PBQ.

PriorityBlockingQueue uses the same ordering rules as PriorityQueue and supplies blocking retrieval operations

<https://hellokoding.com/priority-queue-implementations-in-java/>

1. SynchronousQueue
2. It is
3. Insertion order of key is sorted in some order
4. SynchronousBlockingQueue
5. It is
6. Insertion order of key is sorted in some order
7. ArrayBlockingQueue
8. Once created, we cannot grow or shrink the size of the Queue
9. ArrayBlockingQueue is always bounded.
10. LinkedBlockingQueue
11. LinkedBlockingQueue creates nodes dynamically until the capacity is reached. This is by default Integer.MAX\_VALUE. Unlike array blocking queue Using such a big capacity has no extra costs in space.
12. LinkedBlockingQueue is optionally bounded
13. DelayQueue
14. It is a class
15. When the consumer wants to take an element from the queue, they can take it only when the delay for that particular element has expired
16. Deque

* It is a class that inherits(extends) HashMap and implements HashMap
* Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

1. ArrayDequeue
2. It is an interface that extends Map interface
3. LinkedList
4. It is an interface that extends Map interface
5. Insertion order of key is sorted in some order
6. ConcurrentLinkedDequeue
7. It is an interface that extends Map interface
8. Insertion order of key is sorted in some order

* Insertion order is not preserved but sorted in some order
* Heterogenous elements are not allowed
* Null is not allowed
* Sorting depends upon the supplied comparator

1. BlockingDequeue
2. LinkedBlockingDequeue
3. dkfjaldfka
4. TransferQueue
5. LinkedTransferQueue
6. Kdfksa
7. Dskfajslfa
8. ConcurrentLinkedQueue

* Jkflaskfas

1. **Types of Iterators**

* Enumeration
* Iterator
* ListIterator

1. **Comparable and Comparator Interface**
2. Comparable

* Java.lang package
* campareTo method

1. Comparator

* Java.util package
* Compare and equals methods

1. **Generics**

To provide type safety and resolve type casting problem.

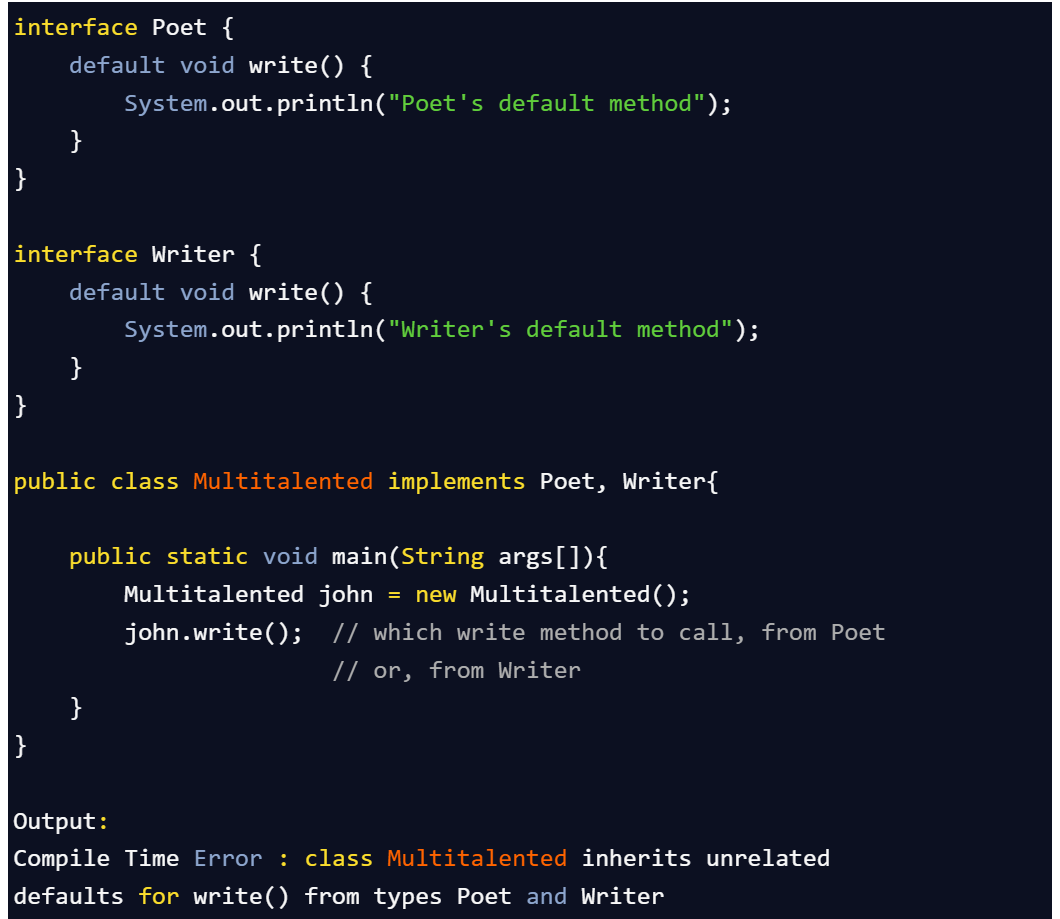
1. **Callback vs Runnable interface**

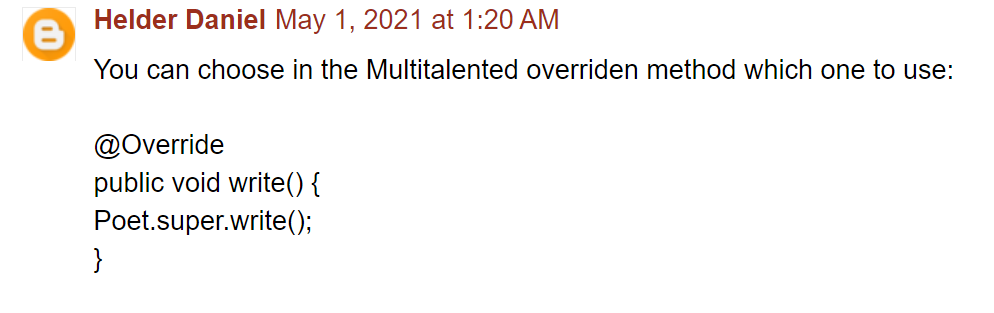
To provide type safety and resolve type casting problem.

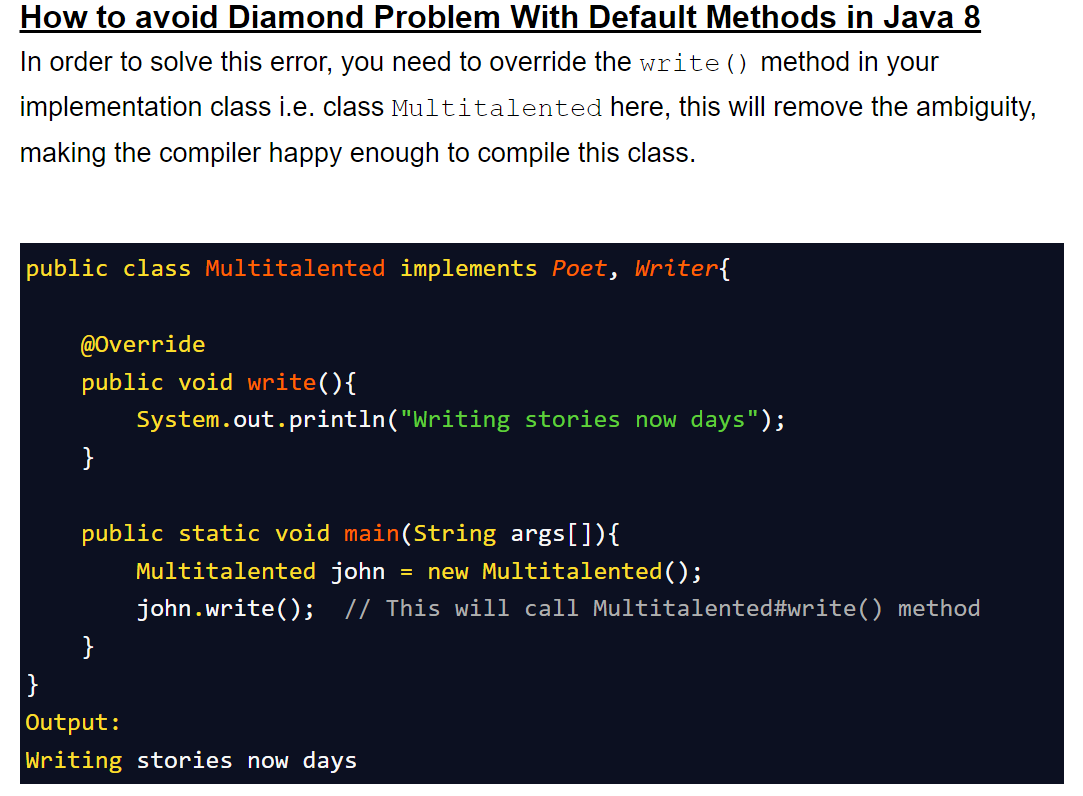
1. **Concurrency model in java**

To provide type safety and resolve type casting problem.

1. **Default method ambiguity in java interface**







1. **Changes done to Hashmap in java8**

To provide type safety and resolve type casting problem.

1. **Generics**

To provide type safety and resolve type casting problem.

1. **Generics**

To provide type safety and resolve type casting problem.

1. **Generics**

To provide type safety and resolve type casting problem.

1. **Generics**

To provide type safety and resolve type casting problem.