Project Questions -

What is general process do you follow?

Check-out / check-in

Servers

Use javap command to check class availability.

Ex : javap java.util.ArrayList

**Important and required concepts.**

Classes,

Inheritance

Interfaces

Dynamic method dispatch

Accesses

Multi threading (synchronization)

Data types

Collections

Generics

Exception handling

**-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**

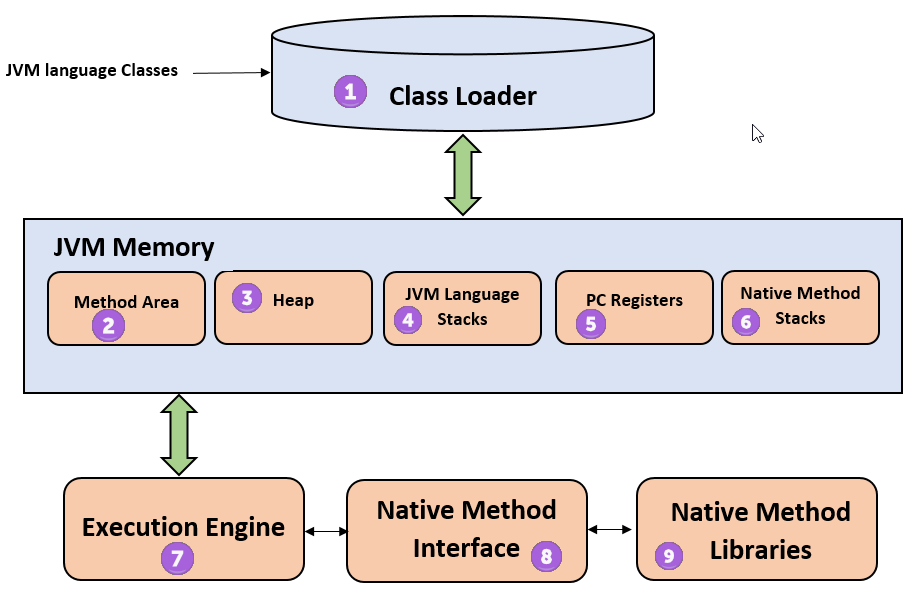
**Java Virtual Machine**

JVM is an engine that provides runtime environment to drive the Java Code or applications. It converts Java bytecode into machines language. JVM is a part of JRE (Java Run Environment). It stands for Java Virtual Machine.

In other programming languages, the compiler produces machine code for a particular system. However Java compiler produces bytecode that gets interpreted by Java Virtual Machine.

**JVM Architecture**

Let's understand the Architecture of JVM. It contains class loader, memory area, execution engine etc.

[](https://www.guru99.com/images/1/2.png)

1. Class Loader: The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

2. Method Area: JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.

3. Heap: All the Objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

4. JVM Language Stack: Java language Stacks store local variables, and its partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.

5. PC Registers: PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

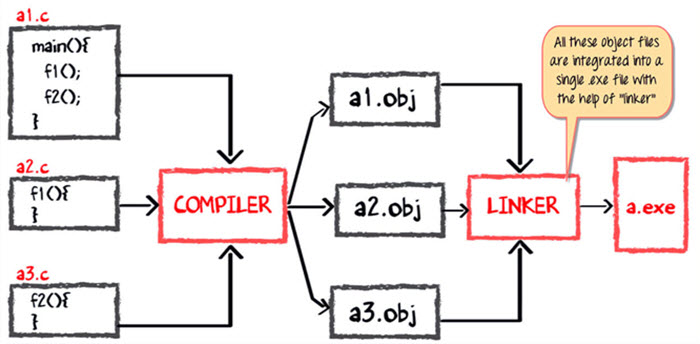
6. Native Method Stacks: Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.

7. Execution Engine: It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

8.  Native Method interface: The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.

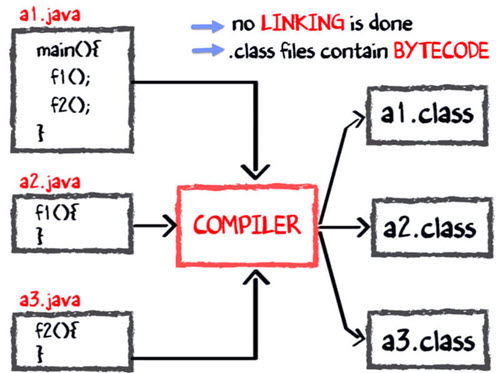
9. Native Method Libraries: Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine.

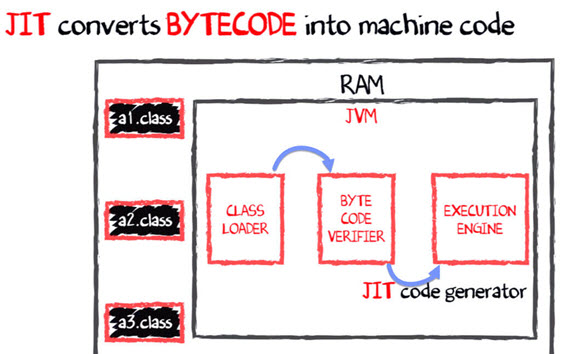
C Compilation:

[](https://www.guru99.com/images/java/052016_0614_WorkingofJa5.jpg)

a.exe will be executing in RAM.

Java Code compilation

[](https://www.guru99.com/images/java/052016_0614_WorkingofJa7.jpg)

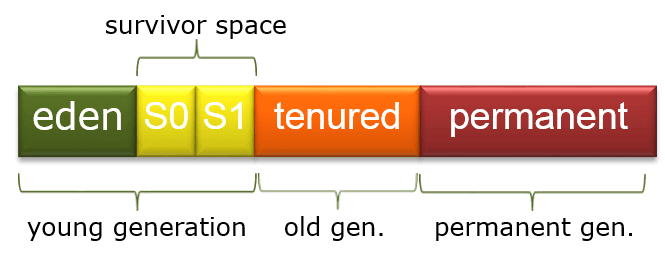
[](https://www.guru99.com/images/java/052016_0614_WorkingofJa9.jpg)

Java virtual machine resides on the RAM. During execution, using the class loader the class files are brought on the RAM. The byte code is verified for any security breaches. Next the execution engine will convert the bytecode into Native machine code.

**How Garbage Collection works in JAVA?**

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

Java garbage collection is an automatic process. The programmer does not need to explicitly mark objects to be deleted.



The heap is divided into [three sections](https://plumbr.eu/handbook/garbage-collection-in-java):

**Young Generation**: Newly created objects start in the Young Generation. The Young Generation is further subdivided into an Eden space, where all new objects start, and two Survivor spaces, where objects are moved from Eden after surviving one garbage collection cycle. When objects are garbage collected from the Young Generation, it is a minor garbage collection event.

**Old Generation:** Objects that are long-lived are eventually moved from the Young Generation to the Old Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.

**Permanent Generation:** Metadata such as classes and methods are stored in the Permanent Generation. Classes that are no longer in use may be garbage collected from the Permanent Generation.

During a full garbage collection event, unused objects in all generations are garbage collected.

**Normal Import & Static import**

We can use normal import to import classes and interfaces of a particular package, whenever we are using normal import, it is not required to use fully qualified name and we can use short names directly.

import java.lang.\*;

We can use static import to import static member of a particular class or interface; whenever we are writing static import it is not required to use class name to access static members and we can access directly

import static java.lang.Integer.sqrt;

**Packages**

Package is group of related things. Package is group of (encapsulation of ) related classes and interfaces. Main purpose of package statements

1.To resolve naming conflicts.

2.Modularity of the application will be improved

3.Maintainability of the application will be improved

4.Security for components will be provided

Universally accepted naming conventions for package

1. Use internet domain in reverse order
2. Use module name
3. Use submodule name
4. Use class name

Example : com.hdfcbank.loan.housing.Account

package com.durgasoft.scjp;

public class Test

{

p s v main(String[] args)

{

Sop(“pkg demo”);

}

}

*javac Test.java* <enter> : generated .class file will be placed in CWD

*javac –d . Test.java* <enter> : generated .class file will be placed in corresponding package structure.

Here d means directory/destination to place .class file

and . means CWD

if the corresponding package structure is not already available then this command itself will create corresponding package structure. As destination instead of .(dot) we can give any valid directory name. example *javac –d F: Test.java* So Test.class will be placed in F:\com\durgasoft\scjp

if the specified directory not available then we will get compile time error. Example *javac –d Z: Test.java*

At the time of execution we have to use fully qualified name *java com.durgasoft.scjp.Test*

Conclusion:

1. In any java source file there can be at most one package statement that is more than one package statements are not allowed otherwise we will get compile time error.
2. In any java program first non comment statement should be package statement( if it is available) otherwise we will get compile time error.

The following is valid java source file structure with same order

Package statement – at most one

Import statements – any number

Class/interface/enums declaration – any number

**Class level Modifiers**

Whenever we are writing our own classes, we have to provide some information about our class to JVM. like where this class can be accessible, anywhere or not; whether child class creation is possible or not ; whether object creation is possible or not; we can specify this information by using appropriate modifiers

The only applicable modifiers for top level classes are : public , default, final, abstract, strictfp. But for inner classes the applicable modifiers are public , default, final, abstract, strictfp + private, protected, static.

**public classes** : If class declared as public then we can access that class from anywhere.

**default classes**: If class declared as default then we can access that class only within the current package. i.e. from outside package we can’t access hence default access is also known as package level access.

**final modifier**: final is the modifier applicable for classes, methods and variables.

Final method: whatever method parent has by default available to the child through inheritance if the child not satisfied with parent method implementation then child is allowed to redefine that method based on it’s requirement this process is called overriding. If parent class method declared as final then we can’t override that method in child class because it’s implementation is Final.

Final class : if a class declared as final we can’t extend functionality of that class i.e. we can’t create child class for that class. i.e. inheritance is not possible for Final classes. Every method declared in Final class is by default final but every variable declared in final class need not be final by default.

Biggest advantage of Final modifier is we can achieve Security but disadvantage is we are missing Inheritance and polymorphism.

**abstract modifier:** abstract is a modifier applicable for classes and methods but not for variables.

abstract method: even though we don’t know about implementation still we can declare a method with abstract modifier i.e. For abstract method only declaration is available but not implementation hence abstract method declaration should end with semicolon(;). Child class is responsible to provide implementation for parent class abstract method.

Ex: public abstract void M1();

By declaring abstract method in the parent class, we can provide guidelines to the child classes such that which method compulsory child has to implement.

class vehicle

{

public abstract int getwheels();

}

class bus extends vehicle

{

public int getwheels()

{

return 7;

}

}

abstract class : for any java class if we are not allowed to create an object(because of partial implementation) such type of class, we have to declare with abstract modifier.i.e for abstract classes instantiation is not possible.

abstract class vs abstract method :

* If class contains at least one abstract method then compulsory we should declare class as abstract otherwise we will get compile time error. Reason is – if class contains at least one abstract method then implementation is not complete and hence it is not recommended to create object, to restrict object instantiation compulsory we should declare class as abstract.
* Even though class doesn’t contain any abstract method still we can declare class abstract if we don’t want instantiation. i.e. abstract class can contain 0 abstract method

Note : - final abstract combination is invalid combination for method as well as classes.

* abstract class can contain final method whereas final class can’t contain abstract method

**strictfp (strict floating point) modifier:** introduced in 1.2 version, we can use strictfp for classes and methods but not for variables.

Usually the result of floating point arithmetic is varied from platform to platform if we want platform independent result for floating point arithmetic then we should go for strictfp modifier.

strictfp method: if a method declared as strictfp all floating point calculations in that method has to follow IEEE 754 standards.

strictfp class: if class declared as strictfp then every floating point calculation present in every concreate method has to follow IEEE 754 standards.so that we will get platform independent results.

Note:strictfp and abstract is illegal combination for method but it’s legal for classes.

**Member (method or variable) level Modifiers:**

**Public members:** if a member declared as public then we can access that member from anywhere but corresponding class should be visible.

**Private members**: if a member declared as private then we can access that member only within class

**Protected members**: if a member declared as protected then we can access that member anywhere in the current package but only in child classes of outside package. Protected = <default> + kids

We can access protected member within current package anywhere either by parent reference or by child reference. But we can access protected member outside package only in child classes and we should use child reference only. I.e. parent reference can’t be used to access protected members from outside package.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Visibility** | **Private** | **Default** | **Protected** | **Public** |
| Within same class | Yes | Yes | Yes | Yes |
| From child class of same package | No | Yes | Yes | Yes |
| From non-child of same package | No | Yes | Yes | Yes |
| From child class of outside package | No | No | Yes( We should use child reference only) | Yes |
| From non-child class of outside package | No | No | No | Yes |

**variables:**

instance variables: if a value of variable varied from object to object then those variables are called instance variable. For every object a separate copy of instance variables will be created. For instance variables we are not required to perform initialization explicitly, JVM will always provide default values.

Now instance variable declared as Final.

1. Compulsory we need to initialize that variable or else we will get compile time error.

2. Compulsory we should initialize that variable before constructor completion. Possible places are below.

2a> at the time of declaration

2b> inside the instance block

2c> inside constructor

static variables: if the value of variable is not varied from object to object such type of variables are not recommended to declare as instance variables, we have to declare those variables at class level by using static modifier. In case of instance variables for every object a separate copy will be created but in case of static variables a single copy will be created at class level and shared by every object of that class. For static variables JVM will always provide default values.

Now static variable declared as Final – 1. Compulsory we need to initialize that variable or else we will get compile time error.

2. Compulsory we should initialize that variable before class loading. Possible places are below

2a> at the time of declaration.

2b> inside static block

local variables: The variables which are declared inside the method, block or constructor are called local variables or temporary or stack or automatic variables. For local variables JVM will not provide default values, we need to initialize those explicitly.

Now local variable declared as Final: 1.before using that variable we need to initialize

2. Only applicable modifier for local variable is final, if we trying to apply other modifiers we will get compile time error

**Static Modifier:** static is the modifier applicable for methods and variables but not for classes. We can’t declare top level class with static modifier however inner classes can be declared as static.

In the case of instance variables, for every object a separate copy will be created however for static variables a single copy will be created at class level and shared by every object of that class. We can’t access instance members directly from static area but we can access from instance area directly. We can access static members from both instance and static area directly.

Case 1 – overloading concept applicable for static methods including main method but JVM can always call String [] args main method only

Example

Class Test {

p s v main(String [] args)

{ sop(“string[]”);}

p s v main(int [] args)

{ sop(“int[]”);}

}

Other overloaded method we have to call just like normal method.

Case 2 – Inheritance concept applicable for static method, including main method hence while executing child class, if child doesn’t contain main method then parent class main method will be executed

Example:

class p

{

p s v main(String args[])

{ sop(“parent main”);}

}

class c extends p

{

}

Case 3 – Method overriding concept is not applicable for static method but method hiding will be applicable.

Inside method implementation if we are using at least one instance variable then that method talks about particular object hence that method is instance method.

Inside method implementation if we are not using any instance variable then this method not related to object then we should declare that method as static.

For static methods implementation should be available whereas implementation is not available for abstract methods hence abstract static combination is illegal for methods.

**Synchronized modifier:** synchronized modifier only applicable for methods and blocks but not for classes and variables.

If a multiple threads trying to operate simultaneously on the same java objects then there may be a chance of data inconsistency problem this is called race condition we can overcome this problem by using synchronized keyword. If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved.

But the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problems hence if there is no specific requirement then it is not recommended to use synchronized keyword.

**Native modifier:** This modifier only applicable for methods. The methods which are implemented in non-java (mostly C or C++) are called native methods or foreign methods. The main objective of native keyword are

1. To improve performance of the system
2. To achieve machine level / memory level communication
3. To use already existing legacy non-java code

Sudeo code to use native keyword in java: 3 steps

1. Load native libraries.
2. Declare native method
3. Invoke a native method

Class Native

{

Static {

System.loadLibrary(‘native library path’); --------------1

}

Public native void M1(); ---------------------------------2

}

Class client

{

P s v main(String args[])

{

Native N = new Native();

N.M1(); -------------------------------------------------3

}

}

Native – abstract combination is illegal.

Native- strictfp combination is illegal – we are not sure if old language follow IEEE 754 standards

Advantage – performance will be improved

Disadvantage of native – breaks platform independence nature of java

**Transient Modifier:** only applicable for variables.

We can use transient in serialization.

At the time of serialization, if we don’t want to save the value of variable to meet security constraint then we should declare that variable as transient. At the time of serialization JVM ignores original value of transient variable and save default value to the file. Hence transient means not to serialize.

**Volatile Modifier:** only applicable for variables.

If the value of variable keep on changing by multiple threads then there may be a chance of data inconsistency problem, we can solve this problem by using volatile modifier. If a variable declared as volatile then for every thread JVM will create a separate local copy. Every modification performed by the thread will take place in local copy, so that there is no effect on the remaining threads.

The main advantage of volatile keyword is we can overcome data inconsistency problem but main disadvantage of volatile keyword is creating and maintaining a separate copy for every thread increases complexity of programming and creates performance problem.

Volatile- final: illegal combination

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Modifier | Classes | | Methods | Variables | Blocks | Interfaces | | Enums | | Constructor |
| Outer | Inner | Outer | Inner | Outer | Inner |
| Public | Y | Y | Y | Y |  | Y | Y | Y | Y | Y |
| Private |  | Y | Y | Y |  |  | Y |  | Y | Y |
| Protected |  | Y | Y | Y |  |  | Y |  | Y | Y |
| Default | Y | Y | Y | Y |  | Y | Y | Y | Y | Y |
| Final | Y | Y | Y | Y |  |  |  |  |  |  |
| Abstract | Y | Y | Y |  |  | Y | Y |  |  |  |
| Static |  | Y | Y | Y | Y |  | Y |  | Y |  |
| Synchronized |  |  | Y |  | Y |  |  |  |  |  |
| Native |  |  | Y |  |  |  |  |  |  |  |
| Strictfp | Y | Y | Y |  |  | Y | Y | Y | Y |  |
| Transient |  |  |  | Y |  |  |  |  |  |  |
| Volatile |  |  |  | Y |  |  |  |  |  |  |

**Inheritance (IS-A) vs. Composition (HAS-A) Relationship:**

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the implementation of inheritance (IS-A relationship), or object composition (HAS-A relationship). Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition.

**IS-A Relationship:**

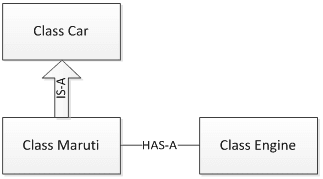
In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

**HAS-A Relationship:**

Composition(HAS-A) simply mean the use of instance variables that are references to other objects. For example Maruti has Engine, or House has Bathroom.

Let’s understand these concepts with an example of Car class.



**Interfaces**

**Introduction-** What is Interface: Definition 1 - Any service requirement specification (SRS) is considered as an Interface.

Ex 1: JDBC API contains/access requirement specification to develop database driver, database vendor(Oracle/MySQL) is responsible to implement this JDBC API.

Ex 2: Servlet API contains/access requirement specification to develop webserver, webserver vendor(Apache tomcat server/Oracle weblogic server) is responsible to implement servlet API.

Definition 2 – from client point of view, interface defines the set of services what he is expecting. From service provider point of view, an interface defines a set of services what he is offering. Hence any contract between client and service provider is considered as Interface.

Definition 3 – Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summary Definition: Any service requirement specification or any contract between client and service provider or 100% pure abstract class is nothing but interface.

**Interface declaration & implementation:**

1. Whenever we are implementing an interface, for each and every method of that interface we have to provide implementation.
2. Every interface method is always public and abstract whether we are declaring or not hence whenever we are implementing an interface method, compulsory we should declare as public otherwise we will get compile time error

Extends Vs Implements

1. A class can extend only one class at a time Ex: class A extends B
2. An interface can extend any number of interfaces simultaneously Ex: interface A extends B,C
3. A class can implement any number of interfaces simultaneously. Ex: class A implements A,B,C
4. A class can extend another class and can implement any number of interfaces simultaneously Ex: class A extend B implements C,D,E

Interface methods: every method inside interface is public and abstract. Why public: to make the method available to every implementation class

Why Abstract: that method is just specification and implementation will be in the class.

Also we can’t declare interface method with: private,protected,final,static,synchronized,native,strictfp.

Interface variables: an interface can contain variables, the main purpose is to define requirement level constants. Every interface variable is always public,static,final. Why?

public– to make this variable available to implementation class

static - Without exist of object, implementation class could access this variable

final - If one implementation class changes value then remaining implementation classes will be impacted so to restrict this we have always final.

For interface variables, compulsory we have to initialize that variable at the time of declaration.

Interface Naming conflicts

1. Method naming conflicts –

Case 1 – If 2 interfaces contains same method with same signature and return type then in the implementation class, we have to provide implementation for only one method.

Case 2 – if 2 interfaces contains a method with same name, but different argument types then in the implementation class, we have to provide implementation for both methods and these methods access overloaded methods.

Case 3 – if 2 interfaces contains a method with same signature but different return types then it is impossible to implement both interfaces simultaneously if return types are not co-variant)

interface Left

{

Public void m1();

}

Interface right

{

Public int m1();

}

Here we can’t write any java class which implements both interfaces simultaneously.

Is a java class can implement any number of interfaces simultaneously?

Ans: Yes, but not in a particular case – if two interfaces contains a method with same signature but different return types then it is impossible to implement both interface simultaneously.

2.Variable naming conflicts –

Two interfaces can contain a variable with same name and there may be chance of variable naming conflicts but we can solve this problem by using interface names.

Interface Left

{

Final static Int x = 777;

}

Interface Right

{

Final static Int x = 888;

}

Class test implements Left,Right

{

P S V Main(String[] args){

SOP(Left.x);

SOP(Right.x); }

}

**Marker Interface**

If an interface doesn’t contain any methods and by implementing that interface if our objects will get some ability such type of interfaces called Marker interfaces or Ability interface or Tag interface.

For Examples : Serializable(I), Clone able(I), RandomAcees(I), SingleThreadModel(I) - these are marked for some ability.

Example 1 – by implementing serializable interface our objects can be saved to the file and can travel across the network.

Example 2 – by implementing Clonable interface our objects are in position to produce exactly duplicates cloned objects.

Without having any method how objects will get some ability in marker interfaces?

Ans: Internally JVM is responsible to provide required ability.

Why JVM is providing required ability in marker interfaces?

Ans: To reduce complexity of programming and to make java language as simple.

Is it possible to create our own Marker interface?

Ans: Yes but customization of JVM must be required.

**Adaptor Classes**

Adaptor class is a simple java class that implements an interface with only empty implementation.

Interface X

{ public void m1();

Public void m2();

Public void m3();

……

Public void m1000();

}

abstract class AdapterX implements X

{

m1{ }

m2{ }

m3{ }

……….

m1000{ }

}

If we implement an interface, for each and every method of that interface compulsory we should provide implementation whether it’s required or not. The problem in this approach is it increases length of the code and reduces readability, we can solve this problem by using Adaptor classes.

Instead of implementing interface, if we extend adaptor class, we have to provide implementation only for required methods and we are not responsible to provide implementation for each and every method of interface so that length of code will be reduced.

class Test extends AdapterX

{ ………………

m3{ --- }

}

class Sample extends AdapterX

{…………….

m80{ }

}

Example : We can develop servlet 3 ways. 1. Using servlet interface 2. By extending Generic Servlet 3.By extending HTTP Servlet.

If we implement servlet interface, for each and every method of interface we should provide implementation which is tedious, instead of implementing servlet interface directly if we extend generic servlet, we have to provide implementation only for servlet methods and for all remaining methods we are not required to provide implementation hence more or less generic servlet access adaptor class for servlet interface.

Note – Marker interface and Adaptor classes simplifies complexity of programming and these are best utilities to the programmer.

Interfaces Vs Abstract Class Vs Concreate class

If we don’t know anything about implementation just we have requirement specification then we should go for interface. Ex: Servlet

If we are talking about implementation but not completely(partial implementation) then we should go for Abstract class ex: GenericSevlet, HTTPServlet

If we are talking about implementation completely and ready to provide service then we should go Concreate class. Ex: MyOwnServlet

Deference between Interfaces & abstract class:

|  |  |
| --- | --- |
| **Interface** | **Abstract class** |
| If we don’t know anything about implementation and just we have requirement specification then we should go for Interface | If we are talking about implementation but not completely(Partial Implementation) then we should go for abstract class. |
| Inside interface every method is always public and abstract whether we are declaring or not hence Interface is considered as 100% pure abstract class | Every method present in abstract class need not be public and abstract and we can take concreate methods also |
| As every interface method is always Public and Abstract. So we can’t declare that method with private,protected,final,static,synchronized,native strictfp modifiers | There are no restrictions on abstract class method modifiers. |
| Every variable present inside interface is always public / static / final , whether we are declaring or not | Every variable present inside abstract class need not be public / static / final |
| As every interface variable is always pubic / static / final, we can’t declare that variable with Private / Protected / volatile / transient modifiers | There is no any restrictions on abstract class variables modifiers |
| For interface variable, we should initialize that variable at the time of declaration otherwise we will get compile time error | For abstract class variables we are not required to perform initialization at the time of declaration. |
| Inside interface we can’t declare static and instance blocks | Inside abstract class we can declare static and instance blocks |
| Inside interface we can’t declare constructors | Inside abstract class we can declare constructor |

We can’t create object for abstract class, but abstract class can contain constructor, what is the need?

Ans: Abstract class constructor will be executed whenever we are creating child class object to perform initialization of child class object.

Anyway we can’t create object class for abstract class and interface but abstract class can contain constructor but interface doesn’t contain constructor, What is the reason?

Ans: the main purpose of constructor is to perform initialization of instance variables. Abstract class can contain instance variables which are required for child objects. To perform initialization of those instance variables constructor is required for abstract class but every variable present inside interface is always public / static / final and there is no chance of instance variables hence constructor concept is not required for interfaces.

Inside interface every method is always abstract and we can take only abstract methods in abstract class only then what is difference between interface and abstract class? Or is it possible to replace interface with abstract class?

Ans: We can replace interface with abstract class but it is not good programming practice, this is something like recruiting IAS officer for sweeping activity. If everything is abstract then it is highly recommended to go for interface.

**New features added in interfaces in JDK 8**

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

// An example to show that interfaces can

// have methods from JDK 1.8 onwards

interface in1

{

    final int a = 10;

    default void display()

    {

        System.out.println("hello");

    }

}

// A class that implements interface.

class testClass implements in1

{

    // Driver Code

    public static void main (String[] args)

    {

        testClass t = new testClass();

        t.display();

    }

}

1. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

|  |
| --- |
| // An example to show that interfaces can  // have methods from JDK 1.8 onwards  interface in1  {      final int a = 10;      static void display()      {          System.out.println("hello");      }  }    // A class that implements interface.  class testClass implements in1  {      // Driver Code      public static void main (String[] args)      {          in1.display();      }  } |

**OOPS Concepts**

Data Hiding: outside person should not access our internal data directly or our internal data shouldn’t go out directly this feature is data hiding. By declaring data members as private we can achieve data hiding.

Abstraction: Hiding the internal implementation and just highlight the setup services we are offering is the concept of Data Abstraction.by using Interfaces and Abstract classes we can implement Data abstraction.

Encapsulation: The process of binding data and corresponding methods into a single unit is nothing but encapsulation. Example of encapsulation is class.

Encapsulation = data hiding + abstraction

Inheritance : for inheriting the parents method, main advantage is Reusability.

Class P

{

Public void m1()

{ SOP(“Parent”);}

}

Class C extends P

{

Public void m2()

{ SOP(“Child”);}

}

Class Test

{

P S V main(String args[])

{

1 - > P P1 = new P();

P1.m1(); - Ok

P1.m2(); - CE

2-> C C1 = new C();

C1.m1(); Ok

C1.m2(); Ok

3-> P P1 = new C();

P1.m1(); Ok

P1.m2(); CE

4-> C C1 = new P(); CE

}

}

Java API based on Inheritance only.

Multiple inheritance: A java class can’t extend more than one class at a time, hence Java won’t provide support for multiple inheritance in class.

Note: Java class can provide support to multi-level inheritance.

Why Java won’t provide support for multiple inheritance?

Ans: there may be a chance of ambiguity problem.

Note: Cyclic inheritance is not allowed and also not required in Java

**Polymorphism: one name multiple forms**

**Method Overloading**

Two methods are said to be overloaded if and only if both methods having same name but different argument types.

Using method overloading we are getting more flexibility and reduces complexity. In method overloading, method resolution is always taken care by compiler so method overloading also called as compile time polymorphism / Static Binding / Early Binding

Case 1: Automatic promotion in overloading: while resolving overloaded methods, if exact matched method is not available then we won’t get any compile time error immediately. First it will promote argument to next level and check whether matched method is available or not. If matched method is available then it will be considered and if the matched method is not available then compiler promotes argument once again to the next level. This process will be continued until all possible promotions. At last if matched method is not available then we will get compile time error. The following are all possible promotions in overloading.

Byte -> Short ->

Int -> long - > float - > double

Char ->

Example

Class Test

{

Public void m1(int i){ SOP(“int-args”);}

Public void m1(float f){ SOP(“float-args”);}

P s v main(String args[])

{

Test T1 = new Test();

T1.m1(10); -------int-args

T1.m1(10.5f);-------float-args

T1.m1(‘a’); ---------int-args

T1.m1(10l);--------float-args

T1.m1(10.5);--------CE-can’t find method m1(double)

}

}

Case 2: While resolving overloaded methods, compiler will always gives precedence for child type argument when compared with parent type argument

Example

Class Test

{

Public void m1(String S){ SOP(“String-args”);}

Public void m1(Object O){ SOP(“Object-args”);}

P s v main(String args[])

{

Test T1 = new Test();

T1.m1(new Object()); -------Object-args

T1.m1(“Durga”);-------String-args

T1.m1(null); ---------String-args: null is applicable for String and Object but String is child of Object class so child type is getting priority.

}

}

Case 3: While resolving overloaded methods, compiler will always gives precedence for child type argument but if there are more than 1 methods matching at same child level (ex String and StringBuffer) then we will get compile time error of Ambiguity.

Example

Class Test

{

Public void m1(String S){ SOP(“String-args”);}

Public void m1(StringBuffer S){ SOP(“StringBuffer-args”);}

P s v main(String args[])

{

Test T1 = new Test();

T1.m1(“Durga”);-------String-args

T1.m1(new StringBuffer()); ---------StringBuffer-args

T1.m1(null);------------------------CE -NULL is valid for both String and StringBuffer and both classes at same level so we will get CE as Ambiguous.

}

}

Case 4:

Class test

{

Public void m1(int I, float f){SOP(“int-float”);}

Public void m1(float f,int i){ SOP(“flaot-int”);}

P S V Main(String[] args)

{ test t1 = new test();

T1.m1(10,10.5f); - OK int-float

T1.m1(10.5f,10); - OK float-int

T1.m1(10,10) ----------CE:Ambiguity as both method matched

T1.M1(10.5f,10.5f)--------CE- No method found as no method match

}

}

Case 5:

Class Animal

{ }

Class Monkey extends Animal

{ }

Class test

{

Public void m1(Animal a)

{ SOP(“Animal version”);}

Public void m1(Monkey m)

{ SOP(“monkey version”);}

P S V main(String[] args)

{ test t1 = new test()

Animal a = new Animal();

T1.m1(a);----------------------------🡪 animal version

Monkey m = new Monkey();

T1.m1(m);-------------------------🡪Monkey version

Animal a1 = new Monkey();

T1.m1(a1);-----------------------🡪Animal version

}

}

Note: in method overloading, compiler is responsible to perform method resolution based on the reference types, run time object never play a role in method overloading.

**Method Overriding**

Whatever methods parents has by default available to child through inheritance if a child class not satisfied with parent class implementation then child is allowed to redefine that method based on it’s requirement this process is called overriding.

In Method overriding, method resolution is always based on run time object so method overriding always called as run time polymorphism / dynamic polymorphism / late binding.

Rules for overriding:

1. In overriding method names and argument types must be matched. i.e. method signature must be same.
2. In overriding return type must be same but this rule is applicable until 1.4 version only, from 1.5 version onwards we can take co-variant return types. According to this child class method return type need not be same as parent method return type, it’s child type also allowed.

Note: Co-variant concept is applicable for object types but not applicable for primitive types.

1. Method overriding is not applicable for private methods of parent’s class. Based on our requirement we can define exactly same private method in child class; it is valid but not overriding.
2. If parent class method is final then we can’t override that method in child class.
3. If parent class method is abstract then we should override that method in child class to provide implementation.
4. We can override non abstract method as abstract. Advantage of this approach is we can stop the availability of parent method implementation to the next level child classes.
5. In method overriding, modifiers like synchronized, native, strictfp won’t keep any restriction.
6. While overriding we can’t reduce scope of access modifiers but we can increase scope of access modifiers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Parent class method** | <default> | Protected | Public | Private |
| **Child class method** | <default>/Protected/Public | Protected/ Public | Public | NOT ALLOWED |

1. In method overriding- If child class method throws any checked exception, compulsory parent class method should throw same checked exception or it’s parent otherwise we will get compile time error but there are no restrictions for un-checked exceptions.

Overriding with respect to Static methods:

1. We can’t override static method as non-static otherwise we will get compile time error.
2. Similarly we can’t override a non-static method as static
3. If both parent and child class methods are static then we won’t get any compile time error. It seems, overriding concept applicable for static method but it is not overriding but it is method hiding.

Method hiding – all rules of method hiding exactly same as method overriding except following differences

|  |  |
| --- | --- |
| Method Hiding | Method overriding |
| Both parent and child class methods should be static | Both parent and child class methods should be non-static |
| Compiler is responsible for method resolution based on reference type | JVM is responsible for method resolution based on run time objects |
| Know as – Compile time polymorphism / Static polymorphism/ early binding | Knows as – Run time polymorphism / dynamic polymorphism / late binding |

Overriding with respect to var-arg method:

1. We can override var-arg method with another var-arg method only if we are trying to override with normal method then it will become overloading but not overriding.

Overriding with respect to variables:

Variable resolution is always takes care by compiler based on reference type, irrespective of whether the variable is static or non-static.(overriding concept applicable only for methods but not for variables)

**Differences between overloading and overriding.**

|  |  |  |
| --- | --- | --- |
| ***Property*** | ***Overloading*** | ***Overriding*** |
| ***Method Names*** | Must be same | Must be same |
| ***Argument types*** | Must be different(at least order) | Must be same including order |
| ***Method signatures*** | Must be different | Must be same |
| ***Return types*** | No restrictions | Must be same until 1.4 version but 1.5 onwards co-variant return types allowed |
| ***Private, Static, Final methods*** | Can be overloaded | Can’t be overridden |
| ***Access modifiers*** | No restrictions | The scope of access modifier can’t be reduced but we can increase |
| ***Throws class*** | No restrictions | If a child class method throws any checked exception compulsory parent class method should throw same checked exception or it’s parent but no restriction for non-checked exception |
| ***Method resolution*** | Always takes care by compiler based on reference type | Always takes care by JVM based on run time objects. |
| ***Knows as*** | Compiler time polymorphism / Static binding / early binding | Run time polymorphism / dynamic binding / late binding |

Parent reference and child reference

|  |  |
| --- | --- |
| Child C = new child()  Ex: ArrayList L = new ArrayList() | Parent P = new Child()  Ex: List L = new ArrayList() |
| We can use this approach if we know exact run time type of object | We can use this approach if we don’t know exact run time type of object |
| Advantage : By using child reference, we can call both parent and child class methods | Disadvantage : By using parent reference, we can call only methods available in parent class and we can’t call child specific methods |
| Disadvantage: We can use child reference to hold only particular child class object | Advantage: we can use parent reference to hold any child class object |

Summary Note:

3 pillars of OOP’s: Encapsulation & Polymorphism & Inheritance

Encapsulation: talks about Security

Polymorphism: talks about flexibility

Inheritance: talks about reusability.

**Coupling**

To measure the dependency of component we have coupling. Only 2 levels of coupling – loosely coupled and tightly coupled.

If dependency is more – tightly coupling

If dependency is less – loosely coupling

Example:

Class A

{

Static int I = B.j;

}

Class B

{

Static int j = C.k;

}

Class C

{

Static int k = D.m1();

}

Class D

{

Public static int m1()

{ return 10; }

The above components are said to be tightly coupled with each other because dependency between components is more. Tightly coupling is not good programming practice because it has several serious disadvantages. Without affecting remaining components we can’t modify any component and hence enhancement will become difficult

Also it suppress reusability and reduces maintainability of the application. Hence we have to maintain dependency between the components as less as possible

i.e. Loosely coupling is good programming practice.

**Cohesion**

For every component clear, well defined functionality is defined then that component is said to be follow high cohesion.

High cohesion is always good programming practice because it has several advantages

1. Without affecting remaining components we can modify any component hence enhancement will become easy
2. It promotes reusability of the code.
3. It improves maintainability of the application

**Object Type Casting**

We can use parent reference to hold child objects

Ex: Object O = new String(“Durga”) ---- Valid

We can use interface reference to hold implemented class object

Ex: Runnable R = new Thread(); ----Valid

A b = ( C ) d;

Mantra 1 (Compile time checking 1) : The type of ‘d’ and ‘C’ must have some relation either child to parent or parent to child or same type otherwise we will get compile time error saying inconvertible types found D type required C

Example 1:

Object O1 = new String(“Durga”);

StringBuffer sb = (StringBuffer) O1;

Example 2:

String S = new String(“Durga”);

StringBuffer sb = (StringBuffer) S; // here StringBuffer and String are having no relation so compile time error

Mantra 2 (Compile time checking 2): ‘C’ must be either same or derived type of A. other wise we will get compile time error saying incompatible types found ‘C’ required ‘A’

Example 1:

Object O1 = new String(“Durga”);

StringBuffer sb = (StringBuffer) O1;

Example 2:

Object O1 = new String(“Durga”);

StringBuffer sb = (String) O1; // here String and Object O1 has relation so first check is passed but for second check String and StringBuffer don’t have parent child relation i.e String is not derived type of StringBuffer so compile time error.

Mantra 3(Run time checking 1): Run time object type of ‘d’ must be either same or derived type of ‘C’ otherwise we will get runtime exception saying “ClassCastException”

Example 1:

Object O1 = new String(“Durga”);

StringBuffer sb = (StringBuffer) O1;

//RunTime Exception: ClassCastException: J.L.String can’t be cast to J.L.StringBuffer

Example 2:

Object O1 = new String(“Durga”);

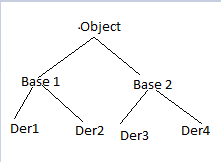
String O2 = (String) O1;

//Perfectly valid : 1st check – O1 type and String are in relation

2nd check: String and String are of same type

3rd check: O1’s run time object type is String which is same as String.

Examples



Base2 b = new Der4();

1. Object O = (Base2) b; // Ok
2. Object O = (Base1) b; // CE – inconvertible types
3. Object O = (Der3) b; // RE – ClassCast Exception
4. Base2 b1 = (Base1) b; // CE - inconvertible types
5. Base1 b1 = (Der4)b; // Incompatible types

Strictly speaking through type casting we are not creating any new object, for the existing object we are providing another type of reference variable i.e. we are performing type casting but not object casting.

String S = new String(“Durga”);

Object O = (Object) S;

Which will same as Object O = new String(“Durga”)

Example 1

P contain M1 method

C contain M2 method

C extends P

C c = new C();

Out of below which are valid?

c.m1() // OK

c.m2() // OK

((P)c).m1() // Ok ..same as P p1 = new C() so parent can access m1 method

((P)c).m2() // CE because – same as P p1 = new C() so parent reference can’t call child specific method.

Example 2

A contains m1(){SOP(“A”);}

B contains m1(){SOP(“B”)}

C contains m1(){SOP(“C”)}

B extends A

C extends B

Then

C c = new C();

c.m1()------// C

((B)c).m1()------//C

((A)((B)c).m1()-----//C

In all the cases C will display because in overriding method resolution is always based on run time object type so here run time object type is C. But if all method are static then result will be C,B,A respectively because for static method resolution based on reference objects.

Example 3

A contain X = 777

B contain X = 888

C contain X = 999

B extends A

then

C extends B

C c = new C();

SOP(c.X) -----------999

SOP(((B)c).X)-------888

SOP((A((B)c)).X)--------777

Because for variable resolution is always based on reference types.

**Static Control flow**

Whenever we are executing a java class the following sequence of steps will be executed as the part of static control flow.

1. Identification of static members from top to bottom [ 1 to 6 steps]
2. Execution of static variables assignments and static blocks from top to bottom [7 to 12]
3. Execution of main method [13 to 15]

Class base

{

Static int I = 10;----1; I=10 ----7

Static---2

{

M1();--------8

SOP(“FIsrt static”);--------10

}

P S V Main(String args[])----3

{ m1();------------13

SOP(“main method”);}---------15

P s v m1()-----4

{

SOP(j);---------9,14

}

Static------5

{

SOP(“second static”);------11

Static int j =20;----6 j = 20--------------12

}

}

O/p

*0*

*First static*

*Second static*

*20*

*Main method*

Read Indirectly Write only (RIWO) : Inside a static block if we are trying to read variable that read operation is called direct read, If we are calling a method and within that method if we are trying to read a variable that read operation is called Indirect read.

Class Test

{ static int I = 10;

Static

{

M1();

SOP(I); ///direct read

}

P s void m1()

{ SOP(I) //indirect read

}

}

If a variable is just identified by the JVM and original value is not yet assigned then the variable is said to be in READ INDIRECTLY WRITE ONLY (RIWO)state. If variable is in RIWO state then we can’t perform direct read but we can perform indirect read. If we are trying to read directly then we will get compile time error saying illegal forward reference.

Examples

Class Test

{ static int x = 10;

Static

{ SOP(x); }

}

O/p : 0 and RE – for main method

Class Test

{

Static

{ SOP(x); }

Static int x = 10;

}

O/p : illegal format reference

Class Test

{

Static

{ m1(); }

P s void m1(){SOP(x);}

Static int x = 10;

}

o/p ; 0 and RE – for main method

**Static Block :**  static blocks will be executed at the time of class loading hence at the time of class loading we want to perform any activity we have to define that inside static block.

Example 1 : At the time of java class loading corresponding native libraries should be loaded hence we have to define this activity inside static block.

Class test

{

Static

{

System.LoadLibrary(“native library path”)

}

}

Example 2 : After loading every database driver class, we have to register driver class with driver manager but inside database driver class there is a static block to perform this activity. And we are not responsible to register explicitly.

Class DbDriver

{

Static

{ Register this driver with driver manager }

}

Note : within a class we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

Q.1 – Without writing main method is it possible to print some statements to console?

- Yes we can print by static block

Class Test

{ static

{ SOP(“Hello”);

System.exit(0);

}

}

Q.2 : without writing and static block, Is it possible to print some statement in console?

Yes we can do it via multiple ways.

Class Test

{

Static int I = m1();

Static int m1()

{ SOP(“Print”);

System.exit(0);

Return 10;

}

}

Class Test

{ static test t = new Test();

{

SOP(“Hello”);

System.exit(0);

}

}

Class Test

{

Static test t = new Test();

Test()

{ SOP(“Hello”);

System.exit(0);

}

}

Note: from 1.7 version onwards main method is mandatory to start a program execution hence from 1.7 version onwards without writing main method it is impossible to print some statements to the console.

**Static control flow in parent to child relationship:**

Whenever we are execution child class the following sequence of events will be executed automatically as part of static control flow

1. Identification of static members from parent to child
2. Execution of static variables assignments and static blocks from parent to child
3. Execution of only child class main method

**Instance control flow**

Whenever we are executing a java class, first static control flow will be executed in the static control flow, if we are creating a object the following sequence of events will be executed as the part of instance control flow.

1. Identification of instance members from top to bottom
2. Execution of instance variables assignments and instance blocks from top to bottom
3. Execution of constructor

Note: static control flow is one time activity which will be performed at the time of class loading but instance control flow is not one time activity and it will be performed for every object creation. Object creation is most costly operation if there is no specific requirement then it is not recommended to create object

**Instance control flow in parent to child relationship**

Whenever we are creating child class object the following sequence of events will be performed automatically as the part of instance control flow

1. Identification of instance members from parent to child
2. Execution of instance variable assignments and instance blocks only in parent class
3. Execution of parent constructor
4. Execution of instance variables assignments and instance block in child class
5. Execution of child constructor

Q-1 In how many ways, we can create an object in java? Or in how many ways we can get object in Java?

Ans – 1 By using new operator : Test T = new Test()

2.By using newInstance() method : Test T = (Test) class.forName(“Test”).newInstance();

3.By using factory method: Runtime R = Runtime.getRuntime()

DateFormat Date = DateFormat.getInstance()

4.by using clone method : Test T1 = new Test()

Test T2 = (Test) t1.clone()

5.By using deserialization

FileInputStream f = new FIS(“abc.ser”);

ObjectInputStream O = new OIS(f);

Dog d2 = (Dog) dis.readObject();

**Constructors**

Once we creates objects compulsory we should perform initialization then only the object is in position to respond properly

Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of the object, this piece of the code is nothing but constructor. Hence the main purpose of constructor is initialization of object.

Class Student

{ String name;

Int rollno;

Student(String S,int i);

{

This.name = s;

This.rollno = I;

}

P S V main(String [] args)

{ Student s1 = new Student(“VBK”,101);

Student s2 = new Student(“VBH”,102);

}

Note: The main purpose of constructor is initialization of object but not to create object.

**Constructor Vs Instance block**

The main purpose of constructor is to perform initialization of an object but other than initialization if we want to perform any activity for every object creation then we should go for instance block(like. Updating one entry in DB for every object creation or incrementing count for every object creation)

Both constructor and instance block have their own different purposes and replacing one concept with another concept may not work always.

Both constructor and instance block will be executed for every object creation but instance block first followed by constructor.

**Rules of writing constructor**

1. Name of the class and name of constructor must be matched.

2. Return type concept is not applicable for constructor even void also

3. The applicable modifiers for constructors are – public / private / protected / Default.

**Default Constructor**

If our class doesn’t contain any constructor then compiler will generate Default constructor.

Prototype: 1. It is always no arg constructor

2.The access modifier of default constructor is exactly same as access modifier of class (This rule is applicable only for public and default)

3.It contains only one line : super(); it is no argument call to super class constructor

|  |  |
| --- | --- |
| Programmers code | Compilers code |
| Class Test  {  } | Class Test  { Test()  { super();  }  } |
| Public class Test  {  } | Public class Test  { public test()  { super();  }  } |
| Public class Test  {  Void test()  {  }  } | Public class Test  {  Public test()  { super();  }  Void test() { }  } |
| Class Test  {  Test()  {  }  } | Class Test  {  Test()  {  Super();  }  } |
| Class Test  {  Test(int i)  { super();}  } | Class Test  {  Test(int i)  { super();}  } |
| Class test  {  Test() { this(10) }  Test(int i) { }  } | Class test  {  Test() { this(10) }  Test(int i) { super(); }  } |

The first line inside every constructor should be either Super() or this() and if we are not writing anything then compiler will always place super();

Case 1:

We can take super() or this only in first line of constructor or else we will get compile time error

Case 2:

Within the constructor we can take either super() or this but not both simultaneously

Case 3:

We can use super() or this only inside constructor, if we are trying to use outside of constructor we will get compile time error

|  |  |
| --- | --- |
| Super () / this() | Super / this |
| These are constructors calls to perform calling super class and current class constructors | These are keywords to refer super class and current class instance members |
| We can use only in constructor as a first line | We can use anywhere except static area |
| We can use only once in constructor | We can use any number of times |

**Overloaded constructor**

Within a class we can declare multiple constructors and all these constructors having same name but different type of arguments hence all these constructors are considered as overloaded constructors. Hence overloading concept applicable for constructors.

For constructors inheritance and overriding concepts are not applicable but overloading concept is applicable. Every class in java (even abstract) contain constructor but interface can’t contain constructor.

Case 1:

Recursive method call (one method calling each other) is a runtime exception saying stack overflow error. But in our program if there is chance of recursion constructor (one constructor calling each other) invocation then we will get compile time error.

Case 2:

Class p

{ }

Class C extends P

{ }

// compile will be fine

Class p

{ p() { }

}

Class c extend P

{ }

// compile will be fine

Class P

{ P(int i) { }

}

Class C extends P

{ }

//CE – can’t find symbol constructor p() in class P

Conclusions: 1.If parent class contains any argument constructor then while writing child classes we have to take special care with respect to constructors.

2. Wherever we are writing any argument constructor it is highly recommended to write no arg constructor also

Case 3: if a parent class constructor throws any checked exception compulsory child class constructor should throw same checked exception or its parent otherwise the code won’t compile

Which of the following is valid?

1. The main purpose of constructor is to create an object – In valid
2. The main purpose of constructor is to perform initialization of an object – valid
3. Name of constructor should same as class name – valid
4. Return type concept applicable for constructors but only void – invalid
5. We can apply any modifier for constructor – invalid
6. Default constructor generated by JVM – false
7. Compiler is responsible to generate default constructor – True
8. Compiler will always generate default constructor- false
9. If we are not writing no arg constructor then compiler will generate default constructor : invalid
10. Every no arg constructor is always default constructor – false
11. Default constructor is always no arg constructor - True
12. The first line inside every constructor should be either super() or this(), if we are not writing anything then compiler will generate this() – false
13. For constructor both overloading and overriding concept are applicable – false
14. For constructors inheritance concept applicable but not overriding – false
15. Only concreate classes can contain constructor but abstract classes can’t – false
16. Interface can contain constructors – false
17. Recursion constructor invocation is run time exception – false
18. If a parent class constructor throws some checked exception the compulsory child class constructor should throw same checked exception or child - false

**Singleton classes:**

For any java class, if we allowed to create only one object such type of class is called singleton class. Example: Runtime, Business Delegate, Service Locator

Advantage: if several people have same requirement then it is not recommended to create separate object for every requirement, we have to create only one object and we can reuse same object for every similar requirement so that performance and memory utilization will be improved. This is the central idea of singleton classes.

Example

Runtime r1 = Runtime.getRuntime();

Runtime r2 = Runtime.getRuntime();

.

.

Runtime r1000 = Runtime.getRuntime();

How to create our own singleton classes?

Ans –

We can create our own singleton classes for this we have to use private constructor and private static variable and public factory method

**Exception Handling**

**Introduction**

An unexpected, unwanted event that disturbs normal flow of program is called exception. Ex: TypePuncturedException, SleepingException , FilenotFoundException.

It is highly recommended to handle exceptions and the main objective of exception handling is graceful termination of the program.

Exception handling doesn’t mean repairing an exception we have to provide alternative way to continue rest of the program normally, this is the concept of exception handling

For example : our program requirement is to read data from remote file locating at London at runtime if London file is not available our program should not be terminated abnormally. We have to provide some local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

Try

{

Read data from remote data located in London

}

Catch (FileNotFoundException)

{ use the local file and continue rest of the program normally

}

**Run time stack mechanism**

For every thread JVM will create a run time stack. Each and every method call performed by that thread will be stored in the corresponding stack. Each entry in the stack is called stack frame or activation record. After completing every method call the corresponding entry from the stack will be removed. After completing all the method calls that stack will become empty and that empty stack will be destroyed by JVM just before terminating the thread.

Class Test

{

P S V Main(String [] args)

{

Dostuff();

}

P S V Dostuff()

{

Domorestuff();

}

P S V Domorestuff()

{

SOP(“Hello”);

}

}

|  |
| --- |
|  |
|  |
| DoMoreStuff() |
| Dostuff() |
| Main() |

Runtime stack for main thread.

**Default exception handling in JAVA**

1. Inside a method if any exception occurs, the method in which it is raised is responsible to create exception object by including following information.1.name of exception 2. Description of exception 3.location at which exception occurs(stack trace)
2. After creating exception object method handovers that object to the JVM
3. JVM will check whether the method contains any exception handling code or not. If the method doesn’t contain exception handling code then JVM terminates that method abnormally and removes corresponding entry from the stack
4. Then JVM identifies caller method and checks whether caller method contains any handling code or not, if the caller method doesn’t contain handling code then JVM terminates that caller method also abnormally and removes corresponding entry from the stack, this process will be continued until main method. And if the main method also doesn’t contain handling code then JVM terminates main method also abnormally and removes corresponding entry from the stack.
5. JVM handovers responsibility of exception handling to default exception handler, which is the part JVM
6. Default exception handler prints exception information in the following format and terminates program abnormally

Exception in thread XXXX: Name of the exception : Description : stack trace

Class Test

{

P S V main(String [] args)

{ Dostuff(); }

P S V Dostuff()

{ Domorestuff(); }

P S V Domorestuff()

{ SOP(10/0);}

}

Note: In a program if at lease one method terminates abnormally then program termination is abnormal termination, if all methods terminated normally then only program termination is normal termination.

**Exception Hierarchy**

Throwable class access ROOT for the java exception hierarchy, throwable class defines 2 child classes – Exception and Error

Exception: Most of the times, exceptions are caused by our program and these are recoverable for example

Try

{

Read data from remote data located in London

}

Catch (FileNotFoundException)

{ use the local file and continue rest of the program normally

}

Errors: Most of the times errors are not caused by our programs and these are due to lack of system resources. Errors are non recoverable. Example : OutOfMemoery error occurs being a programmer we can’t do anything and the program will be terminated abnormally.

System Admin or Server Admin is responsible to increase heap memory

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Throwable | | | | | |
| Exception | | | | Error | |
| RunTimeException | IOException | ServletException | RemoteException……. | VMError | AssertionError….. |

**Checked and Unchecked exception:**

Checked Exception Vs Unchecked exceptions

1. The exceptions which are checked by compiler for smooth execution of the program are called checked exceptions Ex: HallTicketMissingException / PenNotWorkingException/FileNotFoundException. In our program if there is chance of raising checked exception then compulsory we should handle that checked exception either by try catch or by throws keyword otherwise we will get compile time error.
2. The exceptions which are not checked by compilers, whether programmer handling or not. such type of exceptions are called unchecked exceptions Ex: AirthmeticException, BombBlastException etc.

Note: 1.Whether it is checked or unchecked every exception occurs at Runtime only, there is no chance of occurring any exception error at compile time.

2.RuntimeException and it’s child classes, Error and it’s child classes are unchecked, except these remaining are checked exceptions.

**Customized exception handling using try, catch.**

It is highly recommended, we should handle exception in our program. The code in which may raise exception, called as risky code, then we have to define that code in try block and corresponding handling code, we have to define inside catch block.

Try

{ Risky Code }

Catch(Exception e)

{ Handling code }

Throwable class defines the following methods to print exception information –

|  |  |
| --- | --- |
| **Method** | **Printable Format** |
| PrintStackTrace() | Name of the Exception : Description  Stack trace |
| toString() | Name of the Exception : Description |
| getMessage() | Description |

**Try with Multiple catch blocks**

Try

{ Risky Code }

Catch(AirthmaticException e)

{ //perform alternative arithmetic }

Catch(SqlException e)

{ //try Mysql instead of Oracle }

Catch(FileNotFoundException e)

{ //Use local file }

Catch(Exception e)

{ //Use default handling }

Note : If try with multiple catch block present then the order of catch block is very important, we have to take child first and then parent otherwise we will get compile time error saying Exception XXX has already been caught.

**Finally**

Finally is a block always associated with try – catch to maintain cleanup code

Try

{ Risky code }

Catch(Exception e)

{ Handling code }

Finally

{ Clean up code }

The specialty of finally block is it will be executed always irrespective of whether exception is raised or not raised or whether handled or not handled. Finally block is responsible for to perform clean up activities related to try block. i.e. whatever resources we opened as the part of try block will be closed inside finally block.

* In try – catch- finally order is important
* Whenever we are writing try compulsory we should write either catch or finally otherwise we will get CTE. i.e. try without catch / finally is invalid
* Whenever we are writing catch block compulsory try block must be required i.e. catch without try is invalid
* Whenever we are writing finally block compulsory we should write try block. i.e. finally without try is invalid
* Inside try, catch and finally blocks we can declare try-catch-finally block. i.e. nesting on try-catch-finally is allowed.
* For try-catch-finally block curly braces are mandatory

**Throw keyword**

Sometimes we can create exception objects explicitly, we can handover to the JVM manually for this we have to use Throw keyword.

Throw new AirthmeticException(“/ by 0”)

Here new AE(“/ by 0”) : creation of AE object explicitly

Here throw – handover created exception to the JVM manually.

Hence the main objective of throw keyword is to handover our created exception objects to JVM manually.

Note: best use of throw keyword is for user defined exception or customized exceptions.

Case 1

Throw e – if e refers null then we will get NullPointerException

Ex: class Test

{ static AE e = new AE()

Public static void main(String [] args)

{ throw e; }

}

class Test

{ static AE e;

Public static void main(String [] args)

{ throw e; }

}

Case 2

After throw statement, we are not allowed to write any statement directly otherwise we will get compile time error saying unreachable statement.

Class test

{

P s v main(String args[])

{ sop(10/0);

Sop(“hello”);

}

}

//Run time exception

Class test

{

P s v main(String args[])

{ throw AE e;

SOP(“hello”);

}

}

//Compile time error is non reachable

Case 3

We can use throw keyword only for throwable types. If we are trying to use for normal java objects we will get compile time error saying incompatible types.

Class test

{

P S V main(String args[])

{ throw new test(); }

}

//Compile time error; not throwable

Class Test extends RuntimeException

P S V main(String args[])

{ throw new test(); }

}

//run time exception test

**Throws keyword**

in our program, if there is a possibility of raising checked exception then compulsory we should handle that checked exception otherwise we will get compile time error as : Unreported exception must to be caught or to be thrown.

We can handle this compile time error by using following 2 ways – 1. By using try-catch

2.Using throws keyword : we can use throws keyword to delegate responsibility of exception handling to the caller(it may be another method or JVM).then caller method is responsible to handle that exception

Conclusion: 1.Throws keyword required only for checked exception and usage of throws keyword for unchecked exceptions there is no use or impact.

2.Throws keyword required only to convince compiler and usage of throws keyword doesn’t prevent abnormal termination of program

3.we can use throws keyword to delegate responsibility of exception handling to caller.

Case 1 :we can use Throws keyword for methods and constructors but not for classes.

Case 2 :we can use Throws keyword only for throwable types, if we are using for normal java classes then we will get compile time error.

Case 3: Within the try block, if there is no chance of raising an exception then we can’t write catch block for that exception otherwise we will get compile time error saying Exception XXX is never thrown in body of corresponding try statement.

Try – to maintain risky code

Catch – to maintain exception handling code

Finally – to maintain cleanup code

Throw – To handover our created exception object to the JVM manually.

Throws – To delegate responsibility of exception handling to caller.

Various possible compile time errors in exception handling:

1.Unreporetd exception XXX:must be caught or declared to be thrown

2.Exception XXX has already been caught

3.Exception XXX is never thrown in body of corresponding try statement

4.unreachable statement

5.incomatible types found: Test

Required: java.lang.Throwable

6.try without catch or finally

7.catch without try

8.finally without try

**Customized or User defined exceptions -**

Sometimes to meet, programmer’s requirements we can define our own exceptions such type of exceptions are called customized or user defined exceptions.

Ex: TooYongException, TooOldExceptions, InsufficientFundsException etc.

Note: 1.throw keyword is best suitable for user defined exceptions but not for predefined exceptions.

2.when defining user defined exceptions, highly recommend to use / extends un-checked exception for our exception class ex: **class** ToYoungException **extends** RuntimeException

3.super(s) – to make description available to default exception handler

Top 10 Exceptions –

Based on person who is raising an exception, all exceptions are divided into 2 types. 1.JVM exceptions 2.Programmatic exceptions.

1. JVM exceptions: The exceptions which are raised automatically by JVM, whenever a particular event occurs or called JVM exceptions.

Ex: AirithmaticException , NULLPointerException etc.

1. Programmatic exceptions: the exception which are raised explicitly either by programmer or by API developer to indicate that something goes wrong are called Programmatic exceptions.

Ex: ToOldException, IlligalArgumentException etc.

ArrayIndexOutOfBoundsException: child class of RuntimeException, hence it’s unchecked. Raised automatically by JVM whenever we are trying to access array element with out of range index.

NullPointerException: child class of RuntimeException, hence it’s unchecked. Raised automatically by JVM whenever we are trying to perform any operation on NULL.

ClassCastException: child class of RuntimeException, hence it’s unchecked. Raised automatically by JVM whenever we are trying to type cast, parent object to child type

StackOverflowError: child class of Error, hence it’s unchecked. Raised automatically by JVM whenever we are trying to perform recursive method call.

NoClassDefFoundError: child class of Error, hence it’s unchecked. Raised automatically by JVM whenever JVM unable to find required .class file

ExceptionInInitializerError: child class of Error, hence it’s unchecked. Raised automatically by JVM if any exception occurs while executing static variables assignments and static blocks.

IllegalArgumentException: child class of RuntimeException, hence it’s unchecked. Raised explicitly either by programmer or by API developer to indicate that a method has been invoked with illegal argument.

Ex:Thread t = new Thread();

t.setPriority(7) - valid

t.setPriority(15) – IllegalArgumentException

NumberFormatException: direct child class of IllegalArgumentException, which is child of RuntimeException. Hence it’s un-checked. Raised explicitly either by programmer or API developer to indicate that we are trying to convert string to number and the string is not properly formatted.

Ex: int I = Integer.ParseInt(“10”); //Valid

Int I = Integer.ParseInt(“Ten”); // NumberFormatException

IllegalStateException: child class of RuntimeException, hence it’s unchecked. Raised explicitly either by programmer or API developer to indicate that a method has been invoked at wrong time.

Ex: Thread t = new Thread();

t.start();

.

.

.

t.start();//IllegalStateException:

AssertionError: child class of Error, hence it’s unchecked. Raised explicitly by the programmer or API developer to indicate that assert statement fails.

Ex: assert(x>10) : if x is not greater than 10 then we will get run time exception : AssertionError.

**Try with Resources**

Until 1.6 version, it is highly recommended to write finally block to close resources which are opened as part of try block.

Here compulsory programmer is required to close resources inside finally block, it increases complexity of programming. Also we have to write finally block compulsory and hence it increases length of the code and reduces readability. To overcome these problems – in 1.7 version we have try with resources.

Advantage: Whatever resources we opened as part of try block will be closed automatically once control reaches end of try block either normally or abnormally hence we are not required to close explicitly. We are not required to write finally block so that length of the code will be reduced and readability will be improved.

Conclusion: 1. We can declare multiple resources but these resources should be separated with (;). Ex : try(R1;R2;R3)

Ex: try(FileWriter fr = new FileWriter(“in.txt”); FR fr = new FR(“ou.txt”))

2.All resources should be AutoClosable resources. i.e. AutoClosable – if and only if corresponding class implements java.lang.autoclosable interface . All IO related resources, database related resources and Network related resources are already implemented autoclosable interface.

3. All resource reference variables are implicitly final and hence within the try block we can’t perform reassignment.

**Multi catch block**

Until 1.6 version even though multiple different exception having same handling code for every exception type we have to write a separate catch block. It increases length of the code and reduces readability. To overcome this problem in 1.7 version we have multi catch block. – according to this we can write a single catch block that can handle multiple different type of exceptions.

Try

{ ..}

Catch(AE | IOException)

{ }

Catch(NPE | InterruptedException)

Conclusion: In multi catch block, there shouldn’t be any relation between exception types(either child to parent or parent to child or same type). Ex:

Try{ }

Catch(AE | Exception e) { e.PrintStackTrace();} //CE

**Multi-Threading**

**Introduction:**

Multitasking – Executing several task simultaneously is the concept of multitasking. There are 2 types of multi tasking 1. Process Based 2. Thread based

1.Process Based – Executing several task simultaneously, where each task is a separate independent program(process) is called process based MT.

Ex:while typing java program in editor, we can listen audio songs from same system. At the same time we can download a file from internet all these tasks will be executed simultaneously and independent of each other. Hence it is process based multi tasking. Process based multitasking is best suitable at OS level.

2.Thread Based- Executing several task simultaneously, where each task is a separate independent part of the same program is called Thread based MT and each independent part is called a Thread. Thread based MT is best suitable at programmatic level.

Whether it is process based or thread based the main objective of multi tasking is to reduce response time of the system and to improve performance.

The main important application areas of multi threading’s are 1. To develop multimedia graphics 2.To develop animations 3.To develop videogames 4.To develop web servers and application servers etc.

When compared with old languages developing multi threaded applications in java is very easy because java provides inbuilt support for multi threading with rich API.(Thread, Runnable , ThreadGroup…)

**Defining a Thread** : We can define a thread by using 2 ways

1. By extending Thread class
2. By implementing Runnable interface

By extending Thread class:

**class** VBKThread1 **extends** Thread // {Defining Thread

{

**public** **void** run()

{

**for**(**int** i = 0; i<10;i++) // Job of Thread

System.***out***.println("Child Thread" +i);

}

} // }

**public** **class** ThreadDemo1 {

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

// **TODO** Auto-generated method stub

VBKThread1 t1 = **new** VBKThread1(); //Thread initialization

t1.start(); // start of Thread

**for**(**int** i = 0; i<10;i++) //Main thread execution

System.***out***.println("Main Thread" +i);

}

}

Case 1: Thread Scheduler : It is the part of JVM. It is responsible to schedule threads i.e. If multiple threads are waiting the chance of execution then in which order threads will be executed is decided by Thread Scheduler. We can’t expect exact algorithm followed by scheduler it is varied from JVM to JVM. hence we can’t expect thread execution order and exact output. So whenever situation comes to multithreading there is no guaranty for exact output but we can provide several possible outputs.

Case 2: Difference between t.start() and t.run() : in the case of t.start() a new thread will be created which is responsible for the execution of run() but in the case of t.run() a new thread won’t be created and run method will be executed just like normal method call by main thread.

Case 3: Overloading of run method is always possible but thread class start() method, can invoke no argument run() method. The other overloaded method we have to call explicitly like a normal method call.

Case 4:if we are not overriding run method, then thread class run method will be executed which has empty implementation. Hence we won’t get any output. So it is highly recommended to override run method otherwise don’t go for multi threading concept.

Case 5: if we override start method, then our start method will be executed just like a normal method call and new thread won’t be created. So it is not recommended to override start method.

Case 6:but we can use super.start() in override start method then new child thread will be created.

Case 7: Thread lifecycle: 1. Mythread t = new Mythread() – new / born state

1. After t.start() – Ready / Runnable
2. After Thread scheduler processor – Running
3. Completion of run method – Dead

By implementing Runnable interface

We can define a thread by implementing Runnable interface. Runnable interface present in java.lang package and it contain only one method – Run () method.

**package** com.durgasoft.Thread;

**class** MyRunnable **implements** Runnable

{

**public** **void** run()

{

**for**(**int** i = 0; i<10;i++) // Job of Thread

System.***out***.println("Child Thread" +i);

}

}

**public** **class** ThreadDemo2 {

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

// **TODO** Auto-generated method stub

MyRunnable m1 = **new** MyRunnable(); //Thread initialization

Thread t1 = **new** Thread(m1); // Target Runnable

t1.start(); // start of Thread

**for**(**int** i = 0; i<10;i++) //Main thread execution

System.***out***.println("Main Thread" +i);

}

}

Consider below case study.

MyRunnable m1 = **new** MyRunnable();

Thread t1 = **new** Thread();

Thread t2 = **new** Thread(m1);

Case 1: t1.start() – a new thread will be created and which is responsible for the execution of Thread class run method, which has empty implementation.

Case 2: t1.run() – no new thread will be created and Thread class run method will be executed just like a normal method call

Case 3: t2.start() – a new thread will be created which is responsible for the execution of MyRunnable class run method

Case 4: t2.run() – a new thread won’t be created and MyRunnable run method will be executed just like a normal method call

Case 5: m1.start() – we will get compile time error saying MyRunnable calls doesn’t have start capability.

Case 6: m1.run() – No new thread will be created and MyRunnable run method will be executed like normal method call.

Which approach is best to define a thread?

* Among the 2 ways of define a thread implements runnable approach is recommended. In the first approach our class always extends thread call, there is no chance of extending any other class. Hence we are missing inheritance benefit. But in the 2nd approach while implementing runnable interface we can extend any other class so Inheritance benefit is available so 2nd approach is best.

Thread class constructors –

Thread t = new Thread();

Thread t = new Thread(Runnable r);

Thread t = new Thread(String name);

Thread t = new Thread(Runnable r, String name);

Thread t = new Thread(ThreadGroup g, String name);

Thread t = new Thread(ThreadGroup g, Runnable r);

Thread t = new Thread(ThreadGroup g, Runnable r, String name);

Thread t = new Thread(ThreadGroup g, Runnable r, String name, long statcksize);

Getting and Setting a name of Thread:

**package** com.durgasoft.Thread;

**class** MyThread1 **extends** Thread

{

}

**public** **class** TestThread1 {

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

// **TODO** Auto-generated method stub

System.***out***.println(Thread.*currentThread*().getName());

MyThread1 t = **new** MyThread1();

System.***out***.println(t.getName());

Thread.*currentThread*().setName("Vaibhav v1");

System.***out***.println(Thread.*currentThread*().getName());

System.***out***.println(10/0);

}

}

Thread Priorities: Every thread in java has some priority. It may be default priority generated by JVM or customized priority provided by programmer. The valid range of priorities is 1 to 10, where 1 is Minimum and 10 is maximum priority. Thread class defines the following constants to represent some standard priorities.

Thread.MIN\_PRIORITY – 1

Thread.NORM\_PRIORITY – 5

Thread.MAX\_PRIORITY – 10

Thread scheduler will use priorities while allocating processor, the thread which is having highest priority will get chance first. If 2 threads having same priority then we can’t expect exact execution order it depends on thread scheduler.

Thread class defines methods to get and set the priorities of a thread.

Public final int getPriority();

Public final void setPriority(int p);

Allowed values range is 1 to 10, otherwise we will get run time exception : IlligalArgumentException.

Default Priority: The Default priority only for the main thread is 5 but for all remaining thread default priority will be inherited from parent to child. i.e. Whatever priority parent thread has the same priority will be there for the child thread.

**package** com.durgasoft.Thread;

**import** com.durgasoft.Thread.MyThread1;

**public** **class** TestThread2 {

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

System.***out***.println(Thread.*currentThread*().getPriority());

Thread.*currentThread*().setPriority(7);

MyThread1 t1 = **new** MyThread1();

System.***out***.println(t1.getPriority());

}

}

Demo Program:

**package** com.durgasoft.Thread;

**class** VBKThread1 **extends** Thread // {Defining Thread

{

**public** **void** run()

{

**for**(**int** i = 0; i<10;i++) // Job of Thread

System.***out***.println("Child Thread" +i);

}

} // }

**public** **class** ThreadDemo1 {

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

// **TODO** Auto-generated method stub

VBKThread1 t1 = **new** VBKThread1(); //Thread initialization

t1.setPriority(10);

t1.start(); // start of Thread

**for**(**int** i = 0; i<10;i++) //Main thread execution

System.***out***.println("Main Thread" +i);

}

}

**Note: Some platforms won’t provide proper support for Thread priorities.**

We can prevent a thread execution by using the following methods – yield(), join(), sleep().

Yield() method: Yield method causes to pause current executing thread to give the chance for waiting threads of same priority. If there is no waiting thread or all waiting have low priority then same thread can continue it’s execution. If multiple threads are waiting with same priority then which waiting thread will get the chance, we can’t expect. It depends on thread scheduler.

The thread which is yielded, when it will get chance once again it depends on thread scheduler and we can expect exactly.

Public static native void yield()

**Note: Some platforms won’t provide proper support for yield method.**

Join() method: if a thread wants to wait until completing some other thread then we should go for join method(). For example, if a thread t1 wants to wait until completing t2 then t1 has to call t2.join(). If t1 executes t2.join() then immediately t1 will be entered into waiting state until t2 completes. Once t2 completes then t1 can continue it’s execution.

Public final void join() throws InterruptedException

Public final void join(long millisecond) throws InterruptedException

Public final void join(long millisecond, int nanoseconds) throws InterruptedException

Note: Every join method throws interrupted exception which is checked exception hence compulsory we should handle this exception either by using try-catch or by throws keyword otherwise we will get compile time error.

Example Case 1: Main thread waiting for child thread.

**package** com.durgasoft.Thread;

**class** MyThreadJoin **extends** Thread

{

**public** **void** run()

{

**try**{

**for**(**int** i=0; i<10;i++)

{

System.***out***.println("Sita thread");

Thread.*sleep*(2000);

}

}**catch**(InterruptedException e){ }

}

}

**public** **class** ThreadJoinDemo {

**public** **static** **void** main(String[] args) **throws** InterruptedException

{

MyThreadJoin t1 = **new** MyThreadJoin();

t1.start();

t1.join(10000);

**for**(**int** i=0; i<10;i++)

{

System.***out***.println("Rama thread");

}

}

}

Example Case 2:Child thread waiting for main thread

**package** com.durgasoft.Thread;

**class** MyThreadJoin2 **extends** Thread

{

**static** Thread *mt*;

**public** **void** run()

{

**try**

{

*mt*.join();

}

**catch**(InterruptedException e){}

**for**(**int** i=0; i<10;i++)

System.***out***.println("Sita thread");

}

}

**public** **class** ThreadJoinDemo2 {

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

// **TODO** Auto-generated method stub

MyThreadJoin2.*mt* = Thread.*currentThread*();

MyThreadJoin2 t1 = **new** MyThreadJoin2();

t1.start();

**for**(**int** i=0; i<10;i++)

{

System.***out***.println("Rama thread");

Thread.*sleep*(2000);

}

}

}

Example Case 3: If main thread call join method on child thread object and child thread call join method on main thread object then both threads will wait forever and the program will be stuck. This is called as Deadlock.

**package** com.durgasoft.Thread;

**class** MyThreadJoin2 **extends** Thread

{

**static** Thread *mt*;

**public** **void** run()

{

**try**

{

*mt*.join();

}

**catch**(InterruptedException e){}

**for**(**int** i=0; i<10;i++)

System.***out***.println("Sita thread");

}

}

**public** **class** ThreadJoinDemo2 {

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

// **TODO** Auto-generated method stub

MyThreadJoin2.*mt* = Thread.*currentThread*();

MyThreadJoin2 t1 = **new** MyThreadJoin2();

t1.start();

t1.join();

**for**(**int** i=0; i<10;i++)

{

System.***out***.println("Rama thread");

Thread.*sleep*(2000);

}

}

}

Example Case 4: If a thread calls join method on same Thread itself then the program will be stuck. This is also deadlock.

**public** **class** ThreadJoinDemo2 {

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

// **TODO** Auto-generated method stub

Thread.*currentThread*.join();

}

}

Sleep() method: if a thread don’t want to perform any operation for a particular amount of time then we should go for sleep method.

Public static native void sleep(long millisecond) throws InterruptedException

Public static void sleep(long millisecond, int nanosecond) InterruptedException

Example

**package** com.durgasoft.Thread;

**public** **class** SlideRotate {

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

// **TODO** Auto-generated method stub

**for**(**int** i = 0 ; i<5; i++){

System.***out***.println("Slide :" +i);

Thread.*sleep*(2000);

}

}

}

[**Java Thread**](https://www.journaldev.com/1079/multithreading-in-java)**join** method can be used to pause the current thread execution until unless the specified thread is dead. There are three overloaded join functions.

How a thread can interrupt another Thread?

A thread can interrupt a sleeping thread or waiting thread by using Interrupt() method of thread class

Public void interrupt();

Example

**package** com.durgasoft.Thread;

**class** SleepInterrupt **extends** Thread

{

**public** **void** run()

{

**try**

{

**for**(**int** i=0;i<5;i++)

{

System.***out***.println("I am lazy Thread");

Thread.*sleep*(2000);

}

}

**catch**(InterruptedException e)

{

System.***out***.println("I got Interrupt");

}

}

}

**public** **class** TheadSleepDemo {

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

SleepInterrupt t1 = **new** SleepInterrupt();

t1.start();

t1.interrupt();

System.***out***.println("Main Thread");

}

}

Note: Whenever we are calling interrupt method if target thread not in sleeping state or waiting state then there is no impact of interrupt call immediately. Interrupt call will be waited until target thread entered into sleeping or waiting state. If the target thread entered into sleeping or waiting state then immediately interrupt call will interrupt target thread.

If the target thread never entered into sleeping or waiting state in it’s lifetime then there is no impact of interrupt call.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Yield()** | **Join()** | **Sleep()** |
| Purpose | If a thread wants to pause it’s execution to give the chance for remaining threads of same priority then we should go for yield () | If a thread wants to wait until completing some other thread then we should go for join method | If a thread don’t want to perform any operation for a particular amount of time then we should go for sleep() |
| Is it overloaded | No | Yes | Yes |
| Is it Final | No | Yes | No |
| Is it throws Interrupted Exception | No | Yes | Yes |
| Is it Native | Yes | No | One method is Native, another is not |

**Synchronization**

Synchronized is the modifier applicable only for methods and blocks but not for classes and variables. If multiple threads are trying to operate simultaneously on the same java object then there may be chance of data inconsistency problem. To overcome this problem we should go for synchronized keyword. If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved.

The main advantage of synchronized keyword is we can resolve data inconstancy problems but main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problems. Hence if there is no specific requirements then it is not recommended to use synchronized keyword.

Internally synchronization concept is implemented by using a lock. Every object in java has unique lock whenever we are using synchronized keyword then only lock concept will come into the picture. If a thread wants to execute synchronized method on the given object first it has to get lock of that object. Once thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases lock.

Acquiring and releasing lock internally takes care by JVM and programmer not responsible for this activity.

While a thread executing synchronized method on given object the remaining threads are not allowed to execute any synchronized method simultaneously on same object but remaining threads are allowed to execute non-synchronized methods simultaneously.

Class x

{

Sync m1;

Sync m2;

M3

}

On X object T1 came to execute M1, if T2 came to execute m1: Waiting state

If T3 came to execute m2: Waiting state

If T4 came to execute m3: no problem

Lock concept is implemented based on object but not on methods.

Object have 2 areas – Synchronized area and non-synchronized area.

Non-Synchronized – This area can be accessed by any number of threads simultaneously

Synchronized- This area can be accessed by only one thread at a time

Class X

{

Synchronized {

Whenever we are performing update operation.(Add,remove,delete) where state of object is changing.

}

Non-Synchronized {

Read operation, wherever state of object is not changing

}

}

Example

**package** com.durgasoft.Thread;

**class** Display

{

**public** **synchronized** **void** wish(String name)

{

**try**

{

**for**(**int** i = 0; i< 6; i++)

{

System.***out***.print("Good Morning:");

Thread.*sleep*(1000);

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

}

}

**catch**(InterruptedException e){}

}

}

**class** MyThreadDisplay **extends** Thread

{

Display d;

String name;

MyThreadDisplay(Display d, String name)

{

**this**.d = d;

**this**.name = name;

}

**public** **void** run()

{

d.wish(name);

}

}

**public** **class** SynchronizedDemo {

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

Display d1 = **new** Display();

MyThreadDisplay t1 = **new** MyThreadDisplay(d1,"Dhoni");

MyThreadDisplay t2 = **new** MyThreadDisplay(d1,"Yuvraj");

t1.start();

t2.start();

}

}

O/P:

Good Morning

Dhoni

Good Morning

Dhoni

Good Morning

Dhoni

Good Morning

Dhoni

Good Morning

Dhoni

Good Morning

Yuvraj

Good Morning

Yuvraj

Good Morning

Yuvraj

Good Morning

Yuvraj

Good Morning

Yuvraj

Case study : If we change below code in above example

Display d1 = **new** Display();

Display d2 = **new** Display();

MyThreadDisplay t1 = **new** MyThreadDisplay(d1,"Dhoni");

MyThreadDisplay t2 = **new** MyThreadDisplay(d2,"Yuvraj");

t1.start();

t2.start();

o/p:

Good Morning

Good Morning

Dhoni

Good Morning

Yuvraj

Good Morning

Dhoni

Good Morning

Yuvraj

Good Morning

Dhoni

Yuvraj

Good Morning

Good Morning

Yuvraj

Dhoni

Good Morning

Good Morning

Dhoni

Yuvraj

Even though wish method is synchronized we will get irregular output because threads are operating on different java objects.

Conclusion: If multiple threads are operation on same java objects then Synchronization is required. If multiple threads are operating on multiple java objects then synchronization is not required.

Class level lock: every class in java has a unique lock, which is nothing but class level lock. If a thread want’s to execute a static synchronized method then thread required class level lock. Once thread got class level lock then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread release lock.

While a thread executing static synchronized method the remaining threads are not allowed to execute any static synchronized methods of that class simultaneously but remaining threads are allowed to execute the following methods simultaneously.

1. Normal static methods
2. Synchronized instance methods
3. Normal instance methods

Ex: class X {

Static sync m1()

Static sync m2()

Static m3()

Synchronized m4()

M5()

}

T1 on object X executing m1 came first : So gets execution.

T2 executing m2 : waiting

T3 executing m3 : execution go ahead

T4 executing m4 : execution go ahead

T5 executing m5 ; execution go ahead

Synchronized block

If very few lines of the code required synchronization then it is not recommended to declare entire method as synchronized. We have to enclose those few lines of the code by using synchronized block. The main advantage of synchronized block over synchronized method is it reduces waiting time of thread and improves performance of the application.

We can declare synchronized block as follows

1. To get lock of current object

Synchronized(this)

{ }

1. To get lock of particular object

Synchronized(d)

{ }

1. To get class level lock

Synchronized( Display.class)

Example – approach 1

**package** com.durgasoft.Thread;

**class** Display

{

**public** **void** wish(String name)

{

;;;;;;;//1 lakhs lines of code

**try**

{

**synchronized**(**this**)

{

**for**(**int** i = 0; i< 6; i++)

{

System.***out***.print("Good Morning:");

Thread.*sleep*(1000);

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

}

}

}

**catch**(InterruptedException e){}

;;;;;;;//1 lakhs lines of code

}

}

**class** MyThreadDisplay **extends** Thread

{

Display d;

String name;

MyThreadDisplay(Display d, String name)

{

**this**.d = d;

**this**.name = name;

}

**public** **void** run()

{

d.wish(name);

}

}

**public** **class** SynchronizedDemo {

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

Display d1 = **new** Display();

MyThreadDisplay t1 = **new** MyThreadDisplay(d1,"Dhoni");

MyThreadDisplay t2 = **new** MyThreadDisplay(d1,"Yuvraj");

t1.start();

t2.start();

}

}

O/P:

Good MorningDhoni

Good MorningDhoni

Good MorningDhoni

Good MorningDhoni

Good MorningDhoni

Good MorningYuvraj

Good MorningYuvraj

Good MorningYuvraj

Good MorningYuvraj

Good MorningYuvraj

Example – approach 3

**package** com.durgasoft.Thread;

**class** Display

{

**public** **void** wish(String name)

{

**try**

{

**synchronized**(Display.class)

{

**for**(**int** i = 0; i< 6; i++)

{

System.***out***.print("Good Morning:");

Thread.*sleep*(1000);

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

}

}

}

**catch**(InterruptedException e){}

}

}

**class** MyThreadDisplay **extends** Thread

{

Display d;

String name;

MyThreadDisplay(Display d, String name)

{

**this**.d = d;

**this**.name = name;

}

**public** **void** run()

{

d.wish(name);

}

}

**public** **class** SynchronizedDemo {

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

Display d1 = **new** Display();

Display d2 = **new** Display();

MyThreadDisplay t1 = **new** MyThreadDisplay(d1,"Dhoni");

MyThreadDisplay t2 = **new** MyThreadDisplay(d2,"Yuvraj");

t1.start();

t2.start();

}

}

O/p:

Good MorningDhoni

Good MorningDhoni

Good MorningDhoni

Good MorningDhoni

Good MorningDhoni

Good MorningYuvraj

Good MorningYuvraj

Good MorningYuvraj

Good MorningYuvraj

Good MorningYuvraj

Lock concept applicable for object types and class types but not for primitives. Hence we can’t pass primitives type as argument to synchronized block otherwise we will get compile time error.

Int x = 10

Synchronized(x)

{ }

FAQs

What is RACE condition?

If multiple threads operating on same java object then there is chance of data inconsistency. This condition is RACE condition. We can resolve this by using Synchronized keyword.

Is a thread can acquire multiple lock simultaneously?

Yes

While a thread executing synchronized method on the given object, Is remaining thread are allowed to execute any other synchronized method simultaneously on the same object?

No.

What is Synchronized statement?

The statements present in synchronized method and synchronized block are called Synchronized statement

**Inter Thread communication:**

2 threads can communicate with each other by using wait(),notify() and notifyall() methods. The thread which is expecting update is responsible to call wait() method then immediately thread will enter into waiting state, the thread which is responsible to perform updates, after performing updates it is responsible to call notify method then waiting thread will get that notification and continue it’s execution with those updated items.

Wait,notify,notifyall methods present in Object class but not in Thread class because thread can call these methods on any java objects.

To call, wait – notify or notifyall methods on any object, thread should be owner of that object. i.e. thread should has lock of that object. i.e. thread should be inside synchronized area. Hence we can call wait, notify, notifyall methods only from synchronized area otherwise we will get runtime exception saying IlligalMonitorStateException.

If a thread calls wait method on any object it immediately release lock of that particular object and entered into waiting state.

If thread call notify method on any object it release lock of that object but may not immediately, except wait, notify, and notifyall there is no other method where thread releases the lock.

Which of the following is valid?

1. If thread calls wait method, immediately it will enter into waiting state without releasing any lock – invalid
2. If thread call wait method, it releases the lock of that object but may not immediately. – invalid
3. If thread call wait method on any object it releases all locks acquired by that thread and immediately entered into waiting state – invalid
4. If thread call wait method on any object it immediately releases the lock of that particular object and entered into waiting state – valid
5. If thread call notify method on any object it immediately release lock of that particular object – invalid
6. If thread calls notify method on any object, it release lock of that object but may not immediately. – valid

Public final void wait() throws InterruptedException

Public final native void wait(long millisecond) throws InterruptedException

Public final void wait(long millisecond, int nanosecond) throws InterruptedException

Public final native void notify()

Public final native void notifyall()

Note: every wait method throws InterruptedException, which is checked exception hence whenever we are using wait method compulsory we should handle this InterruptedException either by try-catch or throws keyword.

Example : Inter thread communication:

**package** com.durgasoft.Thread;

**class** ThreadB **extends** Thread

{

**int** total = 0;

**public** **void** run()

{

**synchronized**(**this**)

{

System.***out***.println("Child thread starts calculation");

**for**(**int** i=1;i<=100;i++)

total = total + i;

System.***out***.println("Child thread send notification");

**this**.notify();

}

}

}

**public** **class** InterThread {

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

ThreadB b1 = **new** ThreadB();

b1.start();

**synchronized**(b1)

{

System.***out***.println("Main thread calling wait method");

b1.wait();

System.***out***.println("Main thread got notification");

System.***out***.println(b1.total);

}

}

}

O/P:

Main Thread calling wait method

Child Thread starts execution

Child Thread send notification

Main Thread got notification

1225

Difference between notify and notifyall method

We can use notify method, to give the notification for only one waiting thread. If multiple threads are waiting then only one thread will be notified and remaining threads have to wait for further notifications. Which thread will be notified we can’t except it depends on JVM.

We can use notifyall method, to give the notification for all waiting threads of a particular object, even though multiple threads notified but execution will be performed one by one because threads require lock and only one lock is available.

Note: on which object we are calling wait method, thread required the lock of that particular object for example if we are calling wait method on s1 then we have to get lock of s1 object but not s2 object.

**Deadlock**

If 2 threads are waiting for each other forever such type of infinite waiting is called Deadlock. Synchronized keyword is the only reason for deadlock situation hence while using synchronized keyword we have to take special care. There are no resolution technique but several prevention techniques are available

Deadlock Vs Starvation

Long waiting of a thread where waiting never ends is Deadlock. Whereas long waiting of a thread where waiting ends at certain point is Starvation.

For Example – low priority threads has to wait until completing all high priority threads, so here waiting is long but it will end at certain point.

**Daemon Threads**

The threads which are executing in the background are called daemon threads. Ex: Garbage collector, Signal Dispatcher etc.

The main objective of daemon thread is to provide support for non daemon threads ( Main thread). For example if main thread runs with low memory then JVM run garbage collector to destroy useless objects so that number of bytes of free memory will be improved with this free memory main thread can continue it’s execution. Usually daemon threads runs with low priority but based on situation they can run with high priority as well.

We can check, daemon nature of a thread by using isDaemon method of Thread class.

Public Boolean isDaemon()

We can change, Daemon nature of a thread by using setDaemon method but changing daemon nature is possible before starting of thread only. After starting of thread, if we are trying to change, we will get run time exception : IlligalThreadStateException.

Public void setDaemon(Boolean b)

Default nature of thread: by default main thread is always non daemon and for all remaining threads daemon nature will be inherited from parent to child. i.e. if the parent thread is daemon then automatically child thread is also daemon.

Example

**package** com.durgasoft.Thread;

**class** DaemonClass **extends** Thread

{

**public** **void** run()

{

**for** (**int** i = 0; i < 5; i++)

{

System.***out***.println("Child Thread");

**try**

{

Thread.*sleep*(2000);

}

**catch**(InterruptedException e){ }

}

}

}

**public** **class** DaemonThreadDemo {

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

DaemonClass t1 = **new** DaemonClass();

t1.setDaemon(**true**);

t1.start();

System.***out***.println("Main Thread");

}

}

What is green thread?

Java multithreading is implemented by using 2 models

1. Green thread model
2. Native OS model

Green thread model: the threads which are managed completely by JVM without taking underlying OS support is called Green thread. Very few OS like SunSolarise provide support for Green Thread model. Now it’s deprecated and not recommended to use

Native OS model: The thread which is managed by the JVM with the help of underlying OS is called native OS model.

How to stop a thread?

We can stop a thread execution by using stop() method of thread class, if we call stop method then immediately that thread will enter into dead thread. Anyway stop method is deprecated and not recommended to use.

Suspend() and Resume(): deprecated.

**Advanced Multi-Threading**

**TheadGroup:**

Based on functionality, we can group threads into a single unit which is nothing but thread group. i.e. ThreadGroup contains a group of threads. In addition to threads, ThreadGroup can also contains sub thread groups. The main advantage of maintaining threads, in form of thread group is we can perform common operations very easily.

Every thread in java belongs to some group, main thread belongs to Main group. Every thread group in java is the child group of System group either directly or indirectly hence System group root for all thread groups in java

System group contains several system level threads, like Finalizer, ReferenceHandler, signalDispatcher etc.

ThreadGroup is a java class present in java/lang package and it is direct child class of object.

Constructors:

1. ThreadGroup g = new ThreadGroup(String name) : creates a new threadgroup with specified group name. The parent of this is new group is ThreadGroup of currently executing thread.

ThreadGroup g1 = **new** ThreadGroup("First Group");

1. ThreadGroup g = new ThreadGroup(ThreadGroup pg, String groupname) : creates a new threadgroup with specified group name, the parent of this group is specified parent group.

ThreadGroup g1 = **new** ThreadGroup("First Group");

System.***out***.println(g1.getParent().getName());

ThreadGroup g2 = **new** ThreadGroup(g1,"Second Group");

System.***out***.println(g2.getParent().getName());

Methods of ThreadGroup class:

String getName()

Int getMaxPriority()

Void setMaxPriority()

ThreadGroup getParent()

Void list() //it prints information about ThreadGroup to console

Int activeCount() //returns number of active threads presents in ThreadGroup

Int activeGroupCount() // returns number of active groups present in current ThreadGroup

Int enumerate(Thread[]t) //To copy all active threads of ThreadGroup into provided Thread array.in this case sub threadsgroups threads also considered.

Int enumerate(ThreadGrops[] t) // To copy all active subthread groups into ThreadGroup array.

Bolean IsDaemon() : to check if ThreadGroup is daemon

Void setDaemon(Boolean b)

Void interrupt() // To interrupt all waiting or sleeping threads present in threadgroup

Void destroy() // to destroy threadgroup and it’s sub thread groups.

**Java.util.concurrent package:**

The problems with traditional synchronized keyword:

1. We are not having any flexibility to try for a lock without a waiting.
2. There is no way to specify maximum waiting time for a thread to get lock so that thread will wait until getting the lock which may create performance problems, which may cause deadlock.
3. If a thread release lock, then which waiting thread will get that lock, we are not having any control on this.
4. There is no API to list out all waiting threads for a lock.
5. The synchronized keyword compulsory we have to use either at method level or within the method and it’s not possible to use across multiple methods

To overcome these problems, Java introduced java.util.concurrent.locks package in 1.5 version. It also provides several enhancements to programmer to provide more control on concurrency.

**Lock interface:**

Lock object is similar to implicit lock acquired by a thread to execute synchronized method or block. Lock implementation provide more extensive operations than traditional implicit locks.

Important methods of lock interface

Void lock() : we can use this method to acquire a lock. If lock is already available then immediately current thread will get that lock. If lock is not already available then it will wait until getting the lock. It is exactly same behavior of traditional synchronized keyword.

Boolean tryLock() : to acquire the lock without waiting. If lock is available then thread acquire that lock and return true if lock is not available then this method returns false and can continue it’s execution without waiting. In this case thread never be entered into waiting state.

Ex:

if(l.tryLocks)

{

Perform safe operation

}

Else

{ perform alternative operations }

Boolean tryLock(long time, TimeUnit unit) : if lock is available then thread will get the lock and continue its execution, if the lock is not available then thread will wait until specified amount of time, still if the lock is not available then thread can continue its execution.

TimeUnit : Is an enum present in java.util.concurrent package

Enum TimeUnit

{ NANAOSECONDS

MICROSECONDS

MILISECONDS

SECONDS

MINUTES

HOURS

DAYS

}

Ex:

If(l.tryLock(1000,TimeUnit milliseconds)

Void lockInterruptibly() : acquires lock if it’s available in return immediately. If lock is not available then it will wait. While waiting if thread is interrupted then thread won’t get lock.

Void unlock() : to release lock. To call this method compulsory current thread should be owner of the lock otherwise we will get runtime exception: IiilgalMonitorStateException.

ReentrantLock(C) : This is implementation class of lock interface and it is direct child class of Object. Reentrant means a thread can acquire same lock multiple time without any issue. Internally reentrantLock increments threads personal count wherever we call lock method and decrements count value whenever thread call unlock method and lock will be released whenever count reaches zero(0).

Constructrors:

ReentrantLock l = new ReentrantLock() : create instance of reentrantLock

ReentrantLock l = new ReentrantLock(Boolean fairness) : Creates reentrantlock with given fairness policy.If fairness is true then longest waiting thread can acquire lock if it is available i.e. this follows FCFS(First Come First Serve) policy. If fairness is false then which waiting thread will get chance we can’t expect. The default value for fairness is FALSE.

Which of the following declarations are equal?

ReentrantLock l = new ReentrantLock();

ReentrantLock l = new ReentrantLock(false);

ReentrantLock l = new ReentrantLock(true);

All above

Ans: 1 and 2.

Important methods of ReentrantLock:

Void lock()

Boolean tryLock()

Boolean tryLock(long l,TimeUnit t)

Void lockInterruptibly()

Void unlock()

Int getHoldCount() : Returns number of holds on this lock by current thread.

Boolean isHeldByCurrentThread(): Returns true if and only if lock is hold by current thread

Int getQueueLength(): returns number of threads waiting for lock.

Collection getQueuedThreads() : returns a collection of threads which are waiting to get the lock.

Boolean hasQueuedThreads(): returns true if any thread waiting to get lock.

Boolean isLocked(): returns true if lock is acquired by some thread.

Boolean isFair(): returns true if the fairness policy is set with True value.

Thread getOwner(): returns the Thread which acquires the lock.

**package** com.Example2;

**import** java.util.concurrent.locks.\*;

**public** **class** ReentrantDemo1 {

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

ReentrantLock l1 = **new** ReentrantLock();

l1.lock();

l1.lock();

System.***out***.println(l1.isHeldByCurrentThread()); //true

System.***out***.println(l1.isLocked()); //true

System.***out***.println(l1.getQueueLength()); //0

l1.unlock();

System.***out***.println(l1.getHoldCount());//1

System.***out***.println(l1.isLocked());//true

l1.unlock();

System.***out***.println(l1.isLocked());//false

System.***out***.println(l1.isFair());//false

}

}

Instead of Synchronized keyword How we can use ReentrantLock?

Display.java

**package** Example;

**import** java.util.concurrent.locks.\*;

**class** Display

{

ReentrantLock l1 = **new** ReentrantLock();

**public** **void** wish(String name)

{

**try**

{

l1.lock();

**for**(**int** i = 0; i< 6; i++)

{

System.***out***.print("Good Morning:");

Thread.*sleep*(1000);

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

}

l1.unlock();

}

**catch**(InterruptedException e){}

}

}

MyThreadDisplay.java

**package** Example;

**class** MyThreadDisplay **extends** Thread

{

Display d;

String name;

MyThreadDisplay(Display d, String name)

{

**this**.d = d;

**this**.name = name;

}

**public** **void** run()

{

d.wish(name);

}

}

ReentrantDemo.java

**package** Example;

**public** **class** ReentrantDemo {

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

Display d1 = **new** Display();

MyThreadDisplay t1 = **new** MyThreadDisplay(d1,"Dhoni");

MyThreadDisplay t2 = **new** MyThreadDisplay(d1,"Yuvraj");

t1.start();

t2.start();

}

}

O/P:

Good Morning:Dhoni

Good Morning:Dhoni

Good Morning:Dhoni

Good Morning:Dhoni

Good Morning:Dhoni

Good Morning:Dhoni

Good Morning:Yuvraj

Good Morning:Yuvraj

Good Morning:Yuvraj

Good Morning:Yuvraj

Good Morning:Yuvraj

Good Morning:Yuvraj

**Demo program for tryLock() method.**

MyThread.java

**package** com.Example2;

**import** java.util.concurrent.locks.\*;

**public** **class** MyThread **extends** Thread{

**static** ReentrantLock *l1* = **new** ReentrantLock();

**public** MyThread(String name) {

**super**(name);

}

@Override

**public** **void** run() {

**if**(*l1*.tryLock())

{

System.***out***.println(Thread.*currentThread*().getName()+"Got the lock and performing operation");

**try**

{

Thread.*sleep*(2000);

}

**catch**(InterruptedException e) {}

*l1*.unlock();

}

**else**

{

System.***out***.println(Thread.*currentThread*().getName()+"didn't got lock hence performing his other operation");

}

}

}

ReentrantDemo2.java

**package** com.Example2;

**public** **class** ReentrantDemo2 {

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

MyThread t1 = **new** MyThread("First Thread");

MyThread t2 = **new** MyThread("Second Thread");

t1.start();

t2.start();

}

}

o/p:

Second ThreadGot the lock and performing operation

First Threaddidn't got lock hence performing his other operation

**Demo program for tryLock() method with Timeunit.**

MyThread2.java

**package** com.Example2;

**import** java.util.concurrent.TimeUnit;

**import** java.util.concurrent.locks.\*;

**public** **class** MyThread2 **extends** Thread{

**static** ReentrantLock *l1* = **new** ReentrantLock();

**public** MyThread2(String name) {

**super**(name);

}

@Override

**public** **void** run() {

**do**

{

**try**

{

**if**(*l1*.tryLock(5000, TimeUnit.***MILLISECONDS***))

{

System.***out***.println(Thread.*currentThread*().getName()+":got the lock and performing operation");

Thread.*sleep*(30000);

*l1*.unlock();

System.***out***.println(Thread.*currentThread*().getName()+":Released lock");

**break**;

}

**else**

{

System.***out***.println(Thread.*currentThread*().getName()+"didn't got lock hence performing his other operation");

}

}

**catch**(InterruptedException e) {}

}

**while**(**true**);

}

}

ReentrantDemo3.java

**package** com.Example2;

**public** **class** ReentrantDemo3 {

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

MyThread2 t1 = **new** MyThread2("First Thread");

MyThread2 t2 = **new** MyThread2("Second Thread");

t1.start();

t2.start();

}

}

**O/P**

First Thread:got the lock and performing operation

Second Threaddidn't got lock hence performing his other operation

Second Threaddidn't got lock hence performing his other operation

Second Threaddidn't got lock hence performing his other operation

Second Threaddidn't got lock hence performing his other operation

Second Threaddidn't got lock hence performing his other operation

First Thread:Released lock

Second Thread:got the lock and performing operation

Second Thread:Released lock

**Thread Pools (Executor Framework)**

Creating a new thread for every job may create performance and memory problems, to overcome this we should go for ThreadPool. ThreadPool is a pool of already created threads ready to do our job.

Java 1.5 version introduced ThreadPools, ThreadPool framework also known as Executor framework. We can create a ThreadPool as follows

ExecutorService service = Executors.newFixedThreadPool(3);

We can submit a Runnable job by using submit method.

Service.submit(job);

We can shutdown executor service by using shutdown method.

Service.shutdown();

Example

PrintJob.java

**package** com.Example2;

**public** **class** PrintJob **implements** Runnable

{

String name;

**public** PrintJob(String name) {

**this**.name=name;

}

@Override

**public** **void** run() {

System.***out***.println(name+"...Job started by"+Thread.*currentThread*().getName());

**try** {

Thread.*sleep*(5000);

}

**catch**(InterruptedException e) {}

System.***out***.println(name+"...Job Completed by"+Thread.*currentThread*().getName());

}

}

ExecutorDemo.java

**package** com.Example2;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** ExecutorDemo {

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

PrintJob[] jobs = { **new** PrintJob("Vivek"),

**new** PrintJob("Nikita"),

**new** PrintJob("Neha"),

**new** PrintJob("Sarthak"),

**new** PrintJob("Vaishnavi"),

**new** PrintJob("Sidhant")

};

ExecutorService service = Executors.*newFixedThreadPool*(3);

**for**(PrintJob job:jobs)

{

service.submit(job);

}

service.shutdown();

}

}

In above example 3 threads are responsible to execute 6 jobs so that a single Thread can be reused for multiple jobs.

Note: While designing web servers and application servers, we can use ThreadPool concept. Default threadpool size is 60 for most of the servers but we can customize that as well.

**Callable and Future**

In the case of Runnable job, thread won’t return anything after completing job, If a Thread is required to return some result after execution then we should go for callable. In callable interface contain only method – call

Public object call() throws Exception

If we submit callable object to executor then after completing the job, Thread returns an object of type ‘Future’. i.e. Future object can be used to retrieve the result from callable job.

MyCallable.java

**package** com.Example2;

**import** java.util.concurrent.Callable;

**public** **class** MyCallable **implements** Callable<Integer>{

**int** num;

**public** MyCallable(**int** num) {

**this**.num = num;

}

@Override

**public** Integer call() **throws** Exception {

System.***out***.println(Thread.*currentThread*().getName()+"is ...Responsible to find sum of"

+ "first"+num+"numbers");

**int** sum=0;

**for**(**int** i=1;i<=num;i++)

{

sum=sum+i;

}

**return** sum;

}

}

CallableFutureDemo.java

**package** com.Example2;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**public** **class** CallableFutureDemo {

**public** **static** **void** main(String[] args) **throws** Exception

{

MyCallable[] jobs = { **new** MyCallable(10),

**new** MyCallable(20),

**new** MyCallable(30),

**new** MyCallable(40),

**new** MyCallable(50),

**new** MyCallable(60)

};

ExecutorService service = Executors.*newFixedThreadPool*(3);

**for**(MyCallable job:jobs)

{

Future<Integer> f = service.submit(job);

System.***out***.println(f.get());

}

service.shutdown();

}

}

**o/p:**

pool-1-thread-1is ...Responsible to find sum offirst10numbers

55

pool-1-thread-2is ...Responsible to find sum offirst20numbers

210

pool-1-thread-3is ...Responsible to find sum offirst30numbers

465

pool-1-thread-1is ...Responsible to find sum offirst40numbers

820

pool-1-thread-2is ...Responsible to find sum offirst50numbers

1275

pool-1-thread-3is ...Responsible to find sum offirst60numbers

1830

**Difference between Runnable and Callable**

|  |  |
| --- | --- |
| **Runnable** | **Callable** |
| If a Thread is not required to return anything after completing the job then, we should go for Runnable interface | If a Thread required to return something after completing the job then we should go for callable |
| Runnable interface contains only one method  Public void Run() | Callable interface contains only one method  Public object call() |
| Runnable job not required to return anything and hence return type of run() method is void | Callable job is required to return something and hence return type of Call() method is Object |
| Within the Run() method, if there is any chance of any checked exception, compulsory we should handle using try-catch because we can’t use Throws keyword for run() method | Inside call() method if there is any chance of raising checked exception we are not required to handle by using try-catch because call() method already throws exception |
| Runnable interface present in java.lang package | Callable interface present in java.util.concurrent package |
| Introduced in 1.0 version | Introduced in 1.5 version |

**Thread Local**

Thread local class provides, Thread local variable. Thread local class maintains values per Thread basis. Each Thread local object maintains separate value like userid, transaction id etc. For each thread that access that object.

Thread can access its local value, can manipulate its value and even can remove its value. In every part of the code which is executed by the thread we can access its local variable.

Example: Consider a servlet which invokes some business methods, we have a requirement to generate a unique transaction id for each and every request and we have to pass this transaction id to the business methods. For this requirement we can use ThreadLocal to maintain a separate transaction id for every request. i.e for every Thread.

Note: 1. Thread Local class introduced in 1.2 version and enhanced in 1.5 version.

2. Thread local can be associated with Thread scope.

3. Total code which is executed by the Thread, has access to the corresponding Thread local.

4. A Thread can access its own local variable and can’t access other thread’s local variables.

5. Once Thread entered into dead state, all its local variable are by default eligible for garbage collection.

Constructor:

ThreadLocal tl = new ThreadLocal(); Create a ThreadLocal variable.

Methods:

Object get() //return the value of threadlocal variable associated with current Thread.

Object initialValue() //return initial value of thread local variable associated with current thread. The default implementation of this method return null. To customize our own initial value, we have to override this method.

Void set(Object newValue) // To set a new value.

Void remove() // to remove the value of ThreadLocal variable associated with current Thread. It is newly added method in 1.5 version. After removal if we are trying to access it will be re-initialized once again by invoking it’s initial value method.

Example :

**package** com.Example2;

**public** **class** ThreadLocalDemo1 {

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

ThreadLocal<String> tl = **new** ThreadLocal<String>();

System.***out***.println(tl.get());

tl.set("Durga");

System.***out***.println(tl.get());

tl.remove();

System.***out***.println(tl.get());

}

}

O/P:

null

Durga

Null

Overriding of initialValue method:

**package** com.Example2;

**public** **class** ThreadLocalDemo2 {

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

ThreadLocal<String> tl = **new** ThreadLocal<String>()

{

**public** String initialValue()

{

**return** "abc";

}

};

System.***out***.println(tl.get());

tl.set("Durga");

System.***out***.println(tl.get());

tl.remove();

System.***out***.println(tl.get());

}

}

O/P

abc

Durga

Abc

Another Example:

CustomerThread.java

**package** com.Example2;

**public** **class** CustomerThread **extends** Thread {

**static** Integer *custid*=0;

**private** **static** ThreadLocal *tl* = **new** ThreadLocal()

{

**protected** Integer initialValue()

{

**return** ++*custid*;

}

};

**public** CustomerThread(String name) {

**super**(name);

}

@Override

**public** **void** run() {

System.***out***.println(Thread.*currentThread*().getName()+"executing with customerid:"+*tl*.get());

}

}

ThreadLocalDemo3.java

**package** com.Example2;

**public** **class** ThreadLocalDemo3 {

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

CustomerThread c1 = **new** CustomerThread("Cust Thread-1");

CustomerThread c2 = **new** CustomerThread("Cust Thread-2");

CustomerThread c3 = **new** CustomerThread("Cust Thread-3");

CustomerThread c4 = **new** CustomerThread("Cust Thread-4");

c1.start();

c2.start();

c3.start();

c4.start();

}

}

O/P

Cust Thread-3executing with customerid:1

Cust Thread-1executing with customerid:3

Cust Thread-4executing with customerid:2

Cust Thread-2executing with customerid:1

In above program for every customer thread a separate customer Id will be maintained by ThreadLocal object.

**JAVA.UTIL.CONCURRENT package**

1. **Executor:** Executor is an Interface that represent an object that executes provided task. If the executor can’t accept task for execution then it will throw RejectedExecutionException
2. **ExecutorService** is a complete solution for asynchronous processing. It manages an in-memory queue and schedules submitted tasks based on thread availability.

To use ExecutorService, we need to create one Runnable class.

|  |  |
| --- | --- |
|  | public class Task implements Runnable {      @Override      public void run() {          // task details      }  } |

Now we can create the ExecutorService instance and assign this task. At the time of creation, we need to specify the thread-pool size.

|  |  |
| --- | --- |
|  | ExecutorService executor = Executors.newFixedThreadPool(10); |

If we want to create a single-threaded ExecutorService instance, we can use newSingleThreadExecutor(ThreadFactory threadFactory) to create the instance.

Once the executor is created, we can use it to submit the task.

|  |  |
| --- | --- |
|  | public void execute() {      executor.submit(new Task());  } |

1. **ScheduledExceutorService**: ScheduledExecutorService is a similar interface to ExecutorService, but it can perform tasks periodically.
2. Executor and ExecutorService‘s methods are scheduled on the spot without introducing any artificial delay. Zero or any negative value signifies that the request needs to be executed instantly.
3. ***CountDownLatch*** (introduced in JDK 5) is a utility class which blocks a set of threads until some operation completes.

The CountDownLatch class is another helpful class for thread synchronization from the JDK. Similar to the Semaphore class it provides a counter, but the counter of the CountDownLatch can only be decreased until it reaches zero. Once the counter has reached zero all threads waiting on the CountDownLatch can proceed. Such functionality is often necessary when all threads of a pool have to synchronize at some point in order to proceed. A simple example would be an application that has to gather data from different sources before being able to store a new data set to the database.

1. **CyclicBarrier** : In contrast to the CountDownLatch, the CyclicBarrier class implements a counter that can be reset after being counted down to zero. All threads have to call its method await() until the internal counter is set to zero. The waiting threads are then woken up and can proceed. Internally the counter is then reset to its original value and the whole procedure can start again
2. **Semaphore** :

Semaphores are used to control access to a shared resource. In contrast to simple synchronized blocks a semaphore has an internal counter that is increased each time a thread acquires a lock and decreased each time a thread releases a lock it obtained before. The increasing and decreasing operations are of course synchronized, hence a semaphore can be used to control how many threads pass simultaneously through a critical section. The two basic operations of a thread are:

Void acquire();

Void release();

The constructor takes next to the number of concurrently locks a fairness parameter. The fairness parameter decides if new threads, which try to acquire a lock, are set at the beginning or at the end of the list of waiting threads. Putting the new thread at the end of the threads guarantees that all threads will acquire the lock after some time and hence no thread starves

Semaphore(int permits, Boolean fair)

1. **Automic Variables**:

When we have multiple threads sharing a single variable, we should provide synchronized access to variable. The operation like i++ is also not automic, because it contains 3 operation. To have synchronized access we can use like below.

Synchronized(i)

{ i++;

}

But current thread has to acquire lock and this is slow process called as ‘pessimistic locking’. To go for optimistic locking JAVA provided below package, which is based on compare-and-swap approach.

|  |
| --- |
| int currentValue = getValueAtMemoryPosition(pos); |

|  |  |
| --- | --- |
|  | if(currentValue == expectedValue) { |

|  |  |
| --- | --- |
|  | setValueAtMemoryPosition(pos, newValue); |

|  |  |
| --- | --- |
|  | } |

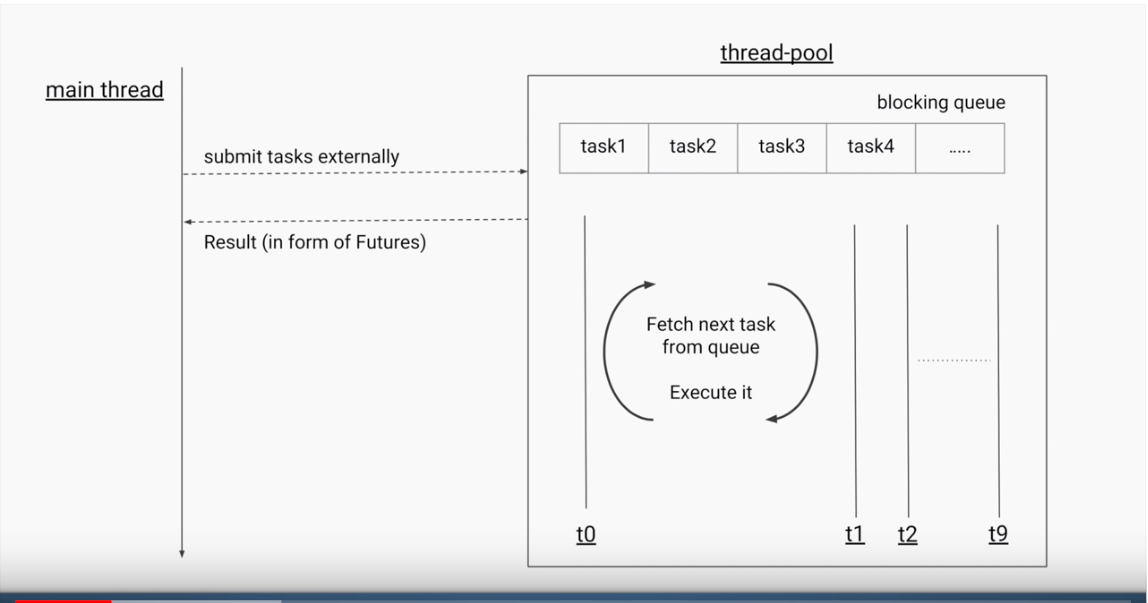
java.util.concurrent.atomic provides a means of efficiently updating the value of a variable without the use of locks.

classes-->AtomicInteger and AtomicLong, and methods -->

Boolean compareAndSet(int expect,int update), decrementAndGet( ), and getAndSet( ). These methods execute as a single, non-interruptible operation

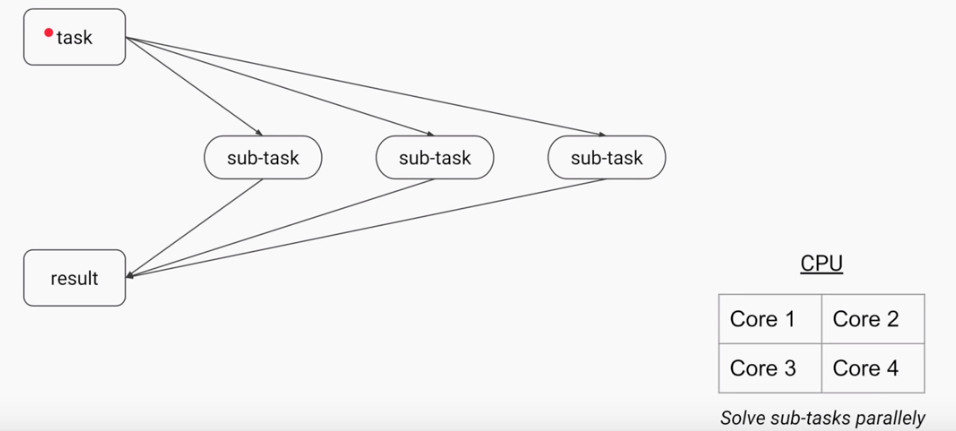
**Fork-join framework:**

In this we can achieve multithreading support for multi core processor. Executor service, contains a Thread pool which works on set of tasks which submitted externally, these tasks are stored in some form of data structure like BlockingQueue. Thread pools, threads take the task from BlockingQueue executes it until all task finishes.

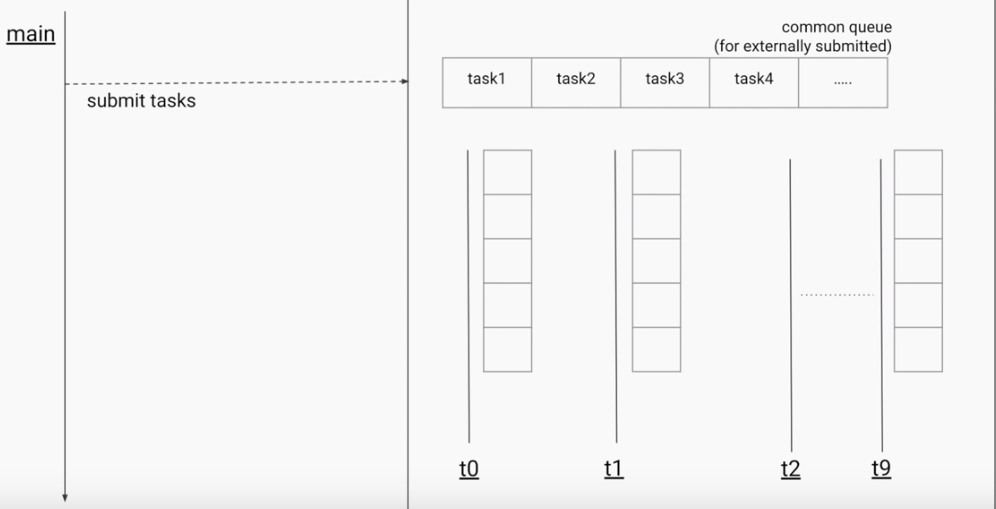


Fork-join is exactly same as Executor service, which submits the task, executes it and optionally gets return value from those task. So what is difference? It is different in 2 ways.

1.Tasks produces sub-tasks (fork-join): every task, divided into sub-tasks and assigned to different CPU Cores and then joined together.



2.Per-thread queuing or Work stealing: If thread is idle in thread pool, then he can steal work of other thread to reduce workload.



To summarize, Fork-join pool is same as executor service, internally it just uses double ended queue(local queue) for each of the thread and all the sub-tasks each thread produces, they are all stored within their own local queue. Fork-join perform really well when there is no synchronization, no shared variables, no blocking i/o operations.

Mostly they are used for Sorting, Matrix multiplication, Tree Traversal.

|  |  |
| --- | --- |
| Thread states | Running, ready to run, suspended, blocked, resumed, terminated |
| Monitor | Each object has its own implicit monitor that is automatically entered when one of the object’s synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. |
| Messaging | Java’s messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out |
|  | currentThread() is public static member of Thread class |
| Runnable | Runnable, Thread object, start , run |
| Thread | Thread, super(), start(), run() |
|  | isAlive(), join() --> called on object of Thead. |
|  | sleep() --> static method called on Thead class. |
|  | MIN\_PRIORITY, NORM\_PRIORITY, MAX\_PRIORITY |
|  | Monitor is object which is used as mutual exclusive lock. There can be only one monitor for given object irrespective of number of synchronized blocks inside class. |
|  | All objects have implicit monitor. Access it by calling method with Synchronized keyword. |
| race condition | calling same method on same object at the same time |
|  | two separate threads cannot enter synchronized methods on same instance and at the same time |
| two ways | synchronized methods and synchronized statements |
|  | wait(),notify(),notifyall() --> final methods of Object class. Can be only called through synchronized content. |
| fork,join | multithreading support for multi-core processors |
| Concurrent API | synchronizers, callable threads, and executors |
|  | features, such as semaphores, thread pools, and execution managers, that facilitate the creation of intensively concurrent programs |
|  | **java.util.concurrent** **(two subpackages: java.util.concurrent.atomic and java.util.concurrent.locks)** java.util.concurrent defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication. It defines the following key features: • Synchronizers (Semaphore, countdownlatch,cyclicbarrier,exchanger,phaser) • Executors (ExecutorService: ThreadPoolExecutor, ScheduledThreadPoolExecutor, and ForkJoinPool) • Concurrent collections (ConcurrentHashMap, ConcurrentLinkedQueue, and CopyOnWriteArrayList) • The Fork/Join Framework2 (ForkJoinTask, ForkJoinPool, RecursiveTask, and RecursiveAction) |
|  | java.util.concurrent.atomic provides a means of efficiently updating the value of a variable without the use of locks. Classes-->AtomicInteger and AtomicLong, and methods --> compareAndSet( ), decrementAndGet( ), and getAndSet( ). These methods execute as a single, non-interruptible operation |
|  | java.util.concurrent.locks provides an alternative to the use of synchronized methods. At the core of this alternative is the Lock interface, which de􀉹nes the basic mechanism used to acquire and relinquish access to an object. The key methods are lock( ), tryLock( ), and unlock( ). The advantage to using these methods is greater control over synchronization. |
|  | semaphor --> count, aquire(), release() --> Only one semaphore object needs to be shared among concurrent threads |
|  | countdownlatch -->await(), countdown() --> behaves like latch |
|  | cyclicbarrier --> await() --> new action on counts reach |
|  | exchanger --> exchange() --> call on object to exchange |
|  | executor --> execute() method will start thread. No thread starting from constructor. |
|  | locks --> alternative to synchronized --> lock(), trylock, unlock() |
| Volatile | tells compiler to refer to master copy of variable instead of threads copy of variable stored locally |
| Transient | variable with transient modifier will not be stored when its object is persisted |

**MultiThreading Interview questions:**

[**https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers**](https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers)

**Collections**

An Array is an indexed collection of fixed number of homogeneous data elements, the main advantage of arrays is we can represent multiple values by using single variable so that readability of the code will be improved.

Limitations of Arrays.

* Arrays are fixed in size
* Homogeneous nature; we can’t hold heterogeneous data types elements.
* No underlying data structure for arrays hence readymade method support is not available.

To overcome above problems of Arrays we should go for Collections concept.

* Collections are grow able in nature. i.e. based on our requirement we can increase or decrease the size
* Collections can hold both homogeneous and heterogeneous objects
* Every collection class is implemented based on some standard data structure hence for every requirement readymade method support is available.

|  |  |
| --- | --- |
| **Arrays** | **Collections** |
| Fixed in size | Grow able in nature |
| With respect to memory – not recommended | With respect to memory – recommended |
| With respect to performance – recommended | With respect to performance – not recommended |
| Only homogeneous | Both homo and heterogeneous. |
| Readymade support is not there | Readymade support |
| Array can hold both primitives and objects | Collection can hold only objects but not primitives |

Collection Framework: it contains several classes and interfaces which can be used to represent a group of individual objects as a single entity.

9key interfaces of collection framework.

1.Collection 2.List 3.Set 4.SortedSet 5.NavigableSet 6.Queue 7.Map 8.SortedMap 9.NavigableMap

**Collection(I):** If we want to represent a group of individual objects as single entity then we should go for collection. Collection interface defines the most common methods which are applicable for any collection object. In general collection interface is considered as root interface of collection framework. There is no concreate class which implements collection interface directly.

Difference between collection and collections;

Collection is an interface, if we want to represent a group of individual objects as a single entity then we should go for collection.

Collections is an utility class present in java.util package to define several utility methods for collection objects. Like sorting, searching etc.

**List(I)**: it is the child interface of collection(I), if we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order must be preserved then we should go for List.

Below implementation classes for List(I):ArrayList, LinkedList, Vector -> Stack

**Set(I)**: It is child interface of collection(I), if we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order not required then we should go for Set interface.

Implementation classes : HashSet -> LinkedHashSet

**SortedSet(I)**:It is child interface of Set(I), if we want to represent a group of individual objects as a single entity where duplicates are not allowed and all objects should be inserted according to some sorting order then we should go for sorted set.

**NavigableSet(I)**: it is the child interface of SortedSet(I), it contains several methods for navigation purposes.

Implementation classes:TreeSet

|  |  |
| --- | --- |
| List(I) | Set(I) |
| Duplications are allowed | Duplicates are not allowed |
| Insertion order preserved | Insertion order not preserved |

**Queue(I)**: It is child interface of Collection(I),if we want to represent group of individual object prior to processing then we should go for Queue(I). Usually Queue follows FIFO order but based on our requirement we can implement our own priority order also. Ex: Before sending a mail, all mail-id’s we have to store in some data structure; in which order we added mail id’s in the same order only mail should be delivered. For this requirement Queue is best choice.

Implementation classes: PriorityQueue , BlockingQueue -> (PriorityBlockingQueue, LinkedBlockingQueue)

Note: All the above interfaces, meant for representing group of individual objects, if we want to represent a group of objects as key value pairs then we should go for Map.

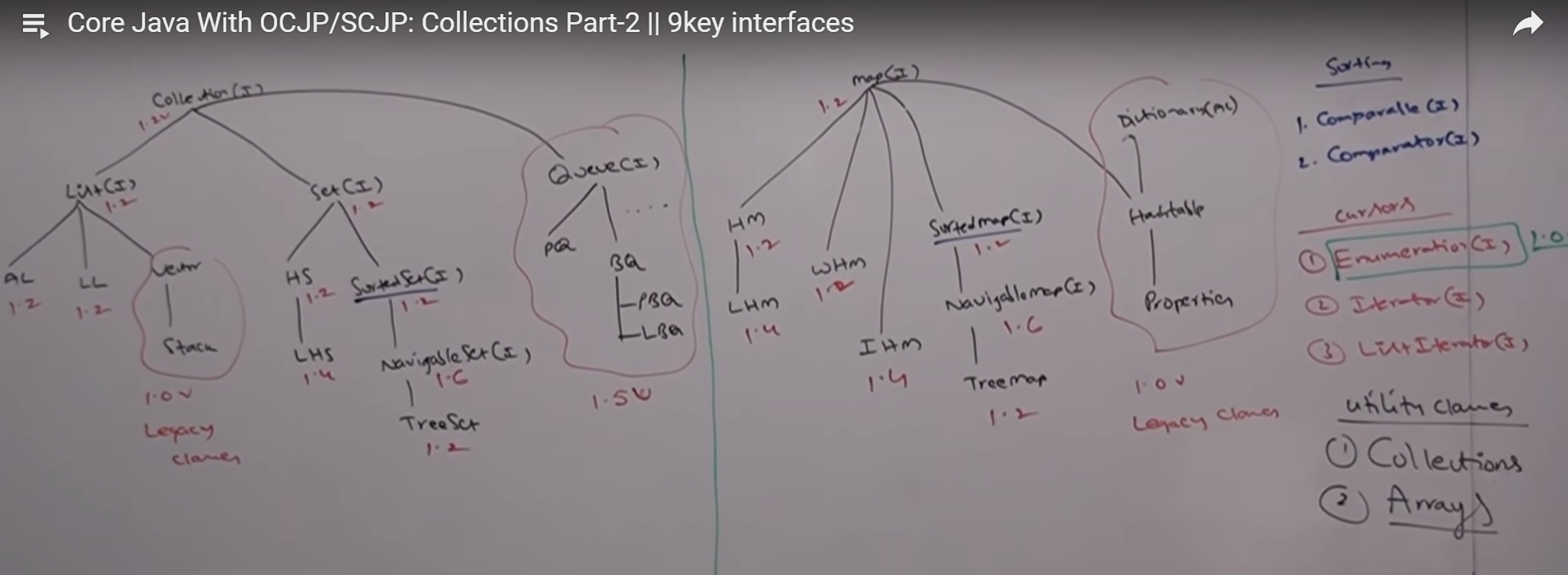
**Map(I):** Map is not child interface of collection(I), if we want to represent a group of objects as key-value pairs then we should go for Map(I).Both key and value are objects only, duplicates keys are not allowed but values can be duplicated.

Implementation classes: HashMap ->LinkedHashMap , WeakHashMap, IdentityHashMap, Hashtable

**SortedMap(I):** It is child interface of Map(I), if we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap(I).in SortedMap the sorting should be based on Key but not based on value.

**NavigableMap(I):** it is child interface of SortedMap(I) . it define several methods for navigation purposes.

Implementation class: TreeMap



**Collection(I):**

if we want to represent a group of individual object as single entity then we should go for collection. Collection(I) defines the most common methods which are applicable for any collection object.

Most common generalized methods present in Collection(I):

Boolean add(Object O)

Boolean addAll(Collection c)

Boolean remove(Object o)

Boolean removeAll(Collection c)

Boolean retainAll(Collection c) : To remove all objects except those present in c

Void clear()

Boolean contains(Object o)

Boolean containsAll(Collection c)

Boolean isEmpty()

Int size()

Object[] toArray()

Iterator iterator()

Note: There is no concrete class which implements collection interface directly.

**List(I):**

it is the child interface of collection(I), if we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order must be preserved then we should go for List

we can preserve insertion order via index and we can differentiate duplicates objects by using index hence index will play very important role in List.

List(I) defines the following specific methods:

Void add(int index, Object o)

Boolean addAll(int index, Collection c)

Object get(int index)

Object remove(int index)

Object set(int index, Object new):

Int indexOf(Object o) returns index of first occurrence of ‘o’

Int lastIndexOf(Object o)

ListIterator listIterator();

Implementation classes of List(I)

1.ArrayList 2.LinkedList 3.Vector - > Stack

**ArrayList** : -underline data structure is Resizable Array or grow able array

* Duplicates allowed
* Insertion order preserved
* Heterogeneous objects allowed ( Excepet TreeSet & TreeMap)
* Null insertion is possible

Constructors –

1. ArrayList l = new ArrayList()

Creates an empty ArrayList objects with default initial capacity 10, once arrayList reaches it’s max capacity then a new arraylist object will be created with

*New capacity = current capacity \* (3/2) + 1*

1. ArrayList l = new ArrayList(int initialcap)

Creates an empty Arraylist objects with specified initial capacity

1. ArrayList l = new ArrayList(Collection c)

Creates an equivalent arraylist object for the given collection.

Example:

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** ArrayListDemo1 {

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

{

ArrayList l = **new** ArrayList();

l.add("A");

l.add(10);

l.add("A");

l.add(**null**);

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

l.remove(2);

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

l.add(2,"M");

l.add("N");

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

}

}

O/P:

[A, 10, A, null]

[A, 10, null]

[A, 10, M, null, N]

Note: Usually we can use collections to hold and transfer objects from one location to another location(Container), to provide support for this requirement every collection class by default implements serializable and clonable interfaces.

ArrayList and Vector classes implements RandomAccess interface so that any random element we can access with the same speed.

RandomAccess : RandomAccess present in java.util package and it doesn’t contains any methods. It is a marker interface. Where required ability will be provided automatically by the JVM.

ArrayList is the best choice if our frequent operation is retrieval operation because ArrayList implements Random-access interface but ArrayList is worst choice if our frequent operation is insertion or deletion in the middle.

Difference between ArrayList and Vector

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| Every method present is non- Synchronized | Every method present is Synchronized |
| At a time multiple Threads are allowed to operate on ArrayList object hence it is not Thread safe | At a time only one thread is allowed to operate on vector object so it is thread safe |
| Relatively performance is high because Threads are not required to wait to operate on ArrayList object | Relatively performance is low because Threads are required to wait to operate on Vector object |

How to get Synchronized version of ArrayList object?

By default ArrayList is non-synchronized, but we can get Synchronized version of ArrayList object by using synchronizedList() method of collection class.

Public static List synchronizedList(List l)

EX: ArrayList l = new ArrayList()

List l1 = Collections.synchronizedList(l);

Similarly we can get synchronized version of set and map objects by using following methods of collections class

Public static Set synchronizedSet(Set s)

Public static Map synchronizedMap(Map m)

**LinkedList :** Underlying data structure is doubly link list

* Insertion order is preserved
* Duplicate objects are allowed
* Heterogeneous objects are allowed
* Null insertion is possible
* LinkedList implements Serializable and clonable interfaces but not random access
* LinkedList is the best choice, if our frequent operation is insertion or deletion in the middle
* LinkedList is the worst choice, if our frequent operation is retrieval operation.

Constructors:

LinkedList l = new LinkedList : Creates empty linked list object

LinkedList l = new LinkedList(Collection c) : Creates an equivalent linked list object for the given collection.

LinkedList class specific methods:

Usually we can use linkedlist to develop stacks and queues. To provide support for this requirement, linkedlist class defines the following specific methods.

Void addFirst(Object o)

Void addLast(Object o)

Object getFirst()

Object getLast()

Object removeFirst()

Object removeLast()

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** LinkedListDemo1 {

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

{

LinkedList l = **new** LinkedList();

l.add("Durga");

l.add(30);

l.add("Software");

l.add(**null**);

l.add("Vaibhav");

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

l.set(0,"M");

l.add("Fur");

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

l.addFirst("Example");

l.removeLast();

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

}

}

O/P:

[Durga, 30, Software, null, Vaibhav]

[M, 30, Software, null, Vaibhav, Fur]

[Example, M, 30, Software, null, Vaibhav]

Difference between ArrayList & LinkedList

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| Best when frequent operation is retrieval | Best when frequent operation is Insertion/deletion in middle |
| Worst when frequent operation is insertion/deletion in middle because internally several shift operations are performed | Worst when frequent operation is retrieval |
| ArrayList elements are stored in consecutive memory location | LinkedList elements are not stored in consecutive memory location |

**Vector:**

* Underlying data structure is resizable array or grow able array.
* Insertion order is preserved
* Duplicate objects are allowed
* Heterogeneous objects are allowed
* Null insertion is possible
* Vector implements Serializable and clonable interfaces and random access interface
* Every method present in Vector is synchronized and hence vector object is thread safe.

Constructor:

Vector v = new Vector() : Creates an empty vector objects with default initial capacity 10, once vector reaches it’s max capacity then a new vector object will be created with new capacity = current capacity \* 2

Vector v = new Vector(int initial-capacity) : Creates an empty vector object with specified initial capacity

Vector v = new Vector(int initial-capacity, int incremental-capacity)

Vector v = new Vector(Collection c): Creates an equivalent vector object for given collection, this constructor meant for interconversion between collection objects

Vector specific methods:

To add objects

Add(Object o) – collection specific

Add(int index,Object o) – List specific

AddElement(Object o)- vector specific

To remove objects

Remove(Object o) – C

RemoveElement(Object o) – V

Remove(int index) – L

RemoveElementAt(int index) – V

Clear() – C

RemoveAllElements() – V

To get objects

Object get(int index) – L

Object elementAt(int index) – V

Object firstElement() – V

Object lastElement() – V

Other methods

int size()

int capacity()

Enumeration elements()

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** VectorDemo {

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

//Vector v = new Vector();

// Vector v = new Vector(15);

Vector v = **new** Vector(10,3);

System.***out***.println(v.capacity());

**for**(**int** i = 0; i<10;i++)

{

v.addElement(i);

}

System.***out***.println(v.capacity());

v.add("AB");

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

System.***out***.println(v.capacity());

}

}

O/P

10

10

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, AB]

13

Stack: It is the child class of vector, it is specially designed class for last in first out order

Constructor:

Stack s = new stack()

Methods:

Object push(Object o)

Object pop(): to remove and return top of the stack

Object peek(): to return top of the stack without removal

Boolean empty(): return true if stack is empty

Int search(Object o) : returns offset if element is present else returns -1

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** StackDemo {

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

Stack s = **new** Stack();

s.push("A");

s.push("B");

s.push("C");

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

System.***out***.println(s.peek());

System.***out***.println(s.pop());

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

System.***out***.println(s.search("A"));

System.***out***.println(s.search("Z"));

}

}

O/P:

[A, B, C]

C

C

[A, B]

2

-1

**3 cursors of Java**

If we want to get objects one by one from collection then we should go for cursor, there are 3 types of cursors available in java

* Enumeration
* Iterator
* ListIterator

Enumeration: we can use Enumeration to get objects one by one from legacy collection object. We can create enumeration objects by using elements() method of vector class.

Public Enumeration elements();

Ex:

Vector v = new vector();

Enumeration e = v.elements();

Methods

Public Boolean hasMoreElements();

Public Object nextElement();

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** EnumerationDemo {

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

Vector v = **new** Vector();

**for**(**int** i = 0; i<=10;i++)

{

v.addElement(i);

}

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

Enumeration e = v.elements();

**while**(e.hasMoreElements())

{

Integer i = (Integer)e.nextElement();

**if**(i%2==0)

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

}

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

}

}

O/P:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

0

2

4

6

8

10

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Limitations of Enumeration:

* We can apply Enumeration concept only for legacy classes and it is not universal cursor
* By using Enumeration we can get only read access and we can’t perform remove operation

To overcome these limitations we should go for Iterator.

Iterator:

we can apply Iterator concept for any collection object and hence it is universal cursor, by using Iterator we can perform both read and remove operations.

Public Iterator iterate()

Example

Iterator itr = c.iterate() // here c is any collection object.

Methods:

Public Boolean hasNext()

Public Object next()

Public void remove()

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** IteratorDemo {

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

ArrayList l = **new** ArrayList();

**for**(**int** i = 0; i <= 10 ; i++)

{

l.add(i);

}

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

Iterator itr = l.iterator();

**while**(itr.hasNext())

{

Integer i = (Integer)itr.next();

**if**(i%2 == 0)

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

**else**

itr.remove();

}

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

}

}

O/P:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

0

2

4

6

8

10

[0, 2, 4, 6, 8, 10]

Limitations of Iterator:

By using enumeration and Iterator we can always move only towards forward direction and we can’t move toward backward direction, these are single direction cursors but not Bi-directional cursor.

By using Iterator, we can perform only read and remove operations and we can’t perform replacement and addition of new objects.

To overcome above limitations we should go for ListIterator.

**ListIterator:**

By using ListIterator we can move bi-directional

By suing ListIterator we can perform Replacement and addition of new objects in addition to Read and Remove operations.

We can create ListIterator by using listIterator() method

Public ListIterator listIterator()

EX: ListIterator ltr = l.listIterator() // Here l is any List object.

Methods:

Forward operation

Public Boolean hasNext()

Public Object next()

Public int nextIndex()

Backward operation

Public Boolean hasPrevious()

Public Object Previous()

Public int previousIndex()

Extra operation

Public void remove()

Public void add(Object o)

Public void set(Object o)

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** ListIteratorDemo {

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

LinkedList l = **new** LinkedList();

l.add("Mahesh");

l.add("Supriya");

l.add("Manisha");

l.add("Mohini");

l.add("Aarati");

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

ListIterator ltr = l.listIterator();

**while**(ltr.hasNext())

{

String str =(String) ltr.next();

**if** (str.equals("Mahesh"))

ltr.remove();

**else** **if**(str.equals("Supriya"))

ltr.add("Rani");

**else** **if**(str.equals("Manisha"))

ltr.set("Snehal");

}

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

}

}

O/P:

[Mahesh, Supriya, Manisha, Mohini, Aarati]

[Supriya, Rani, Snehal, Mohini, Aarati]

The most powerful cursor is ListIterator but it’s limitation is – It is applicable only for List objects.

Comparison of 3 cursors.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Enumeration** | **Iterator** | **ListIterator** |
| Where we can apply? | Applicable only for Legacy class | Applicable for any collection object | Applicable only for List objects |
| Is it Legacy? | Yes | No; came in 1.2 version | No; came in 1.2 version |
| Movement | One directional – Forward | One directional – forward | Bi-directional |
| Allowed operations | Only Read | Read and Remove | Read/Add/Remove/Replace |
| How we can get? | By using elements() of Vector class | By using iterator() of collection interface | By using listIterator() of List interface |
| What methods? | hasMoreElements()  hasNextElements() | hasNext()  next()  remove() | hasNext()  next()  nextIndex()  hasPrevious()  previous()  previousIndex()  add()  set()  remove() |

Internal Implementation of Cursors

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** CursorsDemo {

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

Vector v = **new** Vector();

Enumeration e = v.elements();

Iterator itr = v.iterator();

ListIterator ltr = v.listIterator();

System.***out***.println(e.getClass().getName());

System.***out***.println(itr.getClass().getName());

System.***out***.println(ltr.getClass().getName());

}

}

Output

java.util.Vector$1

java.util.Vector$Itr

java.util.Vector$ListItr

**Set(I)**

Implementation

HashSet - >LinkedHashSet

SortedSet(I) -> NavigableSet(I) ->TreeSet

* Child interface of collection
* Group of individual objects as single entity
* Duplicates are not allowed
* Insertion order not preserved

Set interface doesn’t contain new methods so we have to use only methods available in collections

**HashSet**

* Underlying data structure is Hash table
* Duplication not applicable
* Insertion order not preserved but all objects inserted based on object hash code
* Null insertion possible but only once
* Heterogeneous objects are allowed
* Implements serializable and clonable interface
* HashSet is the best choice if our frequent operation is Search

Note: if we try to add duplicate we will not get any compilation or run time error ; and add method returns FALSE.

Constructors:

HashSet h = new HashSet() : Creates an empty HashSet object with default initial capacity 16. And Default fill ratio is 0.75.

HashSet h = new HashSet(int initialcapcity): Creates an empty HashSet object with provided initial capacity And Default fill ratio is 0.75.

HashSet h = new HashSet(int initialcapcity, float fillratio) : Creates an empty HashSet object with provided initial capacity And specified fill ratio.

HashSet h = new HashSet(Collection c) : Creates an equivalent hash set for given collection.This constructor meant for interconversion between collection objects.

Note: After filling how much ratio, a new HashSet object will be created. This ratio is called Fill Ration or Load factor. For Ex: Fill Ratio 0.75 means, after filling 75% ratio a new Hashset object will be created.

Example

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** HashSetDemo {

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

HashSet h = **new** HashSet();

h.add("A");

h.add("C");

h.add(10);

h.add("Z");

h.add(**null**);

h.add("K");

System.***out***.println(h.add("K"));

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

}

}

O/P

false

[null, A, C, 10, Z, K]

**LinkedHashSet:**

It is child class of HashSet, exactly same as HashSet(Including constructors and methods) except following differences

|  |  |
| --- | --- |
| **HashSet** | **LinkedHashSet** |
| Underlying data structure is Hash table | Underlying data structure is linked list + Hash table |
| Insertion order is not preserved | Insertion order is preserved |
| Introduced in 1.2 Version | 1.4 version |

Example:

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** LinkedHashSetDemo {

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

{

LinkedHashSet lh = **new** LinkedHashSet();

lh.add("A");

lh.add("C");

lh.add(10);

lh.add("Z");

lh.add(**null**);

lh.add("K");

System.***out***.println(lh.add("K"));

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

}

}

O/P

false

[A, C, 10, Z, null, K]

Note: In general we can use LinkedHashSet to develop cache based applications where duplicated are not allowed and insertion order must be preserved.

SortedSet(I):

* Is the child interface of Set(I)
* If we want to represent a group of individual objects according to some sorting order without duplicates then we should go for SortedSet
* SortedSet interface defines following specific methods.

Object first() : returns first element of SortedSet

Object last(): returns last element of SortedSet

SortedSet headset(Object o) : returns sorted set whose elements are less than obj

SortedSet tailSet(Object o): returns SortedSet whose elements are >= obj

SortedSet suSet(Object o1,Object o2); returns SortedSet whose elements are >= o1 and < o2

Comparator comparator(): returns comparator object that describes underlying sorting technique

Note: the default natural sorting order for number is Ascending and for String object is Alphabetical order.

SortedSet is interface so no demo program ☺

**TreeSet:**

* Underlying data structure is balanced tree,
* Duplicates are not allowed
* Insertion order not preserved
* Heterogeneous objects are not allowed
* Null object allowed but only once( not after 1.7 version)
* Implements serializable and clonable interface but not random access.
* All objects will be inserted based on some sorting order it may be Default natural sorting order or customized sorting order

Constructors:

TreeSet t = new TreeSet(): Creates an empty TreeSet object where the elements will be inserted according to default natural sorting order.

TreeSet t = new TreeSet(Comparator c): Creates an empty TreeSet object where the elements will be inserted according to customized sorting order specified by Comparator object.

TreeSet t = new TreeSet(Collection c)

TreeSet t = new TreeSet(SortedSet s)

Examples:

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** TreeSetDemo {

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

TreeSet t = **new** TreeSet();

t.add("A");

t.add("a");

t.add("C");

t.add("L");

t.add("M");

// t.add(10); Class cast exception

System.***out***.println(t); // [A, C, L, M, a] – Default sorting applied

}

}

O/P:

[A, C, L, M, a]

NULL Acceptance: 1. For non-empty tree set if we are trying to insert NULL we will get NULLPOINTEREXCEPTION

2.For empty tree set NULL as first element is acceptable but if we try to add another element in same tree set we will get NULLPOINTEREXCEPTION

\*After 1.7 version NULL is not allowed even as first element for empty tree set.

Example 2;

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** TreeSetDemo1 {

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

TreeSet t = **new** TreeSet();

t.add(**new** StringBuffer("A"));

t.add(**new** StringBuffer("C"));

t.add(**new** StringBuffer("Z"));

t.add(**new** StringBuffer("U"));

}

}

O/P: runtime exception : ClassCastException

If we are dependent on default natural sorting order compulsory the objects should be homogeneous and comparable, otherwise we will get runtime exception : ClassCastException. An object is said to be comparable if and only if corresponding class implements comparable interface. String class and all wrapper classes already implements comparable interface but StringBuffer class doesn’t implement comparable interface hence we got above exception.

Comparable(I): it is present in java.lang package and it contains only one method compareTo().

Public int compareTo(Object obj)

Ex: obj1.compareTo(obj2) // obj1 = obj1 is object trying to insert; obj2 is already inserted

returns –ve : If and only if obj1 has to come before obj2

return +ve : if and only if obj1 has to come after obj2

returns 0 : If and only if obj1 and obj2 are equal.

Note: If default natural sorting order not available or if we are not satisfied with default natural sorting order then we can go for customized sorting by using comparator.

Comparable meant for default natural sorting order

Comparator meant for customized sorting order.

**Comparator**:

It is java.util package, defines 2 methods, compare and equals

Public int compare(Object obj1, Object obj2)

returns –ve : If and only if obj1 has to come before obj2

return +ve : if and only if obj1 has to come after obj2

returns 0 : If and only if obj1 and obj2 are equal.

Public Boolean equals(Object obj)

When ever we are implementing comparator interface compulsory we should provide implementation only for compare method and we are not required to provide implementation for equals methods because it is already available to our class from object class through inheritance

Write a program to insert integer objects into the treeset where sorting order is Descending order.

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** TreeSetDemo2 {

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

// TreeSet t = new TreeSet();

TreeSetDemo2 ex = **new** TreeSetDemo2();

TreeSet t = **new** TreeSet(ex.**new** MyComparator()); //Line 1

t.add(10);

t.add(0);

t.add(15);

t.add(5);

t.add(20);

t.add(20);

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

}

**class** MyComparator **implements** Comparator

{

**public** **int** compare(Object obj1, Object obj2)

{

Integer I1 = (Integer)obj1;

Integer I2 = (Integer)obj2;

**if**(I1<I2)

**return** +100;

**else** **if**(I1>I2)

**return** -100;

**else**

**return** 0;

}

}

} O/P: [20, 15, 10, 5, 0]

Here in above program @ line 1 if we are not passing comparator object then internally JVM will call compareTo() method which is meant for default natural sorting order so output is Ascending order. @ Line 1 if we are passing comparator object then JVM will call compare() method which is meant for customized sorting in this case output is Descending order.

Various possible implementation of compare method

**public** **int** compare(Object obj1, Object obj2)

{

Integer I1 = (Integer)obj1;

Integer I2 = (Integer)obj2;

/\* if(I1<I2)

return +100;

else if(I1>I2)

return -100;

else

return 0; \*/

//Possible approaches of implementation

//1

//return I1.compareTo(I2);

//2

//return -I1.compareTo(I2);

//3

//return I2.compareTo(I1);

//4

//return -I2.compareTo(I1);

//5

//return +1;

//6

//return -1;

//7

//return 0;

}

Write a program to insert String objects into TreeSet where all elements should be inserted according to reverse of Alphabetical order?

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** TreeSetDemo3 {

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

//TreeSet t = new TreeSet();

TreeSet t = **new** TreeSet(**new** MyComparator2());

t.add("Ganga");

t.add("Sonali");

t.add("Aarati");

t.add("Rani");

t.add("Mohini");

t.add("Sonam");

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

}

}

**class** MyComparator2 **implements** Comparator

{

**public** **int** compare(Object o1, Object o2)

{

String s1 = (String)o1;

String s2 = (String)o2;

**return** s2.compareTo(s1);

}

}O/P: [Sonam, Sonali, Rani, Mohini, Ganga, Aarati]

Write a program to insert String buffer objects into TreeSet where Sorting order is Alphabetical order?

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** TreeSetDemo4 {

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

TreeSet t = **new** TreeSet(**new** MyComparator4());

t.add(**new** StringBuffer("A"));

t.add(**new** StringBuffer("C"));

t.add(**new** StringBuffer("Z"));

t.add(**new** StringBuffer("U"));

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

}

}

**class** MyComparator4 **implements** Comparator

{

**public** **int** compare(Object o1, Object o2)

{

String s1 = o1.toString();

String s2 = o2.toString();

**return** s1.compareTo(s2);

}

}o/p: [A, C, U, Z]

Note: If we are depending on default natural sorting order compulsory object should be homogeneous and comparable otherwise we will get runtime CLASSCASTEXCEPTION. If we are defining our own sorting by comparator then objects need not be comparable and homogeneous.

Write a program to insert String and StringBuffer objects into TreeSet where sorting order is increasing length order?If 2 objects having same length then consider there Alphabetical order.

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** TreeSetDemo5 {

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

TreeSet t = **new** TreeSet(**new** MyComparator5());

t.add("AA");

t.add(**new** StringBuffer("AXX"));

t.add(**new** StringBuffer("ABC"));

t.add("XX");

t.add("ABCD");

t.add("A");

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

}

}

**class** MyComparator5 **implements** Comparator

{

**public** **int** compare(Object o1,Object o2)

{

String s1 = o1.toString();

String s2 = o2.toString();

**int** l1 = s1.length();

**int** l2 = s2.length();

**if**(l1<l2)

**return** -100;

**else** **if**(l1>l2)

**return** +100;

**else**

**return** s1.compareTo(s2);

}

}o/p: [A, AA, XX, ABC, AXX, ABCD]

Comparable Vs Comparator

1. For Predefined comparable classes default natural sorting order already available, if we are not satisfied with that default natural sorting order then we can define our own sorting by using comparator.
2. For predefined non-comparable classes(like StringBuffer),Default natural sorting order not already available we can define our own sorting by using comparator
3. For our own classes like Employee, the person who is writing the class is responsible to define default natural sorting order by implementing comparable interface. The person who is using our class, if he is not satisfied with default natural sorting order then he can define his own sorting by using comparator.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Comparable** | **Comparator** | <http://www.javacodegeeks.com/2013/03/difference-between-comparator-and-comparable-in-java.html> |
| Sorting logic | Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects | Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id,name etc. |  |
| Implementation | Class whose objects to be sorted must implement this interface.e.g Country class needs to implement comparable to collection of country object by id | Class whose objects to be sorted do not need to implement this interface.Some other class can implement this interface. E.g.-CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id |  |
| Sorting method | int compareTo(Object o1) | int compare(Object o1,Object o2) |  |
|  | This method compares this object with o1 object and returns a integer.Its value has following meaning | This method compares o1 and o2 objects. and returns a integer.Its value has following meaning. |  |
|  | 1. positive – this object is greater than o1 | 1. positive – o1 is greater than o2 |  |
|  | 2. zero – this object equals to o1 | 2. zero – o1 equals to o2 |  |
|  | 3. negative – this object is less than o1 | 3. negative – o1 is less than o1 |  |
| Calling method | Collections.sort(List) | Collections.sort(List, Comparator) |  |
|  | Here objects will be sorted on the basis of CompareTo method | Here objects will be sorted on the basis of Compare method in Comparator |  |
| Package | Java.lang.Comparable | Java.util.Comparator |  |
|  |  |  |  |
|  | only one sorting pattern | can change sorting pattern ba |  |

Example:

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**class** Employee **implements** Comparable

{

String Name;

**int** Eid;

Employee(**int** e, String n)

{

**this**.Name = n;

**this**.Eid = e;

}

**public** String toString()

{

**return** Eid+"---"+Name;

}

**public** **int** compareTo(Object o)

{

**int** eid1 = **this**.Eid;

Employee e = (Employee)o;

**int** eid2 = e.Eid;

**if**(eid1 < eid2)

**return** -1;

**else** **if**(eid1 > eid2)

**return** +1;

**else**

**return** 0;

}

}

**public** **class** EmployeeDemo {

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

{

Employee e1 = **new** Employee(123,"Sanap");

Employee e2 = **new** Employee(183,"Sathe");

Employee e3 = **new** Employee(83,"Maran");

Employee e4 = **new** Employee(103,"Kohali");

Employee e5 = **new** Employee(123,"Mali");

TreeSet t = **new** TreeSet();

t.add(e1);

t.add(e2);

t.add(e3);

t.add(e4);

t.add(e5);

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

//-------------Customized Approach---------------------------------------

TreeSet t1 = **new** TreeSet(**new** MyComparator6());

t1.add(e1);

t1.add(e2);

t1.add(e3);

t1.add(e4);

t1.add(e5);

System.***out***.println(t1);

}

}

**class** MyComparator6 **implements** Comparator

{

**public** **int** compare(Object o1, Object o2)

{

Employee e1 = (Employee) o1;

Employee e2 = (Employee) o2;

String S1 = e1.Name;

String S2 = e2.Name;

**return** S1.compareTo(S2);

}

}

O/P

[83---Maran, 103---Kohali, 123---Sanap, 183---Sathe]

[103---Kohali, 123---Mali, 83---Maran, 123---Sanap, 183---Sathe]

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| Default natural sorting order | Customized sorting order |
| Java.lang package | Java.util package |
| Only one method – compareTo() | 2 methods – equals() & compare() |
| String and all wrapper classes implement Comparable interface | The only implemented classes are Collator & RuleBasedCollator |

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| Underlying data structure | Hash Table | Linked List + Hash Table | Balanced Tree |
| Duplicates allowed | NO | No | No |
| Insertion Order | Not preserved | Preserved | Not preserved |
| Sorting Order | Not applicable | Not applicable | Applicable |
| Heterogeneous object | Allowed | Allowed | Not allowed by default |
| NULL acceptance | Allowed | Allowed | Not allowed |

**Map(I) :**

* Map is not child interface of collection(I)
* If we want to represent a group of objects as Key-Value pairs then we should go for Map.
* Both keys and values are objects
* Duplicates keys are not allowed but values can be duplicated
* Each key-value pair is called Entry. Hence map is considered as collection of Entry objects.

Methods

Object put(Object key, Object value)

To add one key value pair to the Map.If the key is already present then old value will be replaced with new value and return old value

Ex : m.put(101,”Durga) – returns NULL

m.put(102,”Shiva”) – returns NULL

m.put(101,”Ravi”) – Durga replaced with Ravi and returns Durga

Void putAll(Map m)

Object get(Object key) – returns the value associated with specified key

Object remove(Object key) – removes entry associated with specified key

Boolean containsKey(Objeck key)

Boolean containvalue(Object key)

Boolean isEmpty()

Int size()

Void clear()

Set keyset() – returns only key values

Collection values() – returns values

Set entrySet() – return Entry objects

Entry(I):

A map is a group of key-value pairs and each key-value pair is called an Entry. Hence Map is considered as a collection of Entry objects. Without existing Map object there is chance of existing Entry object hence Entry interface is defined inside Map interface

Interface Map

{

Interface Entry // These methods are Entry specific and we can apply only to Entry objects

{

Object getKey()

Object getvalue()

Object setValue(Object o)

}

}

**HashMap**

The underlying data structure is Hash table

Insertion order is not preserved and it is based on hash codes of keys

Duplicates keys are not allowed but values can be duplicated

Heterogeneous objects are allowed for both key and value

NULL allowed for Key(only once) and NULL allowed for Values(any number of times)

HashMap implements Serializable and Clonable interfaces but not random access.

HashMap is best choice, if our frequent operation is Search

Constructors

HashMap m = new HashMap() – creates an empty HashMap object with Default initial capacity 16 and Default Fill ratio – 0.75

HashMap m = new HashMap(int initialcapacity) - creates an empty HashMap object with specified initial capacity and default Fill ratio – 0.75

HashMap m = new HashMap(int initailcapacity, float fillratio)

HashMap m = new HashMap(Map m)

Example

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** HashMapDemo {

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

HashMap m = **new** HashMap();

m.put("Siren", 101);

m.put("Jaju",501);

m.put("Veleky",1001);

m.put("George",700);

System.***out***.println(m);//{K=V,K=V}

System.***out***.println(m.put("George",5001));

Set s = m.keySet();

System.***out***.println(s);//[K,K,K]

Collection c = m.values();

System.***out***.println(c);//[V,V,V]

Set s1 = m.entrySet();

System.***out***.println(s1);//[K=V,K=V,K=V]

Iterator itr = s1.iterator();

**while**(itr.hasNext())

{

Map.Entry m1 = (Map.Entry)itr.next();

System.***out***.println(m1.getKey() +"..."+ m1.getValue());

**if**(m1.getKey().equals("Jaju"))

{

m1.setValue(10000);

}

}

System.***out***.println(m);//{K=V,K=V}

}

}

O/P

{Jaju=501, Veleky=1001, George=700, Siren=101}

700

[Jaju, Veleky, George, Siren]

[501, 1001, 5001, 101]

[Jaju=501, Veleky=1001, George=5001, Siren=101]

Jaju...501

Veleky...1001

George...5001

Siren...101

{Jaju=10000, Veleky=1001, George=5001, Siren=101}

Difference between HashMap and HashTable

|  |  |
| --- | --- |
| HashMap | HashTable |
| All methods are not synchronized | All methods are Synchronized |
| Not thread safe | Thread safe |
| Relatively performance is high because Threads are not required to wait to operate on HashMap object | Relatively performance is low because Threads are required to wait to operate on HashMap object |
| NULL is allowed for both key-value | NULL is not allowed for keys-values otherwise we will NULLPOINTEREXCEPTION |
| Introduced in 1.2 version | Introduced in 1.0 version and it’s Legacy |

How to get Synchronized version of HashMap object?

By default HashMap is non-synchronized but we can get synchronized version of HashMap by using SynchronizedMap() of Collections class.

HashMap m = new HashMap();

Map m1 = Collections.synchronizedMap(m)

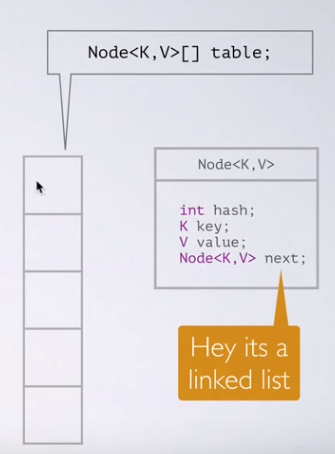
**How HashMap works internally?**

[**https://www.youtube.com/watch?v=c3RVW3KGIIE**](https://www.youtube.com/watch?v=c3RVW3KGIIE)

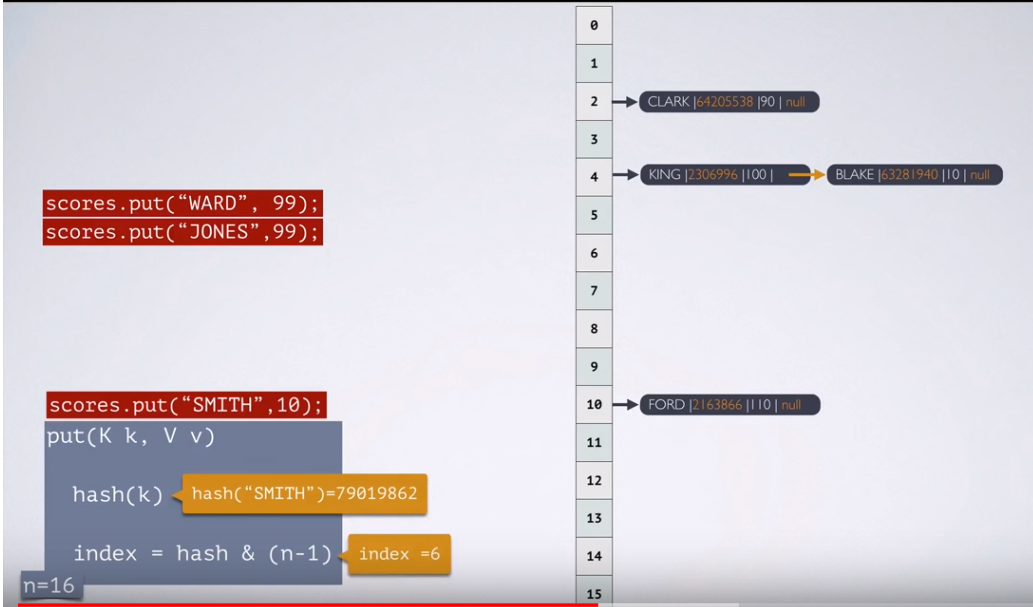
Map is an associative array data structure. Like key-value pairs. HashMap is implementation of Map interface, it is based on the technique called Hashing.

Hashing: Which transform large string (object) into the short fixed length representation. Shorter value helps in indexing and faster searches.

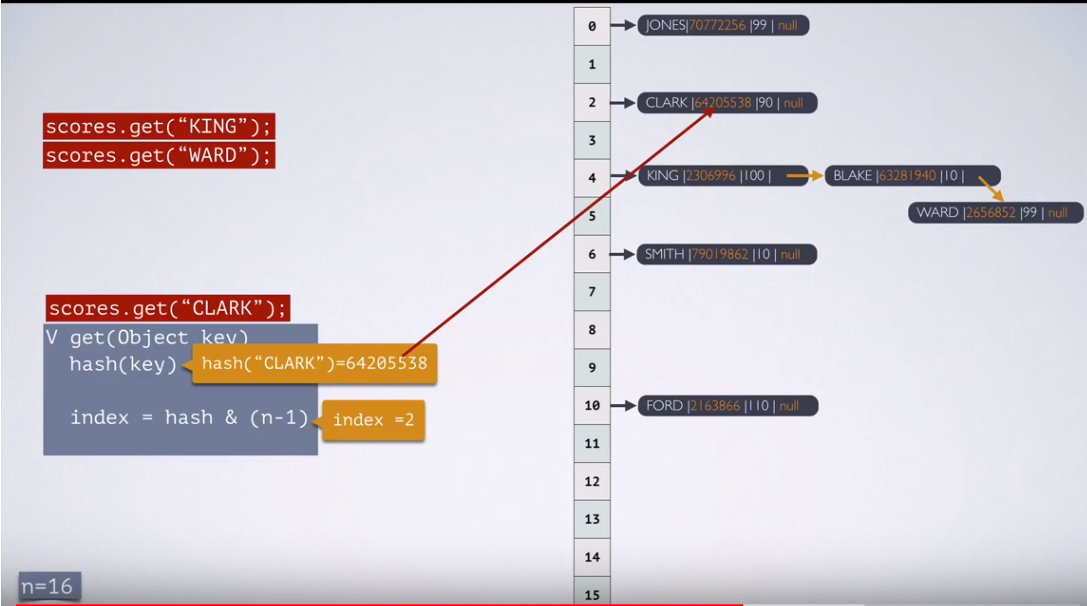
In java every object has a method public int hashcode() that will return a hash value for given object. We use hashvalue to store entries of Hashmap.



Put operation:



Get Operation:



**What has changed in JAVA-8?**

When we have too many unequal keys which gives same hashcode(index) – when the number of items in a hash bucket grows beyond certain threshold, the content of that bucket switches from using a linked list of entry objects to a balanced tree.

**LinkedHashMap:**

It is child class of HashMap

It is exactly same as HashMap(including methods and constructors except following difference)

|  |  |
| --- | --- |
| **HashMap** | **LinkedHashMap** |
| Underlying data structure – Hash Table | Underlying data structure – Linked list + Hash table |
| Insertion order not preserved and based on hash code of keys | Insertion order preserved |
| Introduced in 1.2 version | Introduced in 1.4 version |

Example

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** HashMapDemo {

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

LinkedHashMap m = **new** LinkedHashMap();

m.put("Siren", 101);

m.put("Jaju",501);

m.put("Veleky",1001);

m.put("George",700);

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

}

}

O/P

{Siren=101, Jaju=501, Veleky=1001, George=700} // Insertion order preserved

Difference between == operator and .equals method : In general == operator meant for reference comparison(address comparison) whereas .equals method meant for content comparison.

Ex: Integer I1 = new Integer(10);

Integer I2 = new Integer(10);

SOP(I1==I2) – false

SOP(I1.equals(I2)) – true

**IdentityHashMap:**

It is exactly same as HashMap(including methods & constructors except following difference)

In the case of HashMap, JVM will use .equals method to identify duplicate keys, which is meant for content comparison but in IdentityHashMap JVM will use == operator to identify duplicate keys which is meant for Reference comparison(Address Comparison).

Ex:

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** IdentityHashMap {

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

Integer i1 = **new** Integer(10);

Integer i2 = **new** Integer(10);

HashMap m = **new** HashMap();

m.put(i1,"Pawan");

m.put(i2,"Kalyan");

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

}

}o/p: {10=Kalyan}

If we replace above HashMap with IdentityHashMap then I1 & I2 are not duplicates keys because I1==I2 returns false

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** IdentityHashMap {

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

Integer i1 = **new** Integer(10);

Integer i2 = **new** Integer(10);

IdentityHashMap m = **new** IdentityHashMap();

m.put(i1,"Pawan");

m.put(i2,"Kalyan");

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

}{10=Kalyan,10=Pawan}

**WeakHashMap**

It is exactly same as HashMap except following difference

In the case of HashMap if object doesn’t have any reference and it is associated with HashMap then also it is not eligible for GC. i.e. HashMap dominates Garbage Collector. But in the case of WeakHashMap if object doesn’t contain any references it is eligible for GC even though object associated with WeakHashMap. I.e. GC dominates WeakHashMap.

Ex:

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** WeakHashMap {

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

Temp t = **new** Temp();

//HashMap h = new HashMap();

WeakHashMap h = **new** WeakHashMap();

h.put(t, "Java");

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

t=**null**;

System.*gc*();

Thread.*sleep*(5000);

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

}

}

**class** Temp

{

**public** String toString()

{

**return** "Temp";

}

**public** **void** finalise()

{

System.***out***.print("Garbage Collector Came");

}

}

o/p:

{Temp,Java}

Garbage collector Cam

{ }

**SortedMap**

Child interface of Map(I)

If we want to represent a group of key-value pairs according to some sorting order of keys then we should go for SortedMap.

Sorting is based on the Key but not based on Values

SortedMap defines the following specific methods.

Object firstKey()

Object lastKey()

SortedMap headMap(Object key)

SortedMap tailMap(Object key)

SortedMap subMap(Object key1,Object key2)

Comparator comparator()

Ex;

101=A, 103=B,104=C,107=D,125=E,136=F

firstKey() – 101

lastKey() – 136

headMap(107) – {101=A, 103=B,104=C}

tailMap(107) – {107=D,125=E,136=F}

subMap(103,125) – {103=B,104=C,107=D}

comparator – null

**TreeMap**

* Underlying data structure is RED-BLACK tree
* Insertion order not preserved
* All elements inserted based on some sorted order of keys
* Duplicate keys are not allowed but values can be duplicated
* If we are depending on default natural sorting order then keys should be homogeneous and comparable otherwise we will get run time exception saying classcastexception. If we are defining our own sorting by comparator then keys need not be homogenous and comparable, we can take heterogeneous non-comparable objects also. Whether we are depending on default natural sorting order or customized sorting order there are no restrictions for values we can take heterogeneous – non comparable objects.
* Null Acceptance as below

For non-empty Tree-map if we are trying to insert entry with NULL key then we will get NPE.

For empty tree map as the first entry with Null key is allowed but after inserting that entry, if we are trying to insert other entry then we will get NPE.

After 1.7 version NULL concept is not allowed in TreeMap

Constructors:

TreeMap t = new TreeMap(): For default natural sorting order

TreeMap t = new TreeMap(Comparator c): For customized sorting order

TreeMap t = new TreeMap(SortedMap m)

TreeMap t = new TreeMap(Map m):

Ex-1

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** TreeMapDemo1 {

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

TreeMap t = **new** TreeMap();

t.put(100, "Pawan");

t.put(80, "Kawan");

t.put(120, "RAwan");

t.put(230, "BAwan");

t.put(100, "HAwan");

t.put(90, 120);

//t.put("FFF", "Zawan");

//t.put(null, 100);

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

}

} {80=Kawan, 90=120, 100=HAwan, 120=RAwan, 230=BAwan}

Ex-2

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** TreeMapDemo2 {

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

TreeMap t = **new** TreeMap(**new** MyComparatorZ());

t.put("XXX", 100);

t.put("LLL", 200);

t.put("ZZZ", 300);

t.put("MMM", 400);

t.put("AAA", 500);

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

}

}

**class** MyComparatorZ **implements** Comparator

{

**public** **int** compare(Object o1,Object o2)

{

String s1 = o1.toString();

String s2 = o2.toString();

**return** s2.compareTo(s1);

}

} {ZZZ=300, XXX=100, MMM=400, LLL=200, AAA=500}

**HashTable**

* Underlying data structure is Hash table
* Insertion order not preserved and it is based on Hash code of keys
* Duplicate keys are not allowed but values are allowed
* Null is not applicable
* Implements Serializable, Clonable but not random access
* Every method present in HashTable is Synchronized hence HashTable is thread safe
* HashTable is best choice if our frequent operation is search operation

Constructors

HashTable hs = new HashTable()- creates empty HashTable with default initial capacity is 11 and default fill ration 0.75

HashTable hs = new HashTable(int initial capacity) - creates empty HashTable with specified capacity and default fill ration 0.75

HashTable hs = new HashTable(int initial capacity,float fillration) - creates empty HashTable with specified capacity and specified fill ration 0.75

HashTable hs = new HashTable(Map m)

HashCode representation:

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** HashTableDemo1 {

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

Hashtable h = **new** Hashtable();

h.put(**new** Temp1(5),"A");

h.put(**new** Temp1(2),"B");

h.put(**new** Temp1(6),"C");

h.put(**new** Temp1(15),"D");

h.put(**new** Temp1(23),"E");

h.put(**new** Temp1(16),"F");

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

}

}

**class** Temp1

{

**int** i;

Temp1(**int** i){

**this**.i=i;

}

**public** **int** hashCode()

{

**return** i;

}

**public** String toString()

{

**return** i+"";

}

}

O/p: {6=C, 16=F, 5=A, 15=D, 2=B, 23=E}

**Properties**:

In our program if anything which changes frequently(like username-password, mail-id..) are not recommended to hardcode in java program because if there is any change to reflect that change recompilation, rebuild and redeploy application are required even sometimes server restart also required which creates a big business impact to client

We can overcome this problem by using properties file such type of variable things we have to configure in properties file. From that properties file we have to read into java program and we can use those properties. The main advantage of this approach is if there is change in properties file to reflect that change just redeployment is enough which won’t create any business impact to client. We can use java properties object to hold properties which are coming from properties file

In normal map like (HashMap,hashTable,TreeMap) key-value can be any type but in the case of Properties key-value should be String type.

Constructors

Properties p = new Properties()

Methods

String setProperty(String pname,String pvalue) – to set new property

String getProperty(String pname) – to get value associated with specified property

Enumeration propertyNames(): returns all property names present in object

Void load(InputStream is) – To load properties from file into java properties object

Void store(OutputStream os, String comment) – To store properties from java properties object into properties file

Example

**package** com.durgasoft.Map;

**import** java.util.\*;

**import** java.io.\*;

**public** **class** PropertiesDemo {

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

Properties ps = **new** Properties();

FileInputStream fs = **new** FileInputStream("abc.properties");

ps.load(fs);

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

String s1 = ps.getProperty("Venky");

System.***out***.println(s1);

ps.setProperty("Nagu", "999");

FileOutputStream fo = **new** FileOutputStream("abc.properties");

ps.store(fo, "Updated by Vaibhav Demo Program");

}

}

**Queue**(I) (1.5 version enh)

It is child interface of Collection(I). If we want to represent a group of individual objects, prior to processing then we should got for Queue. For Example before sending SMS message, all mobile numbers we have to store in some data structure, in which order we added mobile numbers in the same order only Message should be delivered. For this first in first out requirement queue is the first choice.

Usulaly queue follows FIFO order but based on our requirement we can implement our own priority order also(PriorityQueue). From 1.5 version onwards LinkedList class also implements Queue interface. LinkedList based implementation of Queue always follows FIFO order.

Methods:

Boolean offer(Object o): to add an object into queue

Object peek() : To return head element of the queue, if queue empty returns null

Object element():To return head element of the queue, if queue empty gives NoSuchElementException

Object poll(): To remove & return head element of queue, if queue is empty returns null

Object remove():To remove & return head element of queue, if queue is empty gives NoSuchElementException

**PriorityQueue**

* If we want to represent a group of individual object prior to processing according to some priority then we should go for PriorityQueue
* Priority can be either default natural sorting order or customized sorting order defined by comparator
* Insertion order is not preserved and it is based on some priority.
* Duplicates objects are not allowed
* If we are depending on default natural sorting order compulsory object should be Homogeneous and comparable otherwise we will get run time exception ClassCastException
* If we are defining our own sorting by comparator then objects need not be Homogeneous and comparable
* NULL is not allowed

Constructors:

PriorityQueue pq = new PriorityQueue(): Creates an empty priority queue with default initial capacity as 11 and all objects will be inserted according to default natural sorting order.

PriorityQueue pq = new PriorityQueue(int initialcap): Creates an empty priority queue with specified capacity

PriorityQueue pq = new PriorityQueue(int initialcal, Comparator c): Creates an empty priority queue with specified capacity and

PriorityQueue pq = new PriorityQueue(SortedSet s )

PriorityQueue pq = new PriorityQueue(Collection c)

Example:

**package** com.durgasoft.Queue;

**import** java.util.\*;

**public** **class** PriorityQueueDemo1 {

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

PriorityQueue pq = **new** PriorityQueue();

System.***out***.println(pq.peek());

//System.out.print(pq.element());

**for**(**int** i = 0; i< 10; i++)

{

pq.offer(i);

}

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

System.***out***.println(pq.poll());

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

}

}

O/p

null

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

0

[1, 3, 2, 7, 4, 5, 6, 9, 8]

Example 2

**package** com.durgasoft.Queue;

**import** java.util.\*;

**public** **class** PriorityQueueDemo2 {

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

PriorityQueue pq = **new** PriorityQueue(15,**new** MyComparator1());

pq.offer("A");

pq.offer("Z");

pq.offer("L");

pq.offer("M");

pq.offer("X");

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

}

}

**class** MyComparator1 **implements** Comparator

{

**public** **int** compare(Object o1,Object o2)

{

String s1 = o1.toString();

String s2 = o2.toString();

**return** s2.compareTo(s1);

}

}

Blocking Queue:

### Producer-Consumer Example in Java using BlockingQueue

Here is our sample Java program to solve the classical producer consumer problem using BlockingQueue in Java:

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

/\*\*

\* Producer Consumer Problem solution using BlockingQueue in Java.

\* BlockingQueue not only provide a data structure to store data

\* but also gives you flow control, require for inter thread communication.

\*

\* @author Javin Paul

\*/

public class ProducerConsumerSolution {

public static void main(String[] args) {

BlockingQueue<Integer> sharedQ = new LinkedBlockingQueue<Integer>();

Producer p = new Producer(sharedQ);

Consumer c = new Consumer(sharedQ);

p.start();

c.start();

}

}

class Producer extends *Thread* {

private BlockingQueue<Integer> sharedQueue;

public Producer(BlockingQueue<Integer> aQueue) {

super("PRODUCER");

this.sharedQueue = aQueue;

}

public void run() {

// no synchronization needed

for (int i = 0; i < 10; i++) {

try {

System.out.println(getName() + " produced " + i);

sharedQueue.put(i);

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer extends *Thread* {

private BlockingQueue<Integer> sharedQueue;

public Consumer(BlockingQueue<Integer> aQueue) {

super("CONSUMER");

this.sharedQueue = aQueue;

}

public void run() {

try {

while (true) {

Integer item = sharedQueue.take();

System.out.println(getName() + " consumed " + item);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Output

PRODUCER produced 0

CONSUMER consumed 0

PRODUCER produced 1

CONSUMER consumed 1

PRODUCER produced 2

CONSUMER consumed 2

PRODUCER produced 3

CONSUMER consumed 3

PRODUCER produced 4

CONSUMER consumed 4

PRODUCER produced 5

CONSUMER consumed 5

PRODUCER produced 6

CONSUMER consumed 6

PRODUCER produced 7

CONSUMER consumed 7

PRODUCER produced 8

CONSUMER consumed 8

PRODUCER produced 9

CONSUMER consumed 9

**Explanation of code**  
If you look at above code example, you will find that we have created and started two threads and named them Producer and Consumer. The Producer thread executes the code inside [run() method](http://java67.blogspot.com/2015/12/difference-between-thread-start-and-run-method-java.html), which adds 10 Integer object starting from 0.  
  
After adding each element, the Producer thread is sleeping for 200 milliseconds by calling [Thread.sleep()](http://java67.blogspot.com/2012/08/difference-between-yield-and-sleep-in.html) method. This gives time to the Consumer thread to consume elements from Queue, that's why our code never blocks.  
  
You can see that our Producer and Consumer threads are working in sync because of Thread.sleep() we have introduced after put() call. You can further experiment with the code by removing the code to [pause the Producer thread](http://java67.blogspot.com/2015/06/how-to-pause-thread-in-java-using-sleep.html) or inserting pause on Consumer thread to  create scenarios where Queue is full or empty.  
  
  
**Benefits of using BlockingQueue to solve Producer Consumer**

* Simple code, much more readable
* less error prone as you don't have to deal with any external synchronization

version 1.6 enhancements in Collection framework

**NavigableSet(I)**

* Child interface of SortedSet
* It defines several methods for navigation purposes.

Methods:

Floor(e) : return highest element which is <=e

Lower(e): return highest element which is <e

Ceiling(e): return lowest element which is >=e

Higher(e): return lowest element which is >e

pollFirst():remove and return first element

pollLast():remove and return last element

descendingSet() : return navigable set in reverse order

Example

**package** com.durgasoft.Collection.Set;

**import** java.util.\*;

**public** **class** NavigableSetDemo1 {

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

TreeSet<Integer> t = **new** TreeSet<Integer>();

t.add(1000);

t.add(2000);

t.add(3000);

t.add(4000);

t.add(5000);

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

System.***out***.println(t.ceiling(2000));

System.***out***.println(t.higher(2000));

System.***out***.println(t.floor(3000));

System.***out***.println(t.lower(3000));

System.***out***.println(t.descendingSet());

System.***out***.println(t.pollFirst());

System.***out***.println(t.pollLast());

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

}

}

**NavigableMap(I)**

* Child interface of SortedMap
* It defines several methods for navigation purposes.

Methods

FloorKey(e) : return highest element which is <=e

LowerKey(e): return highest element which is <e

CeilingKey(e): return lowest element which is >=e

HigherKey(e): return lowest element which is >e

pollFirstEntry():remove and return first entry

pollLastEntry():remove and return last entry

descendingMap() : return navigable map in reverse order

Example

**package** com.durgasoft.Map;

**import** java.util.\*;

**public** **class** NavigableMapDemo1 {

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

TreeMap<String,String> t = **new** TreeMap<String,String>();

t.put("B", "Banana");

t.put("C", "Cat");

t.put("Z", "Zebra");

t.put("M", "Mango");

t.put("V", "Van");

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

System.***out***.println(t.ceilingKey("C"));

System.***out***.println(t.higherKey("C"));

System.***out***.println(t.floorKey("Z"));

System.***out***.println(t.lowerKey("Z"));

System.***out***.println(t.descendingMap());

System.***out***.println(t.pollFirstEntry());

System.***out***.println(t.pollLastEntry());

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

}

}

{B=Banana, C=Cat, M=Mango, V=Van, Z=Zebra}

C

M

Z

V

{Z=Zebra, V=Van, M=Mango, C=Cat, B=Banana}

B=Banana

Z=Zebra

{C=Cat, M=Mango, V=Van}

Utility Classes

**Collections**

Collections class defines several utility methods for Collection objects like Sorting, Searching, Reversing etc.

Sorting Elements of List

2 sort methods

Public static void sort(List l) : To sort based on default natural sorting order, in this case List should contain homogeneous and comparable objects otherwise we will get ClassCastException. Also List should not contain NULL, otherwise NullPointerException

Public static void sort(List l, Comparator c): To sort based on customized sorting order.

Example – (Default Sorting)

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** CollectionsSortDemo1 {

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

ArrayList l = **new** ArrayList();

l.add("Z");

l.add("B");

l.add("S");

l.add("U");

l.add("F");

System.***out***.println("Before Sorting");

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

Collections.*sort*(l);

System.***out***.println("After Sorting");

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

}

}

Before Sorting

[Z, B, S, U, F]

After Sorting

[B, F, S, U, Z]

Example – 2(Customized sorting)

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** CollectionsSortDemo2 {

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

ArrayList l = **new** ArrayList();

l.add("Z");

l.add("B");

l.add("S");

l.add("U");

l.add("F");

System.***out***.println("Before Sorting");

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

Collections.*sort*(l,**new** MycomparatorC());

System.***out***.println("After Sorting");

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

}

}

**class** MycomparatorC **implements** Comparator

{

**public** **int** compare(Object o1, Object o2)

{

String s1 = o1.toString();

String s2 = o2.toString();

**return** s2.compareTo(s1);

}

}

Searching Elements of List

Collection class defines the following binary search methods

Public static int binarySearch(List l, Object target): if the list is sorted according to default natural sorting order then we have to use this method

Public static int binarySearch(List l, Object target, Comparator c): We have to use this method if the list is sorted according to customized sorting order.

Conclusions:

* The above search methods internally will use Binary Search algorithm.
* Successful search returns index, unsuccessful search returns insertion point.
* Insertion point is the location where we can place target element in the sorted list
* Before calling binary search method compulsory list should be sorted otherwise we will get unpredictable results
* If the list is sorted according to comparator then at the time of search operation also we have to pass same comparator object otherwise we will get unpredictable results.

Example -1(Default natural sorting)

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** CollectionsSearchDemo1 {

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

ArrayList l = **new** ArrayList();

l.add("Z");

l.add("A");

l.add("M");

l.add("S");

l.add("c");

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

Collections.*sort*(l);

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

System.***out***.println(Collections.*binarySearch*(l, "Z"));

System.***out***.println(Collections.*binarySearch*(l, "J"));

}

}

[Z, A, M, S, c]

[A, M, S, Z, c]

3

-2

Example -1(Customized sorting)

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** CollectionsSearchDemo2 {

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

ArrayList l = **new** ArrayList();

l.add(15);

l.add(0);

l.add(20);

l.add(10);

l.add(5);

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

Collections.*sort*(l, **new** MyComparatorS());

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

System.***out***.println(Collections.*binarySearch*(l,10,**new** MyComparatorS() ));

System.***out***.println(Collections.*binarySearch*(l,13,**new** MyComparatorS() ));

System.***out***.println(Collections.*binarySearch*(l, 17));

}

}

**class** MyComparatorS **implements** Comparator

{

**public** **int** compare(Object o1, Object o2)

{

Integer i1 = (Integer) o1;

Integer i2 = (Integer) o2;

**return** i2.compareTo(i1);

}

}

[15, 0, 20, 10, 5]

[20, 15, 10, 5, 0]

2

-3

-6

Note: For the list of n elements in the Binary search method

1.Successful search result range : 0 to (n-1)

2.Unsuccessful search result range : -(n+1) to -1

3.Total Result range: -(n+1) to n-1

Reversing Elements of List

Collections class defines following reverse method to reverse elements of List

Public static void reverse(List l)

Example

**package** com.durgasoft.Collection.List;

**import** java.util.\*;

**public** **class** CollectionsReverseDemo {

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

ArrayList l = **new** ArrayList();

l.add(15);

l.add(0);

l.add(20);

l.add(10);

l.add(5);

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

Collections.*reverse*(l);

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

}

}

[15, 0, 20, 10, 5]

[5, 10, 20, 0, 15]

reverse Vs reverseOrder methods

We can use reverse() method to reverse order of elements of list where as we can use reverseOrder() method to get reversed comparator.

Comparator c1 = Collections.reverseOrder(Comparator c)

**Arrays Class**

Arrays class is an utility class to define several utility methods for Array objects.

Sorting elements of Arrays:

Arrays class defines the following sort methods to sort elements of primitive and object type arrays.

Public static void sort(Premitive[] p): to sort according to natural sorting order

Public static void sort(Object[] o): to sort according to natural sorting order

Public static void sort(Object[] o,Comparator c): to sort according to customized sorting order

Example

**package** com.durgasoft.Collections.Arrays;

**import** java.util.\*;

**public** **class** ArraySortDemo1 {

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

**int** [] a = {10,23,5,14,45,17};

System.***out***.println("Premitive Array before sorting");

**for**(**int** a1:a)

{

System.***out***.print(a1+" ");

}

Arrays.*sort*(a);

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

System.***out***.println("Premitive Array After sorting");

**for**(**int** a1:a)

{

System.***out***.print(a1+" ");

}

String [] S = {"Ashoka","Raja","Chitra","Manoj"};

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

System.***out***.println("Object Array before sorting");

**for**(String a1:S)

{

System.***out***.print(a1+" ");

}

Arrays.*sort*(S);

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

System.***out***.println("Object Array After sorting");

**for**(String a1:S)

{

System.***out***.print(a1+" ");

}

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

System.***out***.println("Object Array Customized sorting");

Arrays.*sort*(S, **new** MyComparator1());

**for**(String a1:S)

{

System.***out***.print(a1+" ");

}

}

}

**class** MyComparator1 **implements** Comparator

{

**public** **int** compare(Object o1,Object o2)

{

String s1 = o1.toString();

String s2 = o2.toString();

**return** s2.compareTo(s1);

}

}

Premitive Array before sorting

10 23 5 14 45 17

Premitive Array After sorting

5 10 14 17 23 45

Object Array before sorting

Ashoka Raja Chitra Manoj

Object Array After sorting

Ashoka Chitra Manoj Raja

Object Array Customized sorting

Raja Manoj Chitra Ashoka

Searching elements of Arrays

Arrays class defines the following binary search methods.

Public static int binarySearch(Premitive[] p, Primitive target)

Public static int binarySearch(Object[] 0, Object target)

Public static int binarySearch(Object[] 0, Object target,Comparator c)

Example

**package** com.durgasoft.Collections.Arrays;

**import** java.util.\*;

**public** **class** ArraySearchDemo1 {

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

**int** [] a = {10,5,20,11,6};

Arrays.*sort*(a);

System.***out***.println(Arrays.*binarySearch*(a, 6));

System.***out***.println(Arrays.*binarySearch*(a, 14));

String[] S = {"A","Z","B"};

Arrays.*sort*(S);

System.***out***.println(Arrays.*binarySearch*(S, "Z"));

System.***out***.println(Arrays.*binarySearch*(S, "S"));

Arrays.*sort*(S, **new** MyComparator1());

System.***out***.println(Arrays.*binarySearch*(S, "Z",**new** MyComparator1()));

System.***out***.println(Arrays.*binarySearch*(S, "S",**new** MyComparator1()));

}

}

1

-5

2

-3

0

-2

Conversion of Array to List

Method

Public static List asList(Object[] a)

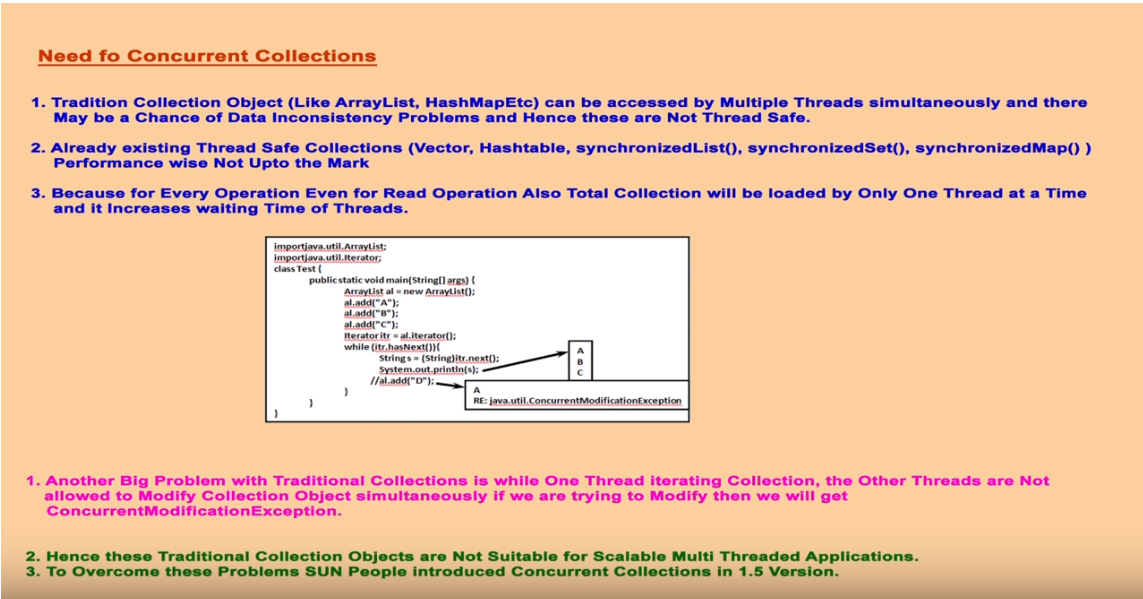
Ex:

String [] s = {“Z”,”A”,”B”};

List l = Arrays.asList(s);

Note: By using List reference we can’t perform any operation which changes the size or else we will get UnsupportedOpretionException

**Concurrent Collection**



ConcurrentModificationException Example:

**package** Example;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** MyThread **extends** Thread{

**static** ArrayList<String> *l* = **new** ArrayList<String>();

@Override

**public** **void** run() {

**try**

{

Thread.*sleep*(2000);

}

**catch**(Exception e) {}

System.***out***.println("Child Thread updating List");

*l*.add("D");

}

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

*l*.add("A");

*l*.add("B");

*l*.add("C");

MyThread t = **new** MyThread();

t.start();

Iterator itr = *l*.iterator();

**while**(itr.hasNext())

{

String s1 = (String) itr.next();

System.***out***.println("Main Thread Iterating List and Current object is:"+s1);

Thread.*sleep*(3000);

}

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

}

}

O/P

Main Thread Iterating List and Current object is:A

Child Thread updating List

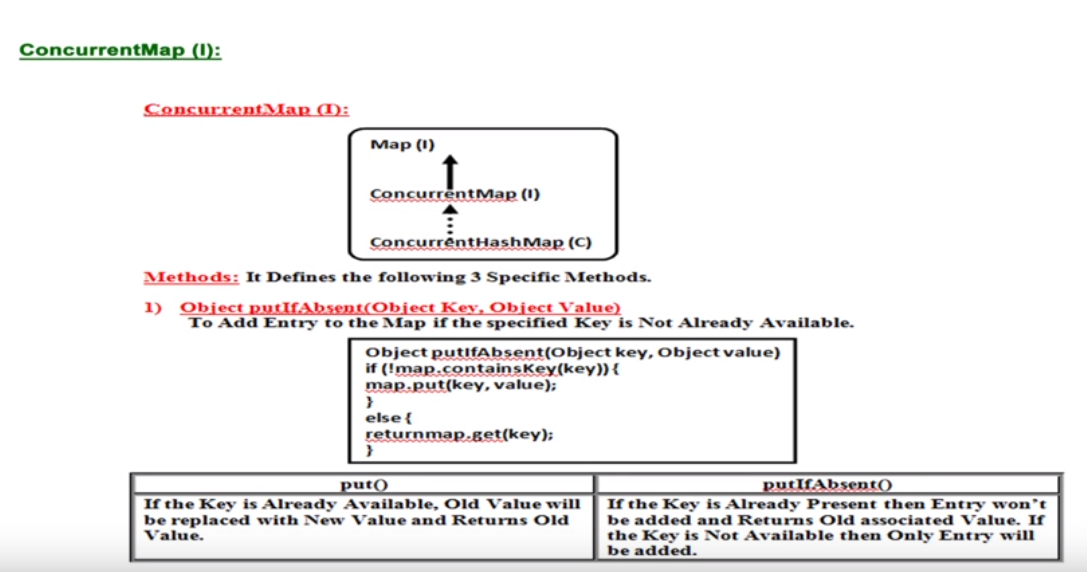
Exception in thread "main" java.util.ConcurrentModificationException

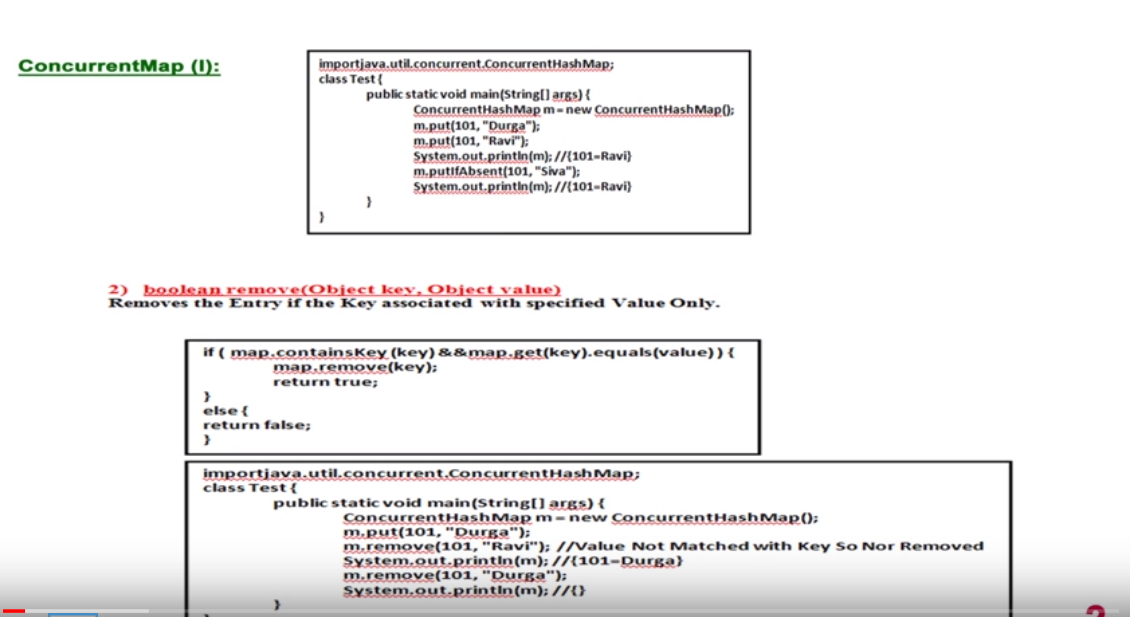
at java.util.ArrayList$Itr.checkForComodification(Unknown Source)

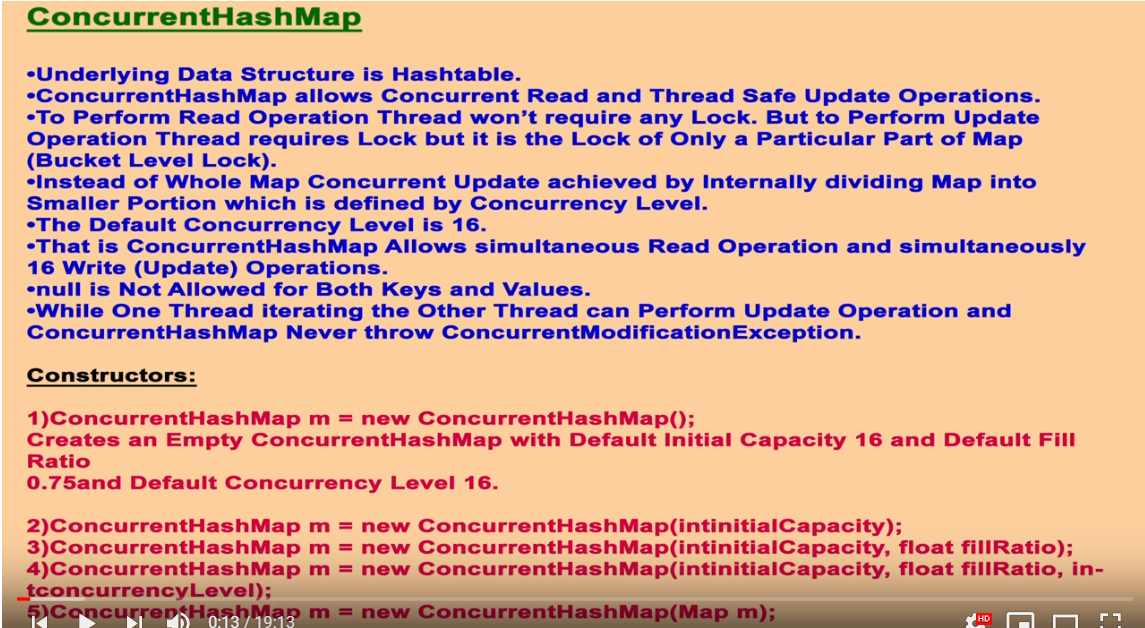
at java.util.ArrayList$Itr.next(Unknown Source)

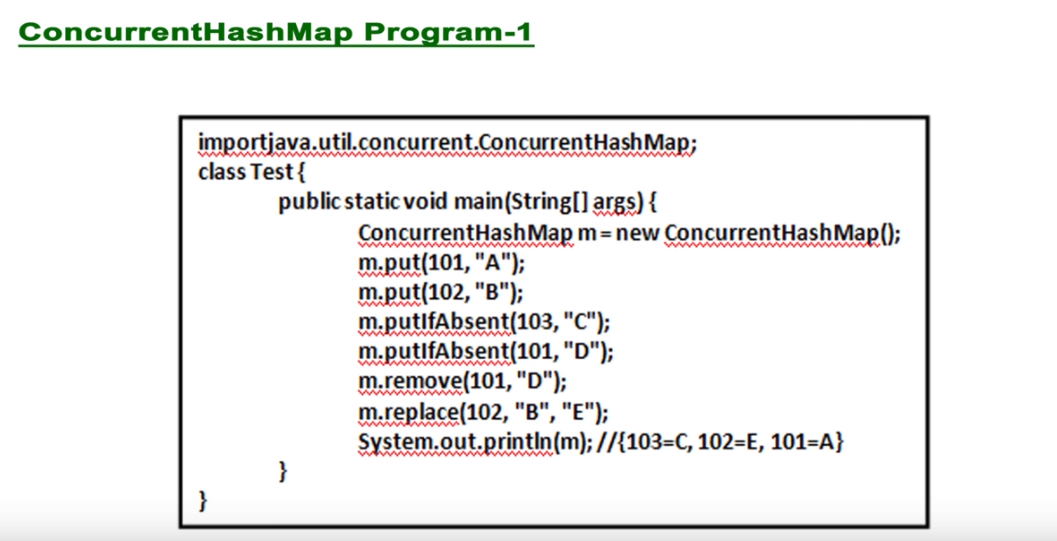
at Example.MyThread.main(MyThread.java:31)

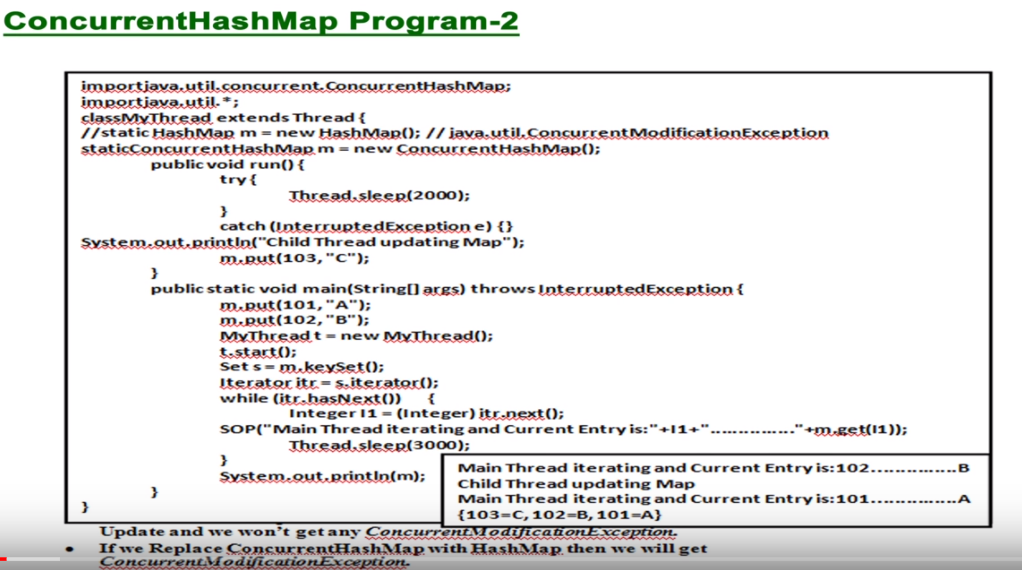


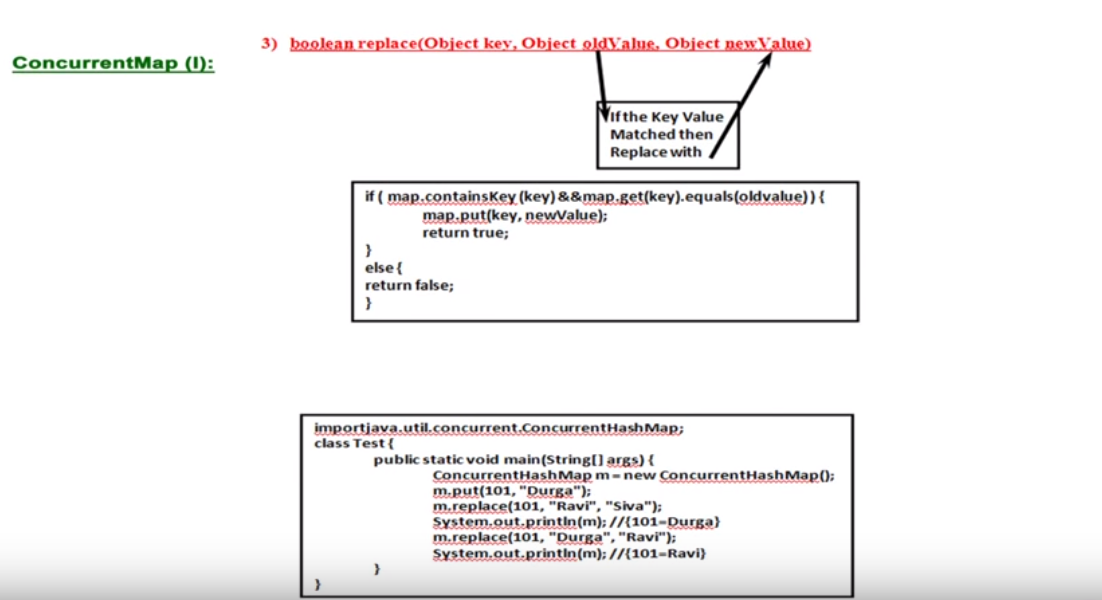


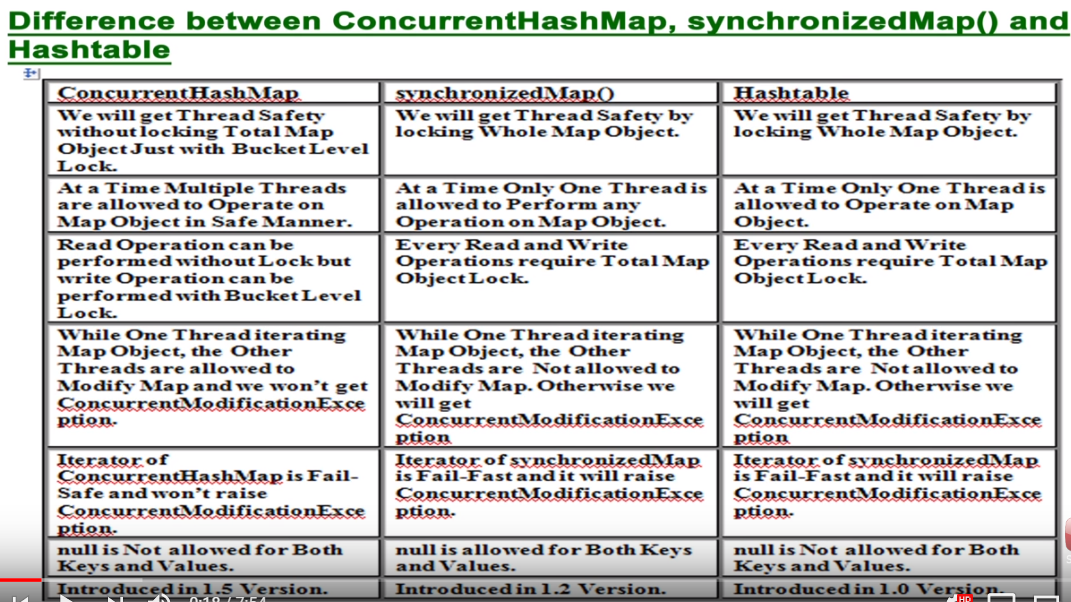


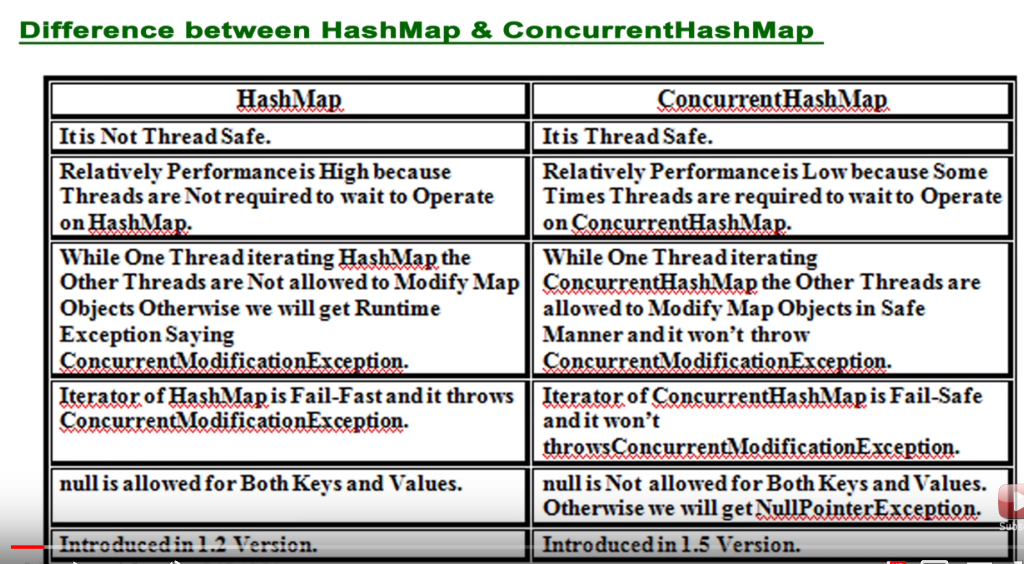


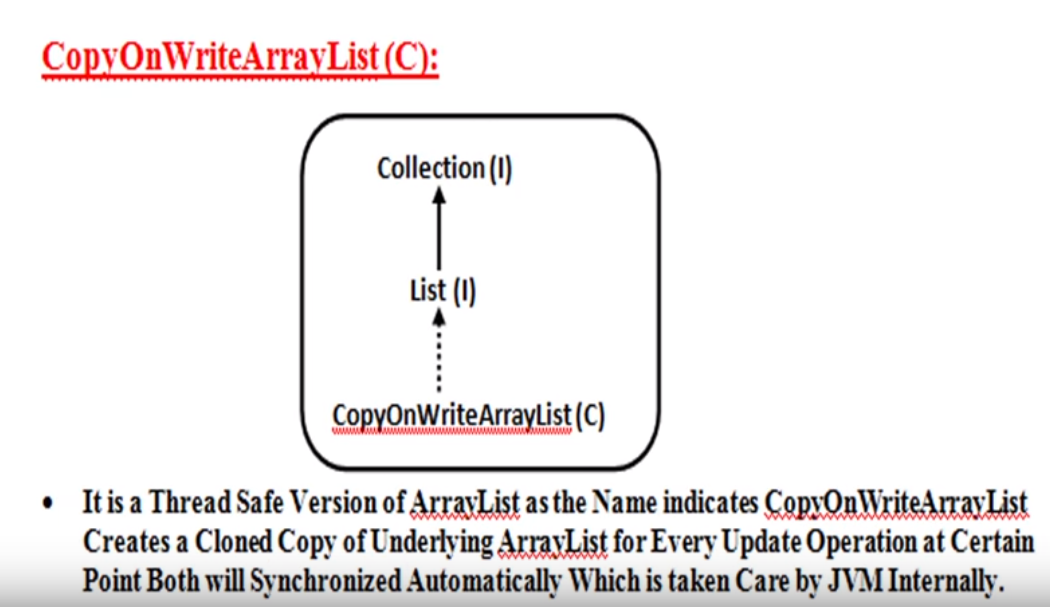


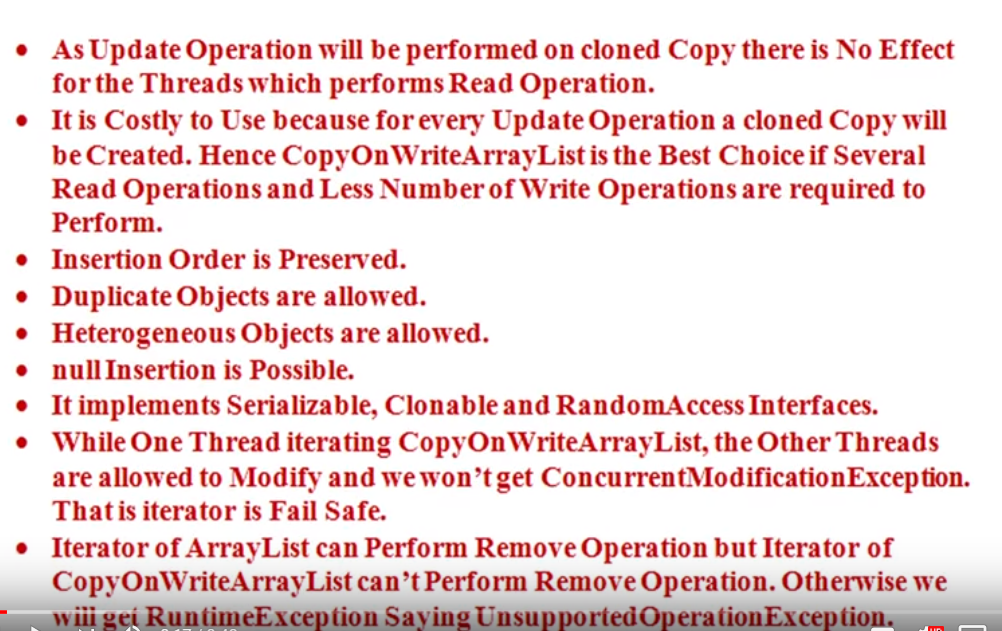












**Equals and Hashcode methods in JAVA and Impact on Collections:**

Importance of overriding equals () and hashcode() method provided by the Object class when using with collections. Also these 2 methods have direct connection with performance of collection.

What is equals () method:

It helps to check if 2 objects are equal or not with respect to content. However the implementation present on the Object class only checks if 2 references refer to the same object or not(which is called Shallow comparison same as ==). This is the default implementation every instance obtains from superclass. So to get proper check of object content(i.e Deep comparison) we need to override equals methods.

This is what the JDK 1.4 API documentation says about the equals method of Object class-

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation:

* It is **reflexive**: for any reference value x, x.equals(x) should return true.
* It is **symmetric**: for any 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 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 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 object is modified.
* For any non-null reference value x, x.equals(null) should return false.

What is hashcode() method:

Which returns an integer, object’s hashcode() method returns the memory location converted into integer. So hashcode() method helps to identify an object in heap of other objects, this is helpful in maintaining the elements in the collections so that collections can easily look for an object as fast as possible.

This method returns the hash code value for the object on which this method is invoked. This method returns the hash code value as an integer and is supported for the benefit of hashing based collection classes such as Hashtable, HashMap, HashSet etc. This method must be overridden in every class that overrides the equals method.

Student s1 = new Student(“H1234”);

Student s2 = new Student(“H1234”);

S1.equals(s2) – returns FALSE.

Here s1 refers to one instance and s2 refers to another so we will get FALSE. Let’s see how this impacts in collection like Map.

Student.java

**package** Core;

**public** **class** Student {

**private** String regNumber;

**public** Student(String reg) {

**this**.regNumber=reg;

}

**public** **void** setRegNumber(String regNumber) {

**this**.regNumber = regNumber;

}

**public** String getRegNumber() {

**return** regNumber;

}

}

ReportCard.java

**package** Core;

**public** **class** ReportCard {

**private** **int** MathScore;

**public** ReportCard() {

**this**.MathScore = (**int**) (Math.*random*()\*101+1);

}

**public** **int** getMathScore() {

**return** MathScore;

}

**public** **void** setMathScore(**int** mathScore) {

MathScore = mathScore;

}

}

Test.java

**package** Core;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Test {

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

Student s1 = **new** Student("H1234");

Student s2 = **new** Student("H1234");

System.***out***.println(s1.equals(s2));

Map<Student,ReportCard> m1 = **new** HashMap<Student,ReportCard>();

m1.put(s1, **new** ReportCard());

m1.put(s2, **new** ReportCard());

System.***out***.println(m1.size());

}

}

O/P:

false

2

i.e : 2 Students with same key got added into map. So now we will override the equals() of Object class with our own implementation. And need to implement hashCode() method as well.

Student.java

**package** Core;

**public** **class** Student {

**private** String regNumber;

**public** Student(String reg) {

**this**.regNumber=reg;

}

**public** **void** setRegNumber(String regNumber) {

**this**.regNumber = regNumber;

}

**public** String getRegNumber() {

**return** regNumber;

}

**public** **boolean** equals(Object o){

**if**(o!=**null** && o **instanceof** Student)

{

String regNum = ((Student)o).getRegNumber();

**if**(regNum!=**null** && regNum.equals(**this**.getRegNumber()))

{

**return** **true**;

}

}

**return** **false**;

}

**public** **int** hashCode()

{

**return** **this**.getRegNumber().hashCode();

}

}

o/p:

true

1

Some Rules:

1. Both hashcode() and equals() needs to be overridden at the same time. Because if you just override hashcode() then equals may be still comparing based on hashcode returned by OriginalHashcode() method
2. Also overridding these methods are necessary in case of hashing comparisons as you expect to compare based on values that object holds and use object as keys for hashing. Otherwise values will be used as keys for hashing and OriginalHashcode() is used for comparison.
3. General contract states that equal objects should have equal hashcodes but opposite is not true

After reviewing the general contracts of these two methods, it is clear that the relationship between these two methods can be summed up in the following statement -

**Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.**

**Summary & Miscellaneous Tips**

* Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.
* The equals method provides "deep comparison" by checking if two objects are logically equal as opposed to the "shallow comparison" provided by the equality operator ==.
* However, the equals method in java.lang.Object class only provides "shallow comparison", same as provided by the equality operator ==.
* The equals method only takes Java objects as an argument, and not primitives; passing primitives will result in a compile time error.
* Passing objects of different types to the equals method will never result in a compile time error or runtime error.
* For standard Java wrapper classes and for java.lang.String, if the equals argument type (class) is different from the type of the object on which the equals method is invoked, it will return false.
* The class java.lang.StringBuffer does not override the equals method, and hence it inherits the implementation from java.lang.Object class.
* The equals method must not provide equality comparison with any built in Java class, as it would result in the violation of the symmetry requirement stated in the general contract of the equals method.
* If null is passed as an argument to the equals method, it will return false.
* Equal hash codes do not imply that the objects are equal.
* return 1; is a legal implementation of the hashCode method, however it is a very bad implementation. It is legal because it ensures that equal objects will have equal hash codes, it also ensures that the hash code returned will be consistent for multiple invocations during the same execution. Thus, it does not violate the general contract of the hashCode method. It is a bad implementation because it returns *same* hash code for all the objects. This explanation applies to all implementations of the hashCode method which return same constant integer value for all the objects.
* In standard JDK 1.4, the wrapper classes java.lang.Short, java.lang.Byte, java.lang.Character and java.lang.Integer simply return the value they represent as the hash code by typecasting it to an int.
* Since JDK version 1.3, the class java.lang.String caches its hash code, i.e. it calculates the hash code only once and stores it in an instance variable and returns this value whenever the hashCode method is called. It is legal because java.lang.String represents an immutable string.
* It is incorrect to involve a random number directly while computing the hash code of the class object, as it would not consistently return the same hash code for multiple invocations during the same execution.

How to Create Immutable class in Java?

An [immutable class](https://en.wikipedia.org/wiki/Immutable_object) is one whose state can not be changed once created. There are certain guidelines to **create a class immutable in Java**.

* Don’t provide “setter” methods — methods that modify fields or objects referred to by fields.
* Make all fields final and private
* Don’t allow subclasses to override methods
* Special attention when having mutable instance variables

Lets first identify **advantages of immutable class**. In Java, immutable classes are:

1. are simple to construct, test, and use
2. are automatically thread-safe and have no synchronization issues
3. do not need a copy constructor
4. do not need an implementation of clone
5. allow [hashCode()](https://howtodoinjava.com/java/related-concepts/working-with-hashcode-and-equals-methods-in-java/) to use lazy initialization, and to cache its return value
6. do not need to be copied defensively when used as a field
7. make good [Map keys and Set elements](https://howtodoinjava.com/java/collections/how-hashmap-works-in-java/) (these objects must not change state while in the collection)
8. have their class invariant established once upon construction, and it never needs to be checked again
9. always have “**failure atomicity**” (a term used by Joshua Bloch) : if an immutable object throws an exception, it’s never left in an undesirable or indeterminate state

Collections Interview questions:

<https://www.journaldev.com/1330/java-collections-interview-questions-and-answers>

<https://www.baeldung.com/java-collections-interview-questions>

<https://www.journaldev.com/552/java-tricky-interview-questions>

%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Generics**

The main objectives of generics are – To provide type-safety and To resolve Type-casting problems.

Type Safety:

* Arrays are type safe, i.e. We can give the guarantee for the type of elements present inside array. For example if our program requirement is to hold only string types of objects and if we can choose String array , by mistake if we are trying to add any other type of objects we will get compile time error.
* Collections are not type safe, i.e. we can’t give the guarantee for the type of elements present inside array. For example if our program requirement is to hold only string types of objects and if we can choose ArrayList , by mistake if we are trying to add any other type of objects we won’t get compile time error but program may fail at run time.

Type Casting:

* In the case of Arrays at the time of retrieval it is not required to perform type casting because there is a guarantee for the type of elements present inside Array
* In the case of Collection at the time of retrieval, compulsory we should perform type casting because there is a not guarantee for the type of elements present inside collection.

To overcome above problems of collections we should go for Generics concept. Introduced in 1.5 version.

For Example to hold only String type of objects we can create generics version of ArrayList object as below

ArrayList<String> l = new ArrayList <String>()

For this ArrayList we can add only string type of objects by mistake if we are trying to add any other type we will get compile time error. So we are getting Type safety through generics.

At the time of retrieval we are no required to perform type-casting.

*Conclusion 1*: Polymorphism concept applicable only for the base type but not for parameter type.( Usage of parent reference to hold child object is Polymorphism)

Ex

ArrayList <String> l = new ArrayList<String>() – Valid

List <String> l = new ArrayList<String>() – Valid

Collection <String> l = new ArrayList<String>() – Valid

ArrayList <Object> l = new ArrayList<String>() – Invalid ; Polymorphism only applicable for Base type but not for Parameter type.

*Conclusion 2*:For the Type Parameter we can provide any class or interface name but not primitives else we will get CTE.

ArrayList <int> l = new ArrayList<int>() – Invalid

**Generics classes**

Until 1.4 version a non-generic version of ArrayList class is declared as follows

Class ArrayList

{

Add(Object o)

Object get(int index)

}

The argument to add method is Object hence we can add any type of object to ArrayList due to this we are missing type safety.

Return type of get method is Object hence at the time of retrieval we have to perform type casting.

But in 1.5 version, a generics version of ArrayList declared as follows

Class ArrayList <T> // T is type parameter

{

Add(T t)

T get(int index)

}

So based on our runtime requirement, T will be replaced with our provided type. For ex to hold only String type of objects a generic version will be as below

ArrayList<String> l = new ArrayList<String>();

So when we are adding element to ArrayList we can add only String objects so we are getting type safety also while retrieving elements we are not required to perform type casting.

In generics we are associating a type parameter to the class. Such type of parameterized classes are nothing but generics classes or template classes. Based on our requirement we can define our own generics classes also.

Ex:

Class Account<T>

{ }

Account <Gold> a1 = new Account<Gold>();

Account <Platinum> a1 = new Account< Platinum >();

Example

**package** com.durgasoft.Generics;

**class** Gen<T>

{

T ob;

Gen(T ob)

{

**this**.ob = ob;

}

**public** **void** show()

{

System.***out***.println("The type of ob "+ ob.getClass().getName());

}

**public** T getOb()

{

**return** ob;

}

}

**public** **class** GenDemo1 {

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

Gen <String> g1 = **new** Gen<String>("Durga");

g1.show();

System.***out***.println(g1.getOb());

Gen <Integer> g2 = **new** Gen<Integer>(10);

g2.show();

System.***out***.println(g2.getOb());

Gen <Double> g3 = **new** Gen<Double>(10.5);

g3.show();

System.***out***.println(g3.getOb());

}

}

The type of ob java.lang.String

Durga

The type of ob java.lang.Integer

10

The type of ob java.lang.Double

10.5

**Bounded Types:**

We can bound the type parameter for a particular range by using extends keyword such types are called Bounded Types.

Ex:

Class Test<T>

{ }

As the type parameter we can pass any type and there are no restrictions and hence it is unbounded type.

Test <Integer> t1 = new Test<Integer>(); //Valid

Test <String> t2 = new Test<String>(); // Valid

Syntax for Bounded type:

Class Test<T extends X>

{

}

X can be either class or Interface. If X is a class then as the type parameter we can pass either X type or it’s child classes.

If X is an interface then as the type parameter we can pass either X type or it’s implementation classes.

EX:1

Class Test<T extends Number>

{

P S V main(String args[])

{ Test<Integer> t1 = new Test<Integer>(); //Valid

Test<String> t2 = new Test<String>()// Invalid CE- Type parameter java.lang.String is not within it’s bound

}

}

EX:2

Class Test<T extends Runnable>

{

P S V main(String args[])

{ Test<Runnable> t1 = new Test< Runnable >(); //Valid

Test<Thread> t2 = new Test<Thread >(); //Valid

Test<Integer> t2 = new Test<Integer>()// Invalid CE- Type parameter java.lang.Integer is not within it’s bound

}

}

We can define bounded types even in combination also.

Ex:

Class T <T extends Number & Runnable>

{ }

As the type parameter we can take anything which should be child class of number and should implements Runnable interface

Class Test<T extends Runnable & Comparable> // Valid

Class Test<T extends Number & Runnable & Comparable> // Valid

Class Test<T extends Runnable & Number> // Invalid because Class should first then interface

Class Test<T extends Number & Thread> // Invalid because can’t extends more than one classes.

Note:

1. We can define bounded types only by using extends keyword and we can’t use implements and super keywords; but we can replace implements keyword purpose with extends keyword.
2. As the type parameter <T> we can take any valid java identifier but it is convention to use T
3. Based on our requirement we can declare any number of type parameters and all these type parameters should be separated with comma(,). Like

Class Test<A,B,C> { }

**Generics methods and wild card character(?)**

1. M1(AL<String> l): We can call this method by passing ArrayList of only String type but within the method we can add only String type of objects to the list if we are trying to add any other type then we will get CTE.
2. M1(AL<?> l): We can call this method by passing ArrayList of any type but within the method we can’t add anything to the list except null because we don’t know the type exactly. This type of methods are best suitable for read only operation.
3. M1(AL<? extends X > l):X can be either class or Interface. If X is class then we can call this method by passing ArrayList of either X type or it’s child classes. If X is an interface then we can call this method by passing ArrayList of either X type or it’s implementation classes. But within the method we can’t add anything to the list except NULL because we don’t know the type exactly. This type of methods are best suitable for read only operation.
4. M1(AL<? super X) l): X can be either class or Interface. If X is class then we can call this method by passing ArrayList of either X type or it’s super classes. If X is an interface then we can call this method by passing ArrayList of either X type or super class of implementation class of X. but Within the method we can add X types of objects and NULL.

Valid/Invalid

1.ArrayList<String> l = new ArrayList<String>(); //Valid

2.ArrayList<?> l = new ArrayList<String>(); //Valid

3.ArrayList<?> l = new ArrayList<Integer>(); //Valid

4.ArrayList<? extends Number> l = new ArrayList<Integer>(); //Valid

5.ArrayList<? extends Number> l = new ArrayList<String>(); // CTE:Incompatible type found

6.ArrayList<? super String> l = new ArrayList<Object>(); //Valid

7.ArrayList<?> l = new ArrayList<?>(); //Invalid CTE: Unexpected type found

8.ArrayList<?> l = new ArrayList<? Extends Number> (); //Invalid CTE: Unexpected type found

We can declare type parameter either at class level or at method level.

Declaring Type parameter at class level.

Class Test<T>{

We can use T within this class based on our requirement

}

Declaring Type parameter at method level

We have to declare type parameter just before return type

Class Test

{

Public <T> void m1(T object)

{

We can use T anywhere within method based on our requirement

}

}

We can define bounded types even at method level also

Public <T> void m1();

Public <T extends Number> void m1();

Public <T extends Runnable> void m1();

Public < T extends Runnable & Comparable > void m1();

Public < T extends Number & Runnable & Comparable> void m1();

Public < T extends Runnable & Number> void m1();// Invalid because Class should first then interface

Public < T extends Number & Thread> void m1();// Invalid because can’t extends more than one classes.

Public <T> void m1();

**Communication with non-generic code:**

If we send generic object to non-generic area then it start behaving like non generic object similarly if we send non-generic object to generic area then it start behaving like generic object. i.e. the location in which object present based on that behavior will be defined.

Example

Class Test

{

P S V main(String [] args)

{

ArrayList<String> l = new ArrayList<String>(); // below Generic area

l.add(“Durga”); //Valid

l.add(“Vive”); //Valid

l.add(10.5); // CTE

m1(l);

sop(l);// [“Durga”,”Vive”,10,10.5,true]

l.add(10.5); //CTE

}

P S V m1(ArrayList l) // Below non-generic area

{

l.add(10);

l.add(10.5);

l.add(true);

}

}

The main objective of Generics is to provide type safety and to resolve type casting problems. Type safety and type casting both are applicable for compile time hence generics concept also applicable only at compile time but not at run time.

At the time of compilation as the last step generics syntax will be removed and hence for the JVM, generics syntax won’t be available hence following declarations are equal.

ArrayList l = new ArrayList<String>();

ArrayList l = new ArrayList<Integer>();

ArrayList l = new ArrayList<Double>();

Comparisions of different Java versions:

**JAVA 8 Features**

Concise code – Enabled functional programming. For this Java 8 came with Lambada expression.

Let’s take a example, where we write a program or function to find out square of number.

**package** Java8;

**public** **class** Example1

{

**public** **static** **int** squareIt(**int** n)

{

**return** n\*n;

}

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

System.***out***.println("Square of 4:"+*squareIt*(4));

System.***out***.println("Square of 5:"+*squareIt*(5));

}

}

Now let’s write this program with Lambada expression. We can replace function like below.

**Function<Integer,Integer> f = i-> i\*i;**

**package** Java8;

**import** java.util.function.Function;

**public** **class** Example1

{

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

Function<Integer,Integer> f = i-> i\*i;

System.***out***.println("Square of 4:"+f.apply(4));

System.***out***.println("Square of 5:"+f.apply(5));

}

}

So like this we can write concise code.

**Lambda Expressions**

The main objective of Lambda expression is to bring benefits of functional programming into java.

It is anonymous function without return type and without modifiers such type of functions are called anonymous function.

How to write Lambada expression:

**Public void m1()**

**{**

**System.out.prrintln(“Hello”);**

**}**

Now convert this function into Lambada expression:

**() -> System.out.println();**

Example 2:

Public void m1(int a,int b)

{

System.out.println(a+b)

}

(a,b) -> System.out.println(a+b)

Example 3:

Public int squareIt(int n)

{

return n\*n;

}

(int n) -> { return n\*n ;}

(int n) -> n\*n;

n->n\*n;

FI🡺Functional Interface

In an interface contains single abstract method called as Functional Interface, also can contain any number of default and static methods.

@FunctionalInterface

Interface interf

{

Public void m1();

Default void m2()

{

}

Public static void m3()

{

}

}

**Lambda expressions with functional interfaces.**

Example: Convert below code into Lambda expression.

Interface interf

{

Public void m1();

}

Class Demo implements interf

{

Public void m1()

{

System.out.println(“Hello…”);

}

}

Class Test

{

Public static void main(String args[])

{

Demo d1 = new Demo();

d1.m1();

}

}

--With Lambada exp—

Interface interf

{

Public void m1();

}

Class Test

{

Public static void main(String args[])

{

Interf i = () -> System.out.println(“Hello…”);

i.m1();

}

}

So here functional interface’s provides reference to the Lambda expression.