# **Java OOPs Concept**

**OOPs**: Object Oriented Programming System: **Object** means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.

**Object**: Any entity that has **state and behaviour** is known as an object. For example: chair, pen, table etc. It can be **physical and logical**.

Class: **Collection of objects** is called class. It is a logical entity. It is a template or blueprint from which objects are created.

* **Abstraction**: **Hiding internal details and showing functionality** is known as abstraction. For example: phone call, we don't know the internal processing. In java, we use abstract class and interface to achieve abstraction.
* **Inheritance**: **When one object acquires all the properties and behaviours of parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism. In java, we use **extends** a class and **implements** an interface.
* **Polymorphism**: When **one task is performed by different ways** i.e. known as polymorphism. For example: to speak something e.g. cat speaks meaw, dog barks woof etc. In java, we use method **overloading** and method **overriding** to achieve polymorphism.
* **Encapsulation**: **Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: capsule, it is wrapped with different medicines. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

# **Java Design Patterns**

**Java Design Patterns** are divided into three categories – **creational**, **structural**, and **behavioural** design patterns.

* [Creational Design Patterns](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#creational-patterns): It provide solution to instantiate an object in the best possible way for specific situations.
  + [Singleton Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#singleton-pattern)
  + [Factory Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#factory-pattern)
  + [Abstract Factory Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#abstract-factory-pattern)
  + [Builder Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#builder-pattern)
  + [Prototype Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#prototype-pattern)
* [Structural Design Patterns](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#structural-patterns)
  + [Adapter Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#adapter-pattern)
  + [Composite Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#composite-pattern)
  + [Proxy Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#proxy-pattern)
  + [Flyweight Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#flyweight-pattern)
  + [Facade Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#facade-pattern)
  + [Bridge Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#bridge-pattern)
  + [Decorator Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#decorator-pattern)
* [Behavioural Design Patterns](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#behavioral-patterns)
  + [Template Method Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#template-method-pattern)
  + [Mediator Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#mediator-pattern)
  + [Chain of Responsibility Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#chain-of-responsibility-pattern)
  + [Observer Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#observer-pattern)
  + [Strategy Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#strategy-pattern)
  + [Command Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#command-pattern)
  + [State Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#state-pattern)
  + [Visitor Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#visitor-pattern)
  + [Interpreter Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#interpreter-pattern)
  + [Iterator Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#iterator-pattern)
  + [Memento Pattern](http://www.journaldev.com/1827/java-design-patterns-example-tutorial#memento-pattern)
* **Singleton pattern:** it restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.

public class ThreadSafeSingleton {

private static ThreadSafeSingleton instance;

private ThreadSafeSingleton(){}

//thread safe Lazy Initialization Singleton

public static **synchronized** ThreadSafeSingleton getInstance(){

if(instance == null){

instance = new ThreadSafeSingleton();

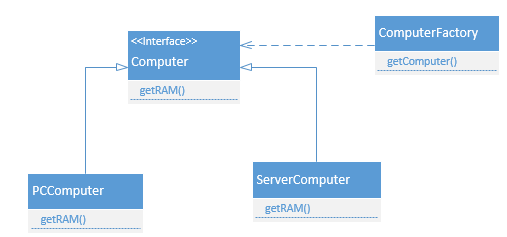
}

return instance;

}

}

* **Factory design pattern: It** is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class.



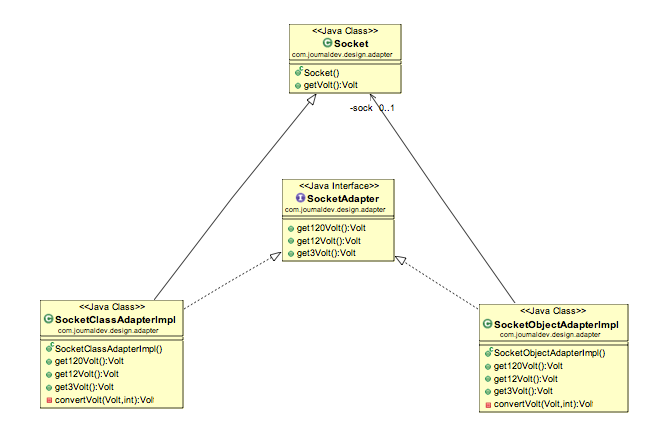
* **Prototype pattern** (Cloning): It provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.

When Would I Use This Pattern? The Prototype pattern should be considered when

* + Composition, creation and representation of objects should be decoupled from the system
  + Classes to be created are specified at runtime
  + You need to hide the complexity of creating new instance from the client
  + Creating an object is an expensive operation and it would be more efficient to copy an object.
  + Objects are required that are similar to existing objects.
* **Adapter Pattern: I**t is used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an Adapter. ***Or It*** **Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.**
  + As a real-life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

**Adapter pattern used in Java as:**

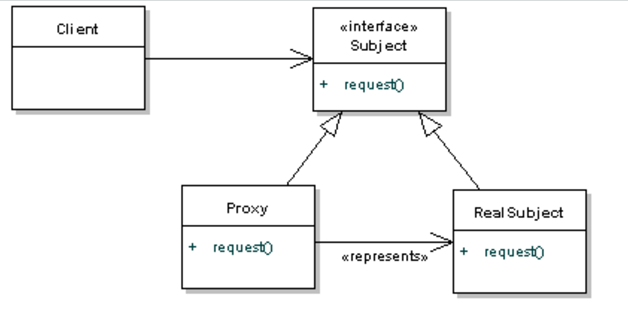
* + java.util.Arrays#asList()
  + java.io.InputStreamReader(InputStream) (returns a Reader)
  + java.io.OutputStreamWriter(OutputStream) (returns a Writer)



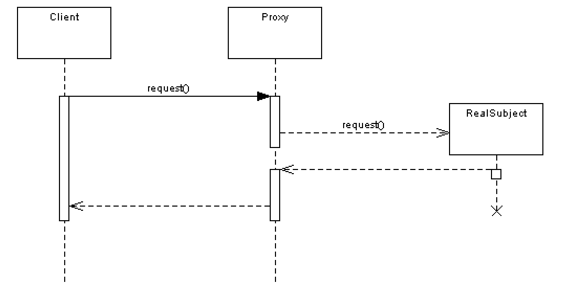
* **Facade (Service/Session Facade) Pattern:** Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

Facade Design Pattern Important Points

* + Facade design pattern is more like a helper for client applications, it doesn’t hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
  + Facade design pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
  + Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.
  + Facade design pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
  + We can use [Factory pattern](http://www.journaldev.com/1392/factory-design-pattern-in-java) with Facade to provide better interface to client systems.
* **Proxy Pattern:** Provide a surrogate or placeholder for another object to control access to it (by GoF). The definition itself is very clear and proxy design pattern is used when we want to provide controlled access of a functionality.
  + Proxy design pattern common uses are to control access or to provide a **wrapper implementation** for better performance.
  + Java RMI package uses proxy pattern. That’s all for proxy design pattern in java.



Let’s look it into sequence diagram:



As you can see it's quite simple - the Proxy is providing a barrier between the client and the real implementation.

You'll have noticed that this is very similar to the Adapter pattern. However, the main difference between bot is that the adapter will expose a different interface to allow interoperability. The Proxy exposes the same interface, but gets in the way to save processing time or memory.

# **J2EE Design Patterns**

**J2EE DESIGN PATTERNS**

|  |
| --- |
| First we should start with what are design pattern?  We can say a design pattern is simply a description of a recurring solution to a problem, given a context. The context is the environment, surroundings, situation, or interrelated conditions within which the problem exists. |

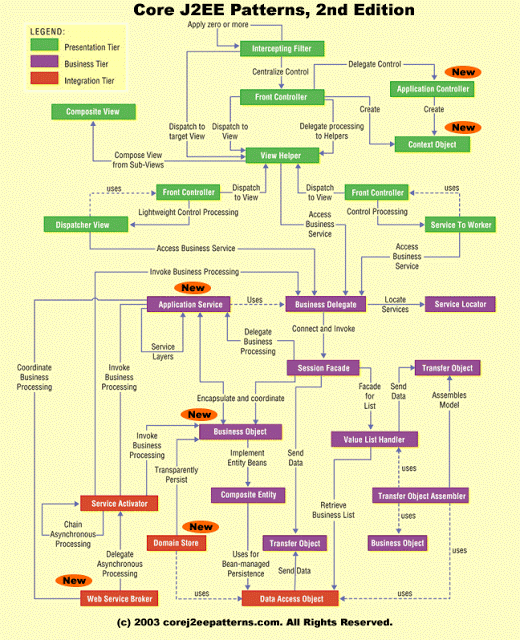
|  |
| --- |
| Second why do need to use design patterns? Design patterns have a number of advantages like - 1. Once described, any level engineer can use the pattern.  2. They allow for reuse without having to reinvent in every project.  3. They allow to better define system structure.  4. They provide a design vocabulary.  5. They provide reusable artifacts. 6. Patterns can form frameworks that can then be used for implementations. |

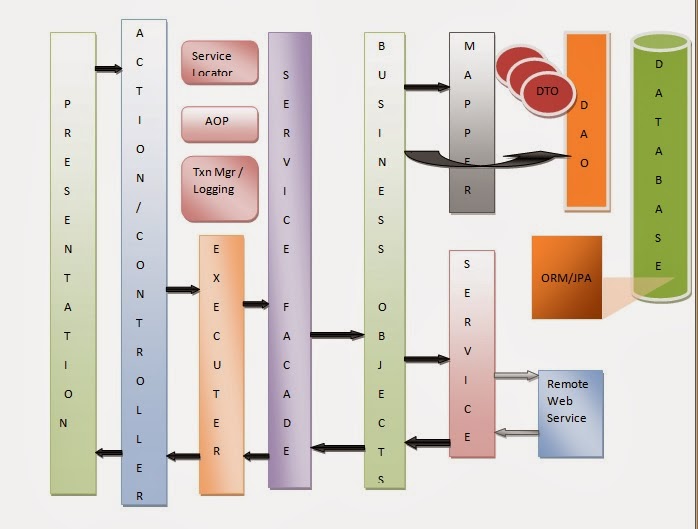
|  |
| --- |
| In this section we have tried to cover how to use and identify design patterns, in J2EE applications. The interest in design patterns has been around for a number of years in the software industry. However, interest among mainstream software developers is a fairly recent development actually it takes a highly experienced engineer to recognize a pattern, it requires collaboration, and it requires ongoing refinements. |

|  |
| --- |
| There are a number of patterns that have been identified by the Sun Java Center for the presentation tier. We have tried to explain them in this section, we are starting with a small explanation and then a complete section on each pattern. |

**Intercepting Filter**: Facilitates pre-processing and post-processing of a request.  
  
**Front Controller:** Provides a centralized controller for managing the handling of requests.  
  
**Composite View**: Creates an aggregate View from atomic subcomponents.  
  
**View Helper**: Encapsulates logic that is not related to presentation formatting into Helper components.   
  
**Dispatcher View**: Combines a Dispatcher component with the Front Controller and View Helper patterns, deferring many activities to View processing.   
  
**Service to Worker**: Combines a Dispatcher component with the Front   
Controller and View Helper patterns.   
  
**Business Delegate**: Reduces coupling between presentation-tier clients and business services. It hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.  
  
**Session Facade**: Encapsulate the complexity of interactions between the business objects participating in a workflow. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.  
  
**Service Locator**: Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.  
  
**Transfer Object Assembler**: It is used to build the required model or sub model. The Transfer Object Assembler uses Transfer Objects to retrieve data from various business objects and other objects that define the model or part of the model.  
  
**Value List Handler** :The most critical concern in a distributed paradigm is the latency time. Value List Handler Pattern suggests an alternate approach of using ejb-finder methods. The pattern is used to control the search, cache the results and provide the results to the client using a lightweight mechanism.  
  
**Composite Entity**:It model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained entity beans. A Composite Entity bean represents a graph of objects.  
  
**Transfer Object** :Encapsulate the business data. A single method call is used to send and retrieve the Transfer Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Transfer Object, populate it with its attribute values, and pass it by value to the client.  
  
**Service Activator** :Service Activator enables asynchronous access to enterprise beans and other business services. It receive asynchronous client requests and messages. On receiving a message, the Service Activator locates and invokes the necessary business methods on the business service components to fulfil the request asynchronously. In EJB2.0, Message Driven beans can be used to implement Service Activator for message based enterprise applications. The Service Activator is a JMS Listener and delegation service that creates a message façade for the EJBs.  
  
**Data Access Object** : Abstracts and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data.

Source: <http://www.allapplabs.com/j2ee_design_patterns/j2ee_design_patterns.htm>





# Design Principles

**SOLID Principles (Class Design Principles)**

* [**Single Responsibility Principle**](http://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#SRP) : One class should have one and only one responsibility.
* [**Open Closed Principle**](http://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#OCP): Software components should be open for extension, but closed for modification.
* [**Liskov's Substitution Principle**](http://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#LSP): Derived type must be completely substitutable for their base types.
* [**Interface Segregation Principle**](http://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#ISP): Clients should not be forced to implement unnecessary methods which they will not use.
* [**Dependency Inversion Principle**](http://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#DI) **:** Depend on abstraction and not on concretions.

# Writing Junit Test Class for an Existing Class

**Test Driven Development (TDD):**

What is TDD:

1. You read and understand requirements for a particular feature.
2. You develop set of tests which check the feature. All of the tests are red, due to absence of the feature implementation.
3. You develop the feature until all tests become green.
4. Refactoring of the code.

That is write unit test class first than the implementation class ( or TDD is a development of tests before a feature implementation).

A Unit test must be:

* Test behaviour but not methods
* Be small and fast
* Have good name
* Be well grained
* Be predictable

**FIRST Principles of Good Unit Tests**

Acronym FIRST stand for below test features:

* [F]ast
* [I]solated
* [R]epeatable
* [S]elf-validating
* [T]imely

**Behaviour Driven Development(BDD):**

Behaviour-driven development is an extension of [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development):

BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

BDD focused on:

* Where to start in the process
* What to test and what not to test
* How much to test in one go
* What to call the tests
* How to understand why a test fails

**A simplified Waterfall model would be:**

1. Requirements specification resulting in requirements document
2. Design resulting in software architecture document
3. Development resulting in actual software
4. Integration
5. Testing
6. Installation
7. Maintenance

Each of these phases tends to have separate teams and departments. BAs work with requirements, architects write design documents, developers code, integration engineers integrate, testers test, and someone installs the software.

# **Java J2EE JDK and JRE Related Questions and Answers**

Primitive: int, long, double, short, byte, char

Non Primitive: String, Integer, StringBuilder etc

Mutable: StringBuilder and StringBuffer (Its value can be changed without changing its reference)

Immutable: String (value cannot be changed, if changed then its reference value will be changed)

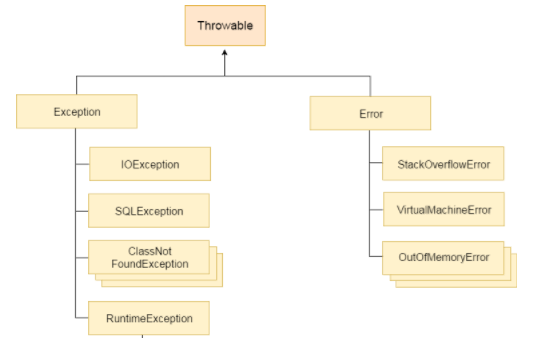
**== and equals() difference**: The == operator determines if the two references are identical to each other. And equals() method determines if the two objects are equal.

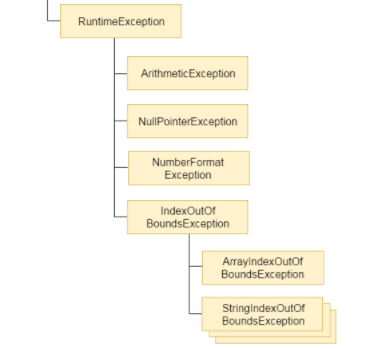
Exceptions: Three Types of Exception

**Checked exceptions** − A checked exception is an exception that occurs at the compile time. e.g.IOException, SQLException, ClassNotFoundException etc.

**Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. e.g. NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException etc.

**Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Error is irrecoverable e.g. StackOverflowError (stack error; increase -Xss size), OutOfMemoryError (Heap error; increase -Xms & -Xmx size), VirtualMachineError, AssertionError etc.





**Try-catch-finally block:** Multiple catch is applied. Nested catch is also used. Finally block always executed except when the program exit by System.exit();

**Method() throws Exception and catch throw new IOException();**

**Checked exception** can be **propagated** with throws.

**Final is keyword** used for class, method and variables.

**Finalize() is a method** to perform clean up processing just before object is garbage collected System.gc();.

Custom Exceptions:

**Public class** InvalidAgeException **extends** Exception{

 Public InvalidAgeException(String s){

**super**(s);

 }

}

**Inheritance** Class and Interface

## **Java Stack and Heap Memory:**

Java Stack memory is used for execution of a **thread**. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order.

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space.

public static void main(String[] args) { // Line 1

int i=1; // Line 2

Object obj = new Object(); // Line 3

Memory mem = new Memory(); // Line 4

mem.foo(obj); // Line 5

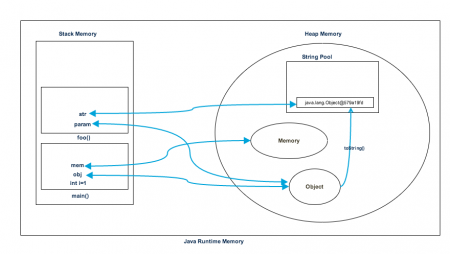
} // Line 9

private void foo(Object param) { // Line 6

String str = param.toString(); //// Line 7

System.out.println(str);

} // Line 8



1. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
2. We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory. We can use -Xss to define the stack memory size.
3. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
4. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

## **Java Collections:**

ArrayList internally uses **dynamic array** to store the elements. LinkedList internally uses **doubly linked list** to store the elements. LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.

|  |  |  |
| --- | --- | --- |
|  | **ArrayList** | **LinkedList** |
|  |  |  |
| Implementation | Resizable Array | Douby-LinkedList |
|  |  |  |
| ReverseIterator | No | Yes , descendingIterator() |
|  |  |  |
| Initial Capacity | 10 | Constructs empty list |
|  |  |  |
| get(int) operation | Fast | Slow in comparision |
|  |  |  |
| add(int) operation | Slow in comparision | Fast |
| Memory Overhead | No | Yes |

Sydnchronization is always important in multithreading programming, especially when we work with collections. In java, there are some collections that are thread safe such as **Vector**, **Stack** and **HashTable**.

public static void mapTest()

{

Map<String, String> map = new HashMap<>();

map.put("key1", "value1");

map.put("key2", "value2");

map.put("key3", "value3");

Set<Map.Entry<String, String>> entrySet = map.entrySet();

for (Map.Entry<String, String> entry : entrySet)

{

String key = entry.getKey();

String value = entry.getValue();

System.out.println(key+":"+value);

}

//OR directly we iterate values as

for (Map.Entry<String, String> entry : map.entrySet())

{

String key = entry.getKey();

String value = entry.getValue();

System.out.println(key+":"+value);

}

}

## **Java Thread:**

public class MyThread extends Thread{

public static void main(String[] args)

{

MyThread th = new MyThread();

th.start();

}

@Override

public void run()

{

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

{

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

}

}

}

public class MyRunnableThread implements Runnable{

public static void main(String[] args)

{

MyRunnableThread rth = new MyRunnableThread();

rth.run();

//OR

//(new Thread(rth)).start();

}

@Override

public void run()

{

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

{

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

}

}

}

## **Uses of equals() and hashCode() method:**

Equals() and HashCode() methods in Java are two fundamental methods from java.lang.Object class, which is used to compare equality of objects, primarily inside hash based collections such as Hashtable and HashMap.

 Both equals() and hashCode() are defined in java.lang.Object class and there default implementation is based upon Object information e.g. default equals() method return true if two objects are exactly same i.e. they are pointing to the same memory address while default implementation of hashcode method return int and implemented as a native method.

The similar default implementation of toString() method, returns type of class, followed by memory address in hex String.  
  
**Example:**

public class Person

{

private int age;

private String name;

private String address;

public Person(int age, String name, String address )

{

this.age=age;

this.name=name;

this.address=address;

}

//…..

@Override

public boolean equals(Object that) { // use same values as in hashCode

return (that instanceof Person)

&& ((Person) that).age ==this.age

&& ((Person) that).name.equals(this.name)

&& ((Person) that).address.equals(this.address);

}

@Override

public int hashCode() { // use same values as in equals

int result = 17;

result = 31 \* result + age;

result = 31 \* result + name.hashCode();

result = 31 \* result + address.hashCode();

return result;

}

@Override

public String toString() {

return String.format("%d %s (%s)", age, name, address);

}

}

public class PerHashtableTest {

public static void main(String[] args)

{

Hashtable<Person, String> person = new Hashtable<>();

//person.put(new Person(12, null, "address"), "young");//Null pointer Exception

person.put(new Person(22, "name", "address"), "young");

person.put(new Person(22, "name", "address"), "youngest");//Override the value if key is same

System.out.println("person:"+person.get(new Person(22, "name", "address")));

}

}

Output:> person:youngest

## **Garbage Collection**

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

### **Advantage of Garbage Collection**

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

### **How can an object be unreferenced?**

There are many ways:

* By nulling the reference

Employee e=**new** Employee();

e=**null**;

* By assigning a reference to another

Employee e1=**new** Employee();

Employee e2=**new** Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection

* By anonymous object etc.

**new** Employee();

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing.

**protected** **void** finalize(){}

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

**public** **static** **void** gc(){}

Example:

public class TestGarbage1 {

public void finalize()

{

System.out.println("object is garbage collected");

}

public static void main(String args[])

{

TestGarbage1 s1 = new TestGarbage1();

TestGarbage1 s2 = new TestGarbage1();

s1 = null;

s2 = null;

System.gc();

}

}