Method name can only be same in two condition only i.e. either overloading or overriding.

**OverLoad :-**

Type of argument or number of argument must be different.

Can not do overloading or same method name only from return type or modifier or through other keywords.

Overloading can be in concreate class or child class.

**Overriding :-**

Overriding only happen in child class.

Private, variable, Final, constructor and static can not be overridden.

Private/Static is always hide not override.

Can not change static to non-static or vice a versa in subclass.

We can increase scope in subclass. i.e. default to either protected or public in child class.

If method have throws runtime exception in parent/child then we can remove it in subclass/parent or throws same level excetion or throws parent/child exception i.e No restriction in case of runtime exception.

In case of throws compile time exception in parent class method, child class method can not throws upper/parent exception. Except this no restriction.

**3 ways to prevent a method from being overridden in Java**

use private, static and final.

**Exception Handling :-**

Abnormal condition raised during execution of program that can be handled is called Exception.

Abnormal condition raised during execution of program that can not be handled is called Error.

Compile Time exception :- Direct child of Exception class or child of any Compile time Exception class. JVM force to handle it at compile time either use try catch block or write throws keyword in method where abnormal condition can occur.

Throws keyword only use for propagate Exception to calling function and no error occur during compile but to handle it calling have to use try catch other wise exception occour.

RunTimeException:- Direct child of RunTimeException class or child of any subclass of this. JVM not force to use either try catch block or throws keyword. These are auto propagate to calling function without throws. These are leave on wish of programmer to handle.

Compile time exception are direct child of Exception class there is no such class like RuntimeException class.

Like class IOException extends Exception :- compile Time exception

ArthmeticException extends RunTimeException :- Runtime Exception

Can create custom Exception either compile time or run time depend only on parent class extended by custom class.

You can throw any exception custom or java defined(Compile/runtime) by using throw keyword.

Always use throws keyword with throw keyword to propagate compile time exception or handle it with try catch block.

No need to add throws keyword with throw keyword to propagate run time exception because these are auto propagate.

**class** InvalidAgeException **extends** Exception{ //compile Time Exp.

InvalidAgeException(String s){

**super**(s);

}

}

Class MainPro{

Pstn{

Try{

Validate(13);

}catch(InvalidAgeException e){}

}

**static** **void** validate(**int** age) **throws** InvalidAgeException{

**if**(age<18)

**throw** **new** InvalidAgeException("not valid");

**else**

System.***out***.println("welcome to vote");

}

}

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called *static nested classes*. Non-static nested classes are called *inner classes*.

#### Local variable can't be private, public or protected. In Java. Only can final.

**Anonymous inner class:-**

class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).

2. Interface

1. **interface** Eatable{
2. **void** eat();
3. }
4. **class** TestAnnonymousInner1{
5. **public** **static** **void** main(String args[]){
6. Eatable e=**new** Eatable(){
7. **public** **void** eat(){System.out.println("nice fruits");}
8. };
9. e.eat();
10. }
11. }
12. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
13. An object of Anonymous class is created that is referred by p reference variable of Eatable type.
14. **import** java.io.PrintStream;
15. **static** **class** TestAnonymousInner1$1 **implements** Eatable
16. {
17. TestAnonymousInner1$1(){}
18. **void** eat(){System.out.println("nice fruits");}
19. }

#### Local inner class cannot be invoked from outside the method.

#### Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

1. **class** localInner2{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **int** value=50;//local variable must be final till jdk 1.7 only
5. **class** Local{
6. **void** msg(){System.out.println(value);}
7. }
8. Local l=**new** Local();
9. l.msg();
10. }
11. **public** **static** **void** main(String args[]){
12. localInner2 obj=**new** localInner2();
13. obj.display();
14. }
15. }

**Multithreading**:-

# Executor Interfaces

The java.util.concurrent package defines three executor interfaces:

* Executor, a simple interface that supports launching new tasks.
* ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
* ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

## The Executor Interface

The [Executor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html) interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start();

with

e.execute(r);

The [ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html" \t "_blank) interface supplements execute with a similar, but more versatile submit method

## ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. The ExecutorService has the following implementation in the java.util.concurrent package:

* [**ThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)
* [**ScheduledThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

## Creating an ExecutorService

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too. Here are a few examples of creating an ExecutorService:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

## ExecutorService Usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

I will take a look at each of these methods in the following sections.

### execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

### submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.

Here is a ExecutorService submit() example:

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

### submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is an ExecutorService Callable example:

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

**difference between callable and runnable:-**

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. **A Runnable, however, does not return a result and cannot throw a checked exception.**

What is the need of having both if Callable can do all that Runnable does?

Because the Runnable interface **cannot** do everything that Callable does!

Runnable has been around since Java 1.0, but Callable was only introduced in Java 1.5 ... to handle use-cases that Runnable does not support. In theory, the Java team could have changed the signature of the Runnable.run() method, but this would have broken binary compatiblity with pre-1.5 code, requiring recoding when migrating old Java code to newer JVMs. That is a BIG NO-NO. Java strives to be backwards compatible ... and that's been one of Java's biggest selling points for business computing.

And, obviously, there are use-cases where a task doesn't need to return a result or throw a checked exception. For those use-cases, using Runnable is more concise than using Callable<Void> and returning a dummy (null) value from the call() method.

Though both the interfaces are implemented by the classes who wish to execute in a different thread of execution, but there are few differences between the two interface which are:

* A Callable<V> instance returns a result of type V, whereas a Runnable instance doesn't.
* A Callable<V> instance may throw checked exceptions, whereas a Runnable instance can't

The designers of Java felt a need of extending the capabilities of the Runnable interface, but they didn't want to affect the uses of the Runnable interface and probably that was the reason why they went for having a separate interface named Callable in Java 1.5 than changing the already existing Runnable.

The **Java Executor framework**creates tasks by using instances of Runnable or Callable. In case of Runnable, the run () method does not return a value or throw any checked exception. But Callable is a more functional version in that area. It defines a call () method that allows the return of some computed value which can be used in future processing and it also throws an exception if necessary.

The**FutureTask** class is another important component which is used to get future information about the processing. An instance of this class can wrap either a Callable or a Runnable. You can get an instance of this as the return value of **submit () method** of an **ExecutorService**. You can also manually wrap your task in a **FutureTask**before calling **execute () method**.

## Class FutureTask<V>:-

**Type Parameters:**

V - The result type returned by this FutureTask's get methods

**All Implemented Interfaces:**

[Runnable](http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html), [Future](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<V>, [RunnableFuture](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RunnableFuture.html" \o "interface in java.util.concurrent)<V>

**Construtor:-**

[**FutureTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html#FutureTask(java.util.concurrent.Callable))([**Callable**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html)<[**V**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html)> callable)

Creates a FutureTask that will, upon running, execute the given Callable.

[**FutureTask**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html#FutureTask(java.lang.Runnable,%20V))([**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) runnable, [**V**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html) result)

Creates a FutureTask that will, upon running, execute the given Runnable, and arrange that get will return the given result on successful completion.

**Methods of Future Interface used by child FutureTask class:-**

**V get()**

**Waits if necessary for the computation to complete, and then retrieves its result.**

**V get(long timeout, TimeUnit unit)**

**Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.**

**boolean isDone()**

**Returns true if this task completed.**

Future offers you method isDone() which is not blocking and returns true if computation has completed, false otherwise.

Future.get() is used to retrieve the result of computation.

You have a couple of options:

* call isDone() and if the result is ready ask for it by invoking get(), notice how there is no blocking
* block indefinitely with get()
* block for specified timeout with get(long timeout, TimeUnit unit)

**Example1:- Using Runnable Interface and FutureTask Class(implements Future).**

ExecutorService executor = Executors.newFixedThreadPool(5);

List<FutureTask<Boolean>> taskList = **new** ArrayList<FutureTask<Boolean>>();

**// FutureTask (Runnable runnable, V result)**

Creates a FutureTask that will, upon running, execute the given Runnable, and arrange that get will return the given result on successful completion.

Method Summary

FutureTask<Boolean> pollTask24 = **new** FutureTask<Boolean>(**new** Runnable() {

@Override

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

**do stuff**

**} },true);**

taskList.add(pollTask24); //Simple ArrayList to check work is done or not

executor.execute(pollTask24);

// Future task get() method will Waits if necessary for the computation to complete, and then retrieves its result.

// Wait until all results are available(use get method of Future Task). **for** (**int** j = 0; j < taskList.size(); j++)

{

Log.info("Contact\_center :: Waiting for thread to get completed");

FutureTask<Boolean> futureTask = taskList.get(j);

**try** {

Log.info("Contact\_center :: For the task[" + j + "]result is[" + futureTask.get() + "]");

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

Log.fatal("Contact\_center ::Exception encountered while waiting for thread to get completed", e);

}

}

executor.shutdownNow();

**Example2:- Using callable Interface and Future Interface**

package test;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import java.util.logging.Level;

import java.util.logging.Logger;

/\*\*

\* Java program to show how to use Future in Java. Future allows to write

\* asynchronous code in Java, where Future promises result to be available in

\* future

\*

\* @author Javin

\*/

public class FutureDemo {

private static final ExecutorService threadpool = Executors.newFixedThreadPool(3);

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

FactorialCalculator task = new FactorialCalculator(10);

System.out.println("Submitting Task ...");

Future future = **threadpool.submit(task);**

System.out.println("Task is submitted");

while (!future.isDone()) {

System.out.println("Task is not completed yet....");

Thread.sleep(1); //sleep for 1 millisecond before checking again

}

System.out.println("Task is completed, let's check result");

long factorial = future.get(); //it will also wait if not done but here we already conform it already.

System.out.println("Factorial of 1000000 is : " + factorial);

threadpool.shutdown();

}

private static class FactorialCalculator implements Callable {

private final int number;

public FactorialCalculator(int number) {

this.number = number;

}

@Override

public Long call() {

long output = 0;

try {

output = factorial(number);

} catch (InterruptedException ex) {

Logger.getLogger(Test.class.getName()).log(Level.SEVERE, null, ex);

}

return output;

}

private long factorial(int number) throws InterruptedException {

if (number < 0) {

throw new IllegalArgumentException("Number must be greater than zero");

}

long result = 1;

while (number > 0) {

Thread.sleep(1); // adding delay for example

result = result \* number;

number--;

}

return result;

}

}

}

Output

Submitting Task ...

Task is submitted

Task is not completed yet....

Task is not completed yet....

Task is not completed yet....

Task is completed, let's check result

Factorial of 1000000 is : 3628800

**Example3:-**

**package** overloading;

**import** java.util.ArrayList;

**import** java.util.Date;

**import** java.util.List;

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

**import** java.util.concurrent.ExecutionException;

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

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

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

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

@Override

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

Thread.*sleep*(1000);

//return the thread name executing this callable task

**return** Thread.*currentThread*().getName();

}

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

//Get ExecutorService from Executors utility class, thread pool size is 10

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

//create a list to hold the Future object associated with Callable

List<Future<String>> list = **new** ArrayList<Future<String>>();

//Create MyCallable instance

Callable<String> callable = **new** MyCallable();

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

//submit Callable tasks to be executed by thread pool

Future<String> future = executor.submit(callable);

//FutureTask<String> future = (FutureTask<String>) executor.submit(callable);

//add Future to the list, we can get return value using Future

list.add(future);

}

**for**(Future<String> fut : list){

**try** {

//print the return value of Future, notice the output delay in console

// because Future.get() waits for task to get completed

System.***out***.println(**new** Date()+ "::"+fut.get());

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

e.printStackTrace();

}

}

//shut down the executor service now

executor.shutdown();

}

}

**Example 4:-**

**package** overloading;

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

**import** java.util.concurrent.ExecutionException;

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

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

**import** java.util.concurrent.FutureTask;

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

**import** java.util.concurrent.TimeoutException;

**public** **class** FutureTaskExample {

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

MyCallableTest callable1 = **new** MyCallableTest(1000);

MyCallableTest callable2 = **new** MyCallableTest(2000);

FutureTask<String> futureTask1 = **new** FutureTask<String>(callable1);

FutureTask<String> futureTask2 = **new** FutureTask<String>(callable2);

ExecutorService executor = Executors.*newFixedThreadPool*(2);

executor.execute(futureTask1);

executor.execute(futureTask2);

**while** (**true**) {

**try** {

**if**(futureTask1.isDone() && futureTask2.isDone()){

System.***out***.println("Done");

//shut down executor service

executor.shutdown();

**return**;

}

**if**(!futureTask1.isDone()){

//wait indefinitely for future task to complete

System.***out***.println("FutureTask1 output="+futureTask1.get());

}

System.***out***.println("Waiting for FutureTask2 to complete");

//wait only 200 milliseconds overloaded get () method in Future Interface

String s = futureTask2.get(200L, TimeUnit.***MILLISECONDS***);

**if**(s !=**null**){

System.***out***.println("FutureTask2 output="+s);

}

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

e.printStackTrace();

}**catch**(TimeoutException e){

//do nothing

}

}

}

}

**class** MyCallableTest **implements** Callable<String> {

**private** **long** waitTime;

**public** MyCallableTest(**int** timeInMillis){

**this**.waitTime=timeInMillis;

}

@Override

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

Thread.*sleep*(waitTime);

//return the thread name executing this callable task

**return** Thread.*currentThread*().getName();

}

}