Contents

[Java 8 Features – Functional Programming 3](#_Toc64225020)

[**Lambdas** 3](#_Toc64225021)

[Lambda Expressions 3](#_Toc64225022)

[**Functional Interface** 4](#_Toc64225023)

[Java Executor Service 8](#_Toc64225024)

[SIMPLE THREAD CREATION IN JAVA 8](#_Toc64225025)

[CONCEPT OF THREAD POOL 9](#_Toc64225026)

[JAVA CLASS LEVERAGING THREAD POOL 9](#_Toc64225027)

[IDEAL THREAD POOL SIZE 11](#_Toc64225028)

[For CPU INTENSIVE OPERATION 11](#_Toc64225029)

[FOR IO OPERATIONS 12](#_Toc64225030)

[TYPES OF THREAD POOLS 12](#_Toc64225031)

[Fixed Thread Pool 12](#_Toc64225032)

[Cached Thread Pool 13](#_Toc64225033)

[Scheduled Thread Pool 14](#_Toc64225034)

[CALLABLE & FUTURES 14](#_Toc64225035)

[JUNIT 5 17](#_Toc64225036)

[ARCHITECTURE 17](#_Toc64225037)

[PLATFORM 17](#_Toc64225038)

[JUPITER 17](#_Toc64225039)

[VINTAGE 17](#_Toc64225040)

[EXTERNAL (3RD PARTY) 17](#_Toc64225041)

[SAMPLE POM.XML/ DEPENDENCIES 17](#_Toc64225042)

[**MAVEN SUREFIRE PLUGIN - INTEGRATION** 19](#_Toc64225043)

[**JUNIT TEST LIFE CYCLE** 19](#_Toc64225044)

[LIFE CYCLE HOOKS 20](#_Toc64225045)

[SomE USEFUL ANNOTATIONS 21](#_Toc64225046)

[JAVA FSE NOTES 22](#_Toc64225047)

[EXCEPTION HANDLING 22](#_Toc64225048)

[POLYMORPHISM & ABSTRACTION 22](#_Toc64225049)

[ACCESS MODIFIERS 22](#_Toc64225050)

[TYPE CASTING 23](#_Toc64225051)

[STATIC 23](#_Toc64225052)

[STATIC BLOCKS 23](#_Toc64225053)

[STATIC IMPORTS 23](#_Toc64225054)

[NESTED CLASS 23](#_Toc64225055)

[BOOTSTRAP BASICS 25](#_Toc64225056)

[META TAGS ON BOOTSTRAP 25](#_Toc64225057)

[FLEX BOX GRIDS 25](#_Toc64225058)

# Java 8 Features – Functional Programming

1. ***Lambda Expressions***
2. ***Functional Interface***
3. ***Method references***
4. ***Stream API***
5. ***Default & Static methods in Interface.***
6. ***Optional Class***

## **Lambdas**

1. Enables functional programming
2. Readable and concise code
3. Easier-to-use API and Libraries
4. Enable support for parallel processing

**CONCEPTS OF LAMBDA**

Lambda is just function that exists in isolation and can be passed as we do in functional programing Languages like JavaScript.

|  |  |
| --- | --- |
| var printName = function(){  console.log("Hello World!");  }  var showName= function(fn){  fn();  }  showName(printName); | IN THE JS CODE (Function Programming)   1. “printName” is a variable which contains a value of function type. 2. As it exists in isolation it can be passed to another function as well as parameter. 3. Till Java 1.7 we cannot able to do this because all the method is always tied to a class. No method can exist in java in isolation. 4. Java1.8 enables this feature using Lambda expression where method can exist in isolation and can be treated as values 5. As function is can be values so it can be passed as parameter. |

**ANALOGY OF LAMBDAS WITH JS**

|  |  |
| --- | --- |
|  | * In Java 1.8 the variable which stores the function has be a functional interface * The RHS of the statement will be a Lambda expression. |

## Lambda Expressions

**Lambda expressions has “no name”, no return types and no Modifiers.**

|  |  |
| --- | --- |
| **JAVA METHOD** | **EQUIVALENT LAMBDA EXPRESSION** |
| **public void print**(){ System.out.println(“Hello”); } | () -> System.out.println(“Hello”); |
| public void add(int a, int b){ System.out.println(a+b); } | (a,b) -> System.out.println(a+b); |
| public int square(int n){  return n\*n;  }  EQUIVALENT LAMBDA EXPRESSION  **n->n\*n** | Step 1 : (n) ->{return n\*n;}   * **If we have only 1 parameter parenthesis are optional** * If compiler can able to predict the return type we can remove the return type * If we have single statement brackets can be removed |

## **Functional Interface**

|  |  |
| --- | --- |
| Prior to Java 1.8 – Interface can have only abstract method, but from Java 1.8 interfaces can have   1. Abstract methods 2. Default methods 3. Static methods   All methods in interface are public methods | public interface Calculator {  public abstract int add(int num1, int num2);  default void print(){  System.out.println("Calculator");  }  public static void calculate(){  System.out.println("Calculator Numbers");  }} |
| * The Interface which has only one abstract method is called Functional Interface. It can have any number of default and static methods. * To make sure interface is adhered to the above rule of functional interface it can be annotated with an annotation @FunctionalInterface | @FunctionalInterface  public interface Calculator {  public abstract int add(int num1, int num2);  default void print(){ }  public static void calculate(){ }  } |

**HOW TO USE LAMBDA EXPRESSION**

Lambda expression can be used in context of functional interface only

|  |  |  |
| --- | --- | --- |
| **TRADITIONAL WAY TO USE AN INTERFACE** | | |
| INTERFACE  public interface Calculator {  public abstract int add(int num1, int num2);  }  IMPL CLASS  public class CalculatorImpl implements Calculator {  @Override  **public** **int** add(**int** num1, **int** num2) {  **return** num1 + num2;  }  } | | CALLING THE METHOD  public class LamdaExpression {  public static void main(String[] args) {  Calculator calculator = new CalculatorImpl();  System.*out*.println(calculator.add(1, 2));  }  } |
| * Similar functionality can be achieved by using functional interface and Lambda expression without creating a “**CalculatorImpl**” class. * Lambda expression checks only the function signature in the functional interface in the LHS. * Lambda expressions are actually an implementation of the method of functional interface. | | |
| Example1  Step 1: Create a functional interface  **public** **interface** Calculator {  **public** **abstract** **int** add(**int** num1, **int** num2);  } | **Step 2: Call the method in the functional interface using Lambda expression**  **public** **class** LamdaExpression {  **public** **static** **void** main(String[] args) {  Calculator calculator = (a, b) -> a + b;  System.***out***.println(calculator.add(1, 2));  }} | |
| Example 2:  **public** **interface** Square {  **public** **abstract** **int** square(**int** num);  } | **public** **class** LamdaExpression {  **public** **static** **void** main(String[] args) {  Square square = n->n\*n;  System.***out***.println(square.square(2));  }} | |
| PASSING THE LAMBDA EXPRESSION AS VALUES  public class LamdaExpression {  public static void main(String[] args) {  Calculator calculator = (num1, num2) -> num1 + num2;  *calculateAdd*(calculator);  }  public static void calculateAdd(Calculator calculator) {  System.*out*.println(calculator.add(1, 2));  }  } | | |

|  |  |
| --- | --- |
| **Example:**   1. Create a Person object 2. Sort the person based on Last Name 3. Print All person 4. Print person whose first name start with “J” | public class Person {  private String firstName;  private String lastName;  public Person(String firstName, String lastName) {  super();  this.firstName = firstName;  this.lastName = lastName;  }  // getter & setters  @Override toString() {}  } |
| FUNCTIONAL INTERFACE  public interface PersonFilter {  public abstract boolean test(Person person);  } | |
| LAMBDA EXPRESSION  public class Exercise {  public static void main(String[] args) {  List<Person> persons = Arrays.*asList*(new Person("Charles", "Dickens", 25),  new Person("John", "Doe", 28),  new Person("Samanta", "Beth", 17),  new Person("Dan", "Wilson", 60));  /\*Sort Person Object with Last Name\*/  Comparator<Person> comparator = (p1, p2) -> p1.getLastName().compareTo(p2.getLastName());  Collections.*sort*(persons, comparator);  /\*Print All\*/  *printPersons*(persons,p -> true);  /\*Filter Person's first name start with "J"\*/  *printPersons*(persons,p -> p.getFirstName().startsWith("J"));  /\*Filter Person's Last name start with "W"\*/  *printPersons*(persons,p -> p.getLastName().startsWith("W"));  }  public static void printPersons(List<Person> persons, PersonFilter personFilter) {  for (Person person : persons) {  if (personFilter.test(person)) {  System.*out*.println(person);  }  }  }  } | |

**EXCEPTION HANDLING IN LAMBDA**

**VALID AND INVALID LAMBDA EXPRESSIONS**

|  |  |
| --- | --- |
| **n->n\*n** | Valid |
| **(n)->n\*n** | **Valid** |
| **n->return n\*n** | Invalid. if we are using a return statement , we need to have curly braces “{}” |
| **n->{return n\*n;};** | Valid |
| **n->{return n\*n};** | Invalid |
| **n->{return n\*n;}** | Invalid |
| **n->{ n\*n;};** | Invalid . Because of we have curly braces we need to have return statement |

**RUNNABLE INTERFACE IMPLEMENTATION USING LAMBDA EXPRESSION**

**Example**

**public** **class** LamdaExpression {

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

Runnable runnable = () -> {

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

System.***out***.println(i + "-" + Thread.*currentThread*().getName());

};

Thread thread = **new** Thread(runnable);

thread.setName("Child Thread");

thread.start();

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

System.***out***.println(i + "-" + Thread.*currentThread*().getName());

}

}

# Java Executor Service

* Threads are the way of executing tasks in an asynchronously

## SIMPLE THREAD CREATION IN JAVA

|  |  |
| --- | --- |
| Java Class  public class ThreadDemo {  public static void main(String[] args) {  Thread thread = new Thread(new Task());  thread.start();  System.out.println(Thread.currentThread().getName());  }  }  class Task implements Runnable {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  } |  |
| STARTING MULTIPLE THREAD  public class ThreadDemo {  public static void main(String[] args) {  for(int i =0 ; i<10; i++){  Thread thread = new Thread(new Task());  thread.start();  System.out.println(Thread.currentThread().getName());  }  }  }  class Task implements Runnable {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  } |  |

## CONCEPT OF THREAD POOL

|  |  |  |
| --- | --- | --- |
| Considering the above example - let’s say we thousands of task that has to be performed. Then it will create thousands of threads to perform that task. But creating a thread is itself an expensive task because  ***1 Java thread = 1 OS thread*** | |  |
| So the idea is why we can’t have fixed number for threads and we submit those 1000s tasks to those threads.  Those fixed number thread will pick the submitted task, completes it and then pick another one.  Those fixed number of threads are called **THREAD POOL** | |  |
| JAVA CLASS LEVERAGING THREAD POOL **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  //Create a Thread Pool  **ExecutorService executorService = Executors.*newFixedThreadPool*(10);**  //submit the task for execution  **for**(**int** taskCount =0 ; taskCount<1000; taskCount++){  **executorService.execute(new Task());**  }  System.***out***.println(Thread.*currentThread*().getName());  }  }  **class** Task **implements** Runnable {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  }  } | | |
| 1. All the thread in the thread pool fetches the task from the queue and executes it. 2. While fetching the task, there is a possibility of race condition when the threads tries to access the task concurrently. So to avoid such situation ExecutorService uses a Blocking Queue, which is a thread safe queue. |  | |

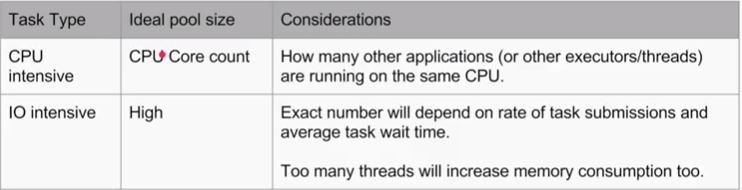
## IDEAL THREAD POOL SIZE

### For CPU INTENSIVE OPERATION

|  |  |
| --- | --- |
| The pool size always depends on the type of task we doing using thread. Let’s say we are doing a CPU intensive operation.  So if you have a CPU having 4 core then maximum 4 threads can run at a time irrespective of the thread pool size.  Even if have large number of thread in the pool, they will be execute in a Round Robin fashion.  So the best idea is for CPU intensive operation, the thread pool size should be equal to CPU cores. |  |
| **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **public** **class** CPUIntensiveTask **implements** Runnable {  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  }  **public** **static** **void** main(String[] args) {  //get count of available cores  **int threadPoolSize = Runtime.*getRuntime*().availableProcessors();**  ExecutorService executorService = Executors.*newFixedThreadPool*(threadPoolSize);  **for**(**int** taskCount = 0; taskCount<100;taskCount++){  executorService.execute(**new** CPUIntensiveTask());  }  }  } | |

### FOR IO OPERATIONS

|  |  |
| --- | --- |
| These operation involves the task like fetch the data from DB or the task is to get the data from an HTTP url  Whenever these kinds of tasks are performed by a thread, other threads go in a waiting state. So even if we have 4 cores  In these scenarios we keep the pool size higher |  |



## TYPES OF THREAD POOLS

Java provides 4 types of thread pool

1. **FixedThreadPool**
2. **CachedThreadPool**
3. **ScheduledThreadPool**
4. **SingleThreadedExecutor**

### Fixed Thread Pool

|  |  |
| --- | --- |
| * The thread pool size is fixed * We keep submitting the task to same set of thread |  |

### Cached Thread Pool

|  |  |
| --- | --- |
| 1. It will not have fixed number of threads 2. It does not main a queue of tasks.   Instead of this   1. It maintains a queue called synchronous queue, which always have one task at a time. 2. Every time we submit a particular task , the pool will hold the task in the synchronous queue and look for a thread in the pool which is free. 3. If it won’t finds a free thread , it creates a new thread in the pool and executes the task 4. If the thread is idle for 60 secs then it kills that particular thread. |  |
| public class ThreadDemo {  public static void main(String[] args) {  //Create a Cached Thread Pool. We are not passing the pool size.  ExecutorService executorService = Executors.*newCachedThreadPool*();  //submit the task for execution  for(int taskCount =0 ; taskCount<1000; taskCount++){  executorService.execute(new Task());  }  System.*out*.println(Thread.*currentThread*().getName());  }  }  class Task implements Runnable {  @Override  public void run() {  System.*out*.println(Thread.*currentThread*().getName());  }  } | |

### Scheduled Thread Pool

|  |  |
| --- | --- |
| This kind of thread pools are used when we want the task to execute after certain delay. |  |

### CALLABLE & FUTURES

Whenever we implement runnable we don’t have scope returning a value from the Task. What if the task is implemented in such that that it return value .The way we can able to return value from the task is by implementing “**Callable**”

1. The callable interface has a generic based method “call” which returns a value.
2. The “**submit()”** method is used to submit a callable or runnable task. Note we used “execute()” method cannot be used to submit callable task
3. The return type of the call method is stored in a variable of type **Future (**not Integer**)**
4. Future acts a placeholder that will arrive sometime in future (from call method). The arrival of value in Future depends upon the time taken by call method(task)
5. Get() 🡪 This method give the actual return type of the call method. It’s a blocking operation , because the get() will block the main thread will be till call() method returns value

|  |
| --- |
| TASK : **call method returns a value, which is a generic type**.  **public** **class** TaskForThread **implements** Callable<Integer> {  @Override  **public** Integer call() **throws** Exception {  **return** **new** Random().nextInt();  }  } |
| **SIMPLE FUTURE**  **public** **class** SimpleFuture {  **public** **static** **void** main(String[] args) {  ExecutorService executorService = Executors.*newFixedThreadPool*(10);  Future<Integer> future = executorService.submit(**new** TaskForThread());  **try** {  System.***out***.println(future.get());  } **catch** (InterruptedException | ExecutionException e) {  e.printStackTrace();  }  }  } |
| **MULTIPLE FUTURES**  **public** **class** CallableThreadDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  ExecutorService executorService = Executors.*newFixedThreadPool*(10);  List<Future<Integer>> allFutures = **new** ArrayList<Future<Integer>>();  **for**(**int** taskCount = 0 ; taskCount<100 ; taskCount++){  **Future<Integer> future = executorService.submit(new TaskForThread());**  allFutures.add(future);  }  **for**(Future<Integer> future:allFutures)  System.***out***.println(future.get());  }} |

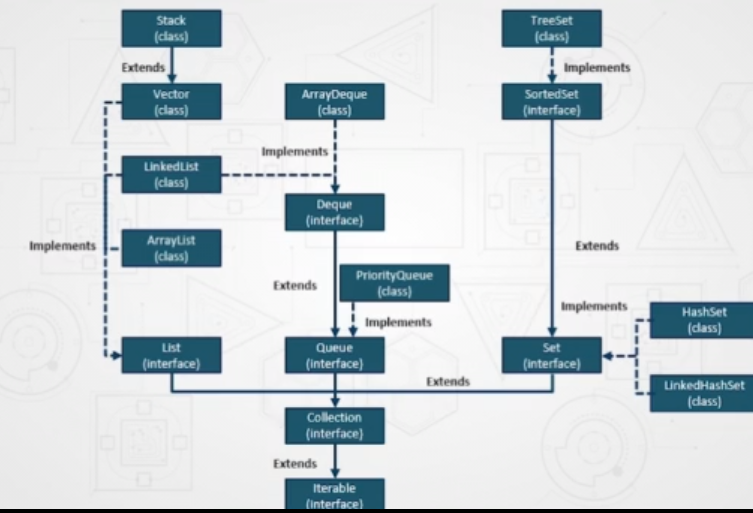
**VISUALIZATION OF CALLABLE TASK**

|  |  |
| --- | --- |
| As shown in diagram   1. Future act as placeholder for the return value of call method 2. The placeholder is filled once the result is ready . 3. Later the get() method is used to fetch the value from the place holder |  |
| Get() method is a blocking operation , because the get() will block the main thread will be till we get method returns value |  |

# JAVA FSE NOTES

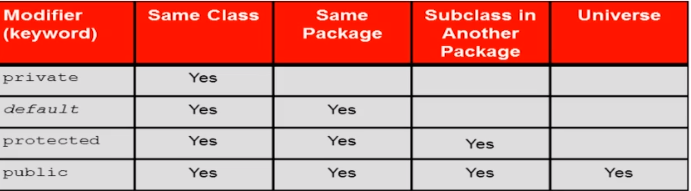
## EXCEPTION HANDLING

1. When the exception is not handles the exception finally propagates to JVM.



## POLYMORPHISM & ABSTRACTION

### ACCESS MODIFIERS



#### METHOD OVERRIDING

* The overriding method cannot have more restricted access modifier than the overridden method.
* Equals() and hashCode() methods are Object class methods. The general contract for Object states if two objects are considered to be equal (using equals method), then integer hashcode returned for those two object should also be equal.

##### VIRTUAL METHOD INVOCATION

* Virtual method invocation is the invocation of the correct overridden method, which is based on the type of the object referred to by an object reference and not by the object reference itself.
* It’s determined at runtime, not at compilation time.

### TYPE CASTING

* Upward casting is always permitted and does not need a casting operator

|  |  |
| --- | --- |
|  | CASTING WILL FAIL |

### STATIC

* Static keywords are used to declare the field & methods as class level resources
* Static class members
  + Can be used without object instance
  + Can be used when objects of the same type need to share the field
* Static methods
  + Also called class methods
  + Cannot able to access non-static members withing the same class
* Static variables
  + Called class variables
  + It has single copy per JVM. All object instances shares a single copy of single variable.
  + Initialized when the class is first loaded

### STATIC BLOCKS

* Static initialization blocks are called only once when the class is first loaded
* Can be used to initialize the variables during the class load.
* Class can have any number of static blocks and will be called in the order it is written in source code.

### STATIC IMPORTS

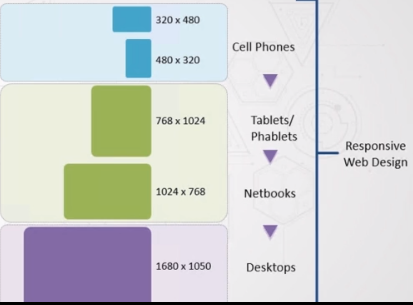
Ssd

### NESTED CLASS

* Nested class are the class declared withing another class. They are used as helper class. The nested class has multiple categories
  + **STATIC NESTED CLASS**
  + **INNER CLASS**
    - Method Local inner class 🡪 Non static class defined inside a method
    - Anonymous Classes🡪 Local inner class without a name
    - Member Class 🡪 Defined within the class(outer class)

## BOOTSTRAP BASICS

**WHY BOOTSTRAP?**

* Responsive Grid System (12 column grids)
* ****Ready to use component
* In build themes
* Little or no programing needed

**WHAT BOOTSTAP NEEDS TO RUN?**

* jQuery Library
* Popper.js . Popper.js is used for
  + TOOLTIP & POPOVER
  + POSITIONING ENGINE
* Bootstrap.Js

## META TAGS ON BOOTSTRAP

Example

* <meta charset="utf-8">
* <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

## FLEX BOX GRIDS

* Flex grid helps in positioning the elements without using any explicit floats . For example - The content which can are getting rendered in multi columns in desktop view , If they are expected to get rendered in a single column in mobile devices- The flex box grid system help in achieving this.

**EXAMPLE**

|  |  |
| --- | --- |
| <div class="container-fluid">  <div class="d-flex flex-column">  <div>Red</div>  <div>Blue</div>  <div>Green</div>  </div>  </div> |  |
| <div class="container-fluid">  <div class="d-flex flex-column">  <div class="align-self-start">Red</div>  <div class="align-self-end">Blue</div>  <div class="align-self-center">Green</div>  </div>  </div> |  |
| * Flex grid system helps in positioning the element |

*Native Font Stack*