

An Interface that contains exactly one abstract method is known as **functional interface**. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class. Functional Interface is also known as Single Abstract Method Interfaces or **SAM** Interfaces.

@FunctionalInterface

**public** **interface** Supplier<**T**> {

**T** get();

}

final Supplier<Employee> supplier = () -> new Employee("empId","","");

Consumer is a functional interface. Like Supplier, it has one abstract functional method accept(T t)and a default method andThen(Consumer<? super T> after)

@FunctionalInterface

**public** **interface** Consumer<T> {

**void** accept(T t);

**default** Consumer<T> andThen(Consumer<? **super** T> after) {

Objects.*requireNonNull*(after);

**return** (T t) -> { accept(t); after.accept(t); };

}

}

**final** Consumer<MdmsTitleId> createAndSaveDeleteReceipt = (**final** MdmsTitleId identity) -> createDeletedAssetReceipt(identity);

createAndSaveDeleteReceipt.accept(identity);

**createAndSaveDeleteReceipt.accept(identity);**

**Accessing private members using reflection**

The following code shows how to access private member variables and methods in Java.

**Foo class**

@SuppressWarnings("unused")

**public** **class** Foo {

**private** **int** a = 10;

**private** **void** m() {

System.*out*.println("this is private method from class Foo");

}

**Bar class**

**class** Bar {

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

Foo foo = Foo.**class**.newInstance();

Class<Foo> clazz = Foo.**class**;

Method[] methods = clazz.getDeclaredMethods();

**for** (**int** i = 0; i < methods.length; i++) {

methods[i].setAccessible(**true**);

methods[i].invoke(foo);

}

Field[] fields = clazz.getDeclaredFields();

**for** (**int** i = 0; i < fields.length; i++) {

fields[i].setAccessible(**true**);

fields[i].setInt(foo, 1);

System.*out*.println(fields[i].getInt(foo));

}

}

}

Now to restrict it below two ways

1. Adding code in private method

**public** **class** Foo {

**private** **int** a = 10;

**private** **void** m() {

/\*\*One way to restrict access checks [**Only Applicable for methods**] \*/

ReflectPermission perm = **new** ReflectPermission ("suppressAccessChecks", "");

AccessController.*checkPermission*(perm);

System.*out*.println("this is private method from class Foo");

}

1. Adding custom security manager code

**public** **class** Foo {

**static** {

**try** {

System.*setSecurityManager*(**new** MySecurityManager());

} **catch** (SecurityException se) {

System.*out*.println("SecurityManager already set!");

}

}

**private** **int** a = 10;

**private** **void** m() {

System.*out*.println("this is private method from class Foo");

}

}

**class** MySecurityManager **extends** SecurityManager {

@Override

**public** **void** checkPermission(Permission perm) {

**if** (perm.getName().equals("suppressAccessChecks"))

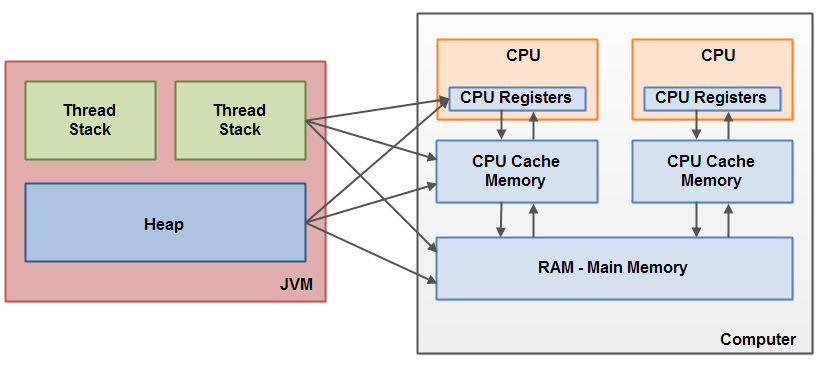
**throw** **new** SecurityException("Can not change the permission.");

}

}

## Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture

As already mentioned, the Java memory model and the hardware memory architecture are different. The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers. This is illustrated in this diagram:



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. **The two main problems are**:

* Visibility of thread updates (writes) to shared variables. (volatile)
* Race conditions when reading, checking and writing shared variables. (Synchronization)

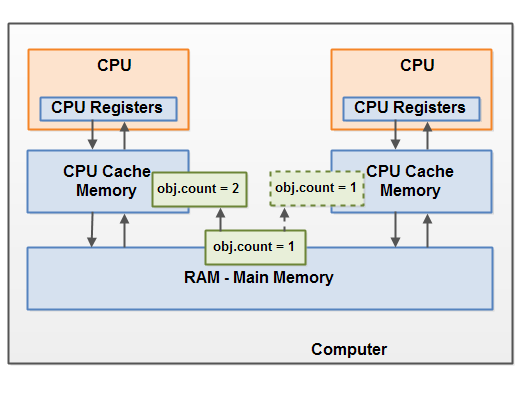
Both of these problems will be explained in the following sections.

### Visibility of Shared Objects

If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to the shared object made by one thread may not be visible to other threads.

Imagine that the shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. There it makes a change to the shared object. As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.

The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2. This change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.



To solve this problem you can use [Java's volatile keyword](http://tutorials.jenkov.com/java-concurrency/volatile.html). The volatile keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.

### Race Conditions

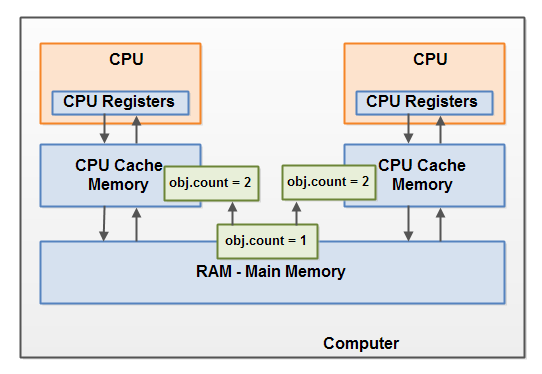
If two or more threads share an object, and more than one thread updates variables in that shared object, [race conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) may occur.

Imagine if thread A reads the variable count of a shared object into its CPU cache. Imagine too, that thread B does the same, but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.

If these increments had been carried out sequentially, the variable count would be been incremented twice and had the original value + 2 written back to main memory.

However, the two increments have been carried out concurrently without proper synchronization. Regardless of which of thread A and B that writes its updated version of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments.

This diagram illustrates an occurrence of the problem with race conditions as described above:



To solve this problem you can use a [Java synchronized block](http://tutorials.jenkov.com/java-concurrency/synchronized.html). A synchronized block guarantees that only one thread can enter a given critical section of the code at any given time. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

**ThreadLocal Class**

The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to a ThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables.

**Parallel Programming**

Parallel computing or parallelization is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (“in parallel”). In essence, if a CPU intensive problem can be divided in smaller, independent tasks, then those tasks can be assigned to different processors.

Java Fork and Join using ForkJoinPool

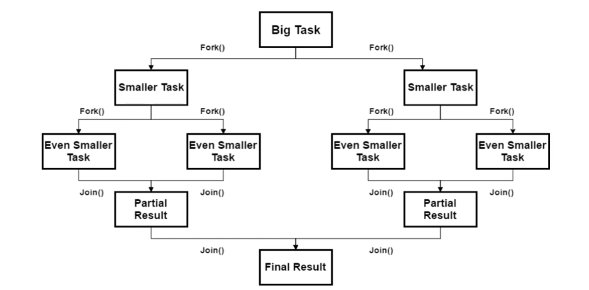
The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the [Java ExecutorService](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html) but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Tasks can keep splitting their work into smaller subtasks for as long as it makes to split up the task. It may sound a bit abstract, so in this fork and join tutorial I will explain how the ForkJoinPool works, and how splitting tasks up work.

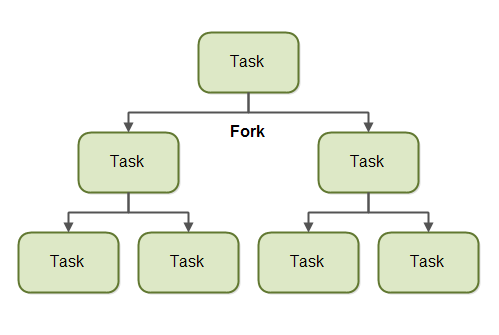
## Fork and Join Explained

Before we look at the ForkJoinPool I want to explain how the fork and join principle works in general. The fork and join principle consists of two steps which are performed recursively. These two steps are the fork step and the join step.

### Fork

A task that uses the fork and join principle can fork (split) itself into smaller subtasks which can be executed concurrently. This is illustrated in the diagram below:



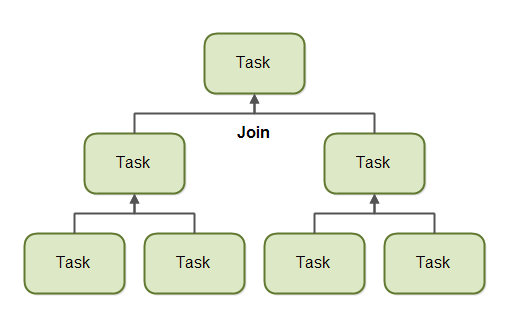


By splitting itself up into subtasks, each subtask can be executed in parallel by **different CPUs**, or **different threads on the same CPU**. A task only splits itself up into subtasks if the work the task was given is large enough for this to make sense. There is an overhead to splitting up a task into subtasks, so for small amounts of work this overhead may be greater than the speedup achieved by executing subtasks concurrently.

The **limit** for when it makes sense to fork a task into subtasks is also called a **threshold**. It is up to each task to decide on a sensible threshold. It depends very much on the kind of work being done.

### Join

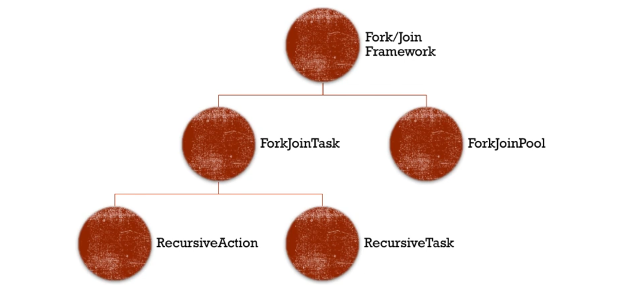
When a task has split itself up into subtasks, the task waits until the subtasks have finished executing. Once the subtasks have finished executing, the task may *join* (merge) all the results into one result. This is illustrated in the diagram below:



Of course, not all types of tasks may return a result. If the tasks do not return a result then a task just waits for its subtasks to complete. No result merging takes place then.

## The ForkJoinPool

The ForkJoinPool is a special thread pool which is designed to work well with fork-and-join task splitting. The ForkJoinPool located in the java.util.concurrent package, so the full class name is java.util.concurrent.ForkJoinPool.



### Creating a ForkJoinPool

You create a ForkJoinPool using its constructor. As a parameter to the ForkJoinPool constructor you pass the indicated level of parallelism you desire. The parallelism level indicates how many threads or CPUs you want to work concurrently on tasks passed to the ForkJoinPool. Here is a ForkJoinPool creation example:

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

This example creates a ForkJoinPool with a parallelism level of 4. So generally take the all core available.

int cores = Runtime.getRuntime().availableProcessors();

### Submitting Tasks to the ForkJoinPool

You submit tasks to a ForkJoinPool similarly to how you submit tasks to an ExecutorService. You can submit **two** **types** of tasks. A task that does not return any result (an "action"), and a task which does return a result (a "task"). These two types of tasks are represented by the RecursiveAction and RecursiveTask classes. How to use both of these tasks and how to submit them will be covered in the following sections.

## RecursiveAction

A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.

A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.

You implement a RecursiveAction by subclassing it. Here is a RecursiveAction example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

private long workLoad = 0;

public MyRecursiveAction(long workLoad) {

this.workLoad = workLoad;

}

@Override

protected void compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveAction> subtasks =new ArrayList<>();

subtasks.addAll(createSubtasks());

for(RecursiveAction subtask : subtasks){

subtask.fork();

}

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

}

}

private List<MyRecursiveAction> createSubtasks() {

List<MyRecursiveAction> subtasks = new ArrayList<>();

MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);

MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is very simplified. The MyRecursiveAction simply takes a fictive workLoad as parameter to its constructor. If the workLoad is above a certain threshold, the work is split into subtasks which are also scheduled for execution (via the .fork() method of the subtasks. If the workLoad is below a certain threshold then the work is carried out by the MyRecursiveAction itself.

You can schedule a MyRecursiveAction for execution like this:

MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

forkJoinPool.invoke(myRecursiveAction);

## RecursiveTask

A RecursiveTask is a task that returns a result. It may split its work up into smaller tasks, and merge the result of these smaller tasks into a collective result. The splitting and merging may take place on several levels. Here is a RecursiveTask example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Long> {

private long workLoad = 0;

public MyRecursiveTask(long workLoad) {

this.workLoad = workLoad;

}

protected Long compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveTask> subtasks =new ArrayList<>();

subtasks.addAll(createSubtasks());

for(MyRecursiveTask subtask : subtasks){

subtask.fork();

}

long result = 0;

for(MyRecursiveTask subtask : subtasks) {

result += subtask.join();

}

return result;

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

return workLoad \* 3;

}

}

private List<MyRecursiveTask> createSubtasks() {

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);

MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is similar to the RecursiveAction example except it returns a result. The class MyRecursiveTask extends RecursiveTask<Long> which means that the result returned from the task is a Long .

The MyRecursiveTask example also breaks the work down into subtasks, and schedules these subtasks for execution using their fork() method.

ForkJoinPool Javadoc it states that the **number of threads must be a power of two**.

Additionally, this example then receives the result returned by each subtask by calling the join() method of each subtask. The subtask results are merged into a bigger result which is then returned. This kind of joining / merging of subtask results may occur recursively for several levels of recursion.

You can schedule a RecursiveTask like this:

MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);

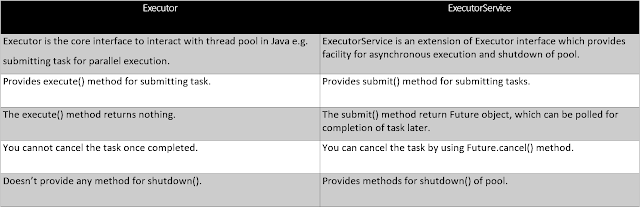
Notice how you get the final result out from the ForkJoinPool.invoke() method call.

**Executor, Executors and ExecutorService**

**Executor** is the core interface which is an abstraction for parallel execution. It separates task from execution.

**ExecutorService** is an extension of Executor interface and provides a facility for returning a Future object and terminate, or shut down the thread pool. Once the shutdown is called, the thread pool will not accept new task but complete any pending task. It also provides a [submit()](http://javarevisited.blogspot.sg/2016/04/difference-between-ExecutorServie-submit-vs-Executor-execute-method-in-Java.html) method which extends Executor.execute() method and returns a Future.

**Executors** is a utility class similar to Collections, which provides [factory methods](http://javarevisited.blogspot.com/2015/06/difference-between-dependency-injection.html) to create different types of thread pools e.g. fixed and cached thread pools.



**ThreadPool**

The most common of these pools are:

* newSingleThreadExecutor() Executors.newSingleThreadExecutor() creates a new Thread and executes all tasks passed to it on that thread. It doesn't block the current thread - it wouldn't be of much use if it did. It is called single thread because, if you execute multiple tasks on it, it does not create multiple threads. Instead, it waits for one task to complete before starting the next one on the same thread.
* **static ExecutorService newFixedThreadPool(int nThreads)**

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

* ***static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)***

Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

* ***static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)***

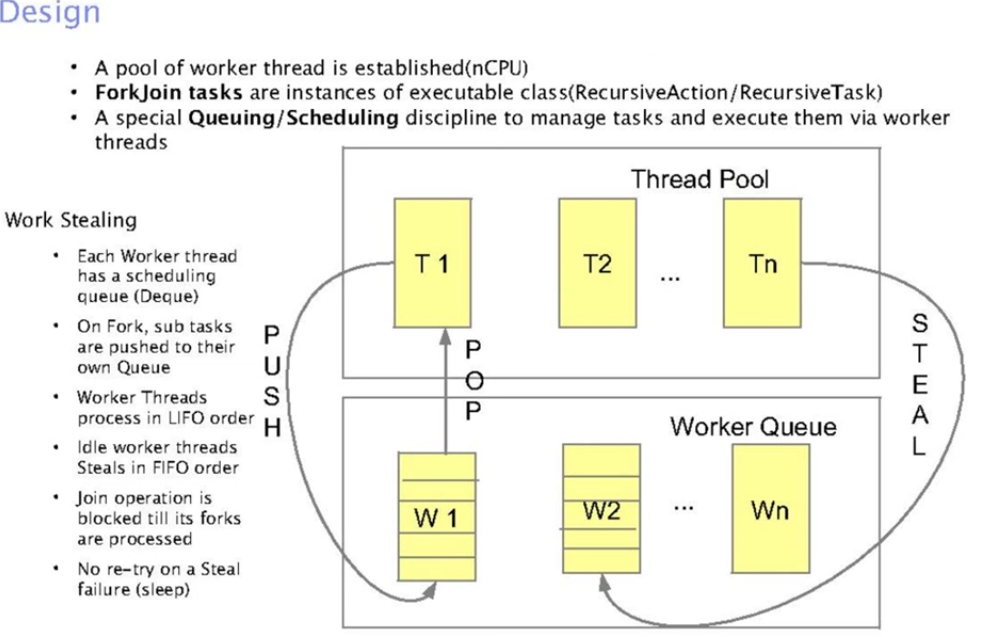
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available, and uses the provided ThreadFactory to create new threads when needed.

* ***static ExecutorService newWorkStealingPool(int parallelism)***

Creates a thread pool that maintains enough threads to support the given parallelism level, and may use multiple queues to reduce contention.

In Java 8, a new type of thread pool is introduced as **newWorkStealingPool()** to complement the existing ones. Java gave a very succinct definition of this pool as:

*“Creates a work-stealing thread pool using all available processors as its target parallelism level.”*

****

**ForkJoinPool** class implements ExecutorService. The task submitted to this pool should be subclass of ForkJoinTask.

Since this class is not extending ThreadPoolExecutor, it does not have all those core pool size, queuing, maximum pool size concepts, instead it implements work-stealing algorithm.

/\*With parallelism level i.e. how many cores to be used\*/

ForkJoinPool(int parallelism)

public static ExecutorService newWorkStealingPool() {

return new ForkJoinPool(Runtime.getRuntime().availableProcessors(),

ForkJoinPool.defaultForkJoinWorkerThreadFactory,

null, true);

}

**TransferQueue:**

**ThreadLocalRandom: random class for thread local**

**ConcurrentLinkedDeque:**

ConcurrentLinkedDeque is another concurrent collection which is part of the java.util.concurrent package. Unlike many other concurrent collections like ConcurrentHashMap, CopyOnWriteArrayList which were added in Java 5 ConcurrentLinkedDeque was added in Java 7.

ConcurrentLinkedDeque is an unbounded thread-safe Deque which stores its elements as linked nodes. Since it implements deque interface ConcurrentLinkedDeque supports element insertion and removal at both ends. You will find methods like addFirst(), addLast(), getFirst(), getLast(), removeFirst(), removeLast() to facilitate operations at both ends.

**Usage of ConcurrentLinkedDeque**

A ConcurrentLinkedDeque is an appropriate choice when many threads will share access to a common collection as concurrent insertion, removal, and access operations execute safely across multiple threads.

Note that it doesn't block operations as done in the implementation of BlockingDequeue interface like LinkedBlockingDeque. So there are no putFirst(), takeFirst() or putLast(), takeLast() methods which will wait if required.

Like most other concurrent collection implementations, this class does not permit the use of null elements.

Although both of these queues have certain similarities, there are substantial characteristics differences, too:

|  |  |  |
| --- | --- | --- |
| **Feature** | **LinkedBlockingQueue** | **ConcurrentLinkedQueue** |
| **Blocking Nature** | It is a blocking queue and implements the *BlockingQueue* interface | It is a non-blocking queue and does not implement the *BlockingQueue*interface |
| **Queue Size** | It is an optionally bounded queue, which means there are provisions to define the queue size during creation | It is an unbounded queue, and there is no provision to specify the queue size during creation |
| **Locking Nature** | It is**a lock-based queue** | It is **a lock-free queue** |
| **Algorithm** | It implements**its locking based on *two-lock queue* algorithm** | It relies on the **Michael & Scott algorithm for non-blocking, lock-free queues** |
| **Implementation** | In the *two-lock queue* algorithm mechanism, *LinkedBlockingQueue*uses two different locks – the *putLock* and the *takeLock*. The *put/take* operations uses the first lock type, and the *take/poll* operations use the other lock type | **It uses CAS (Compare-And-Swap**) for its operations |
| **Blocking Behavior** | It is a blocking queue. So, it blocks the accessing threads when the queue is empty | It does not block the accessing thread when the queue is empty and returns *null* |

RecursiveTask returns a value, RecursiveAction doesn't. It's like Task vs Runnable

**CompletableFuture**

**CompletableFuture** is one of the **cool features promoting an asynchronous**, event-driven programming model.

**Future Vs CompletableFutue**

**Multiple Futures cannot be chained together**

**No Exception Handling**

**You can’t combine multiple Futures together.**

**You cannot perform further action on a Future’s result without blocking.**

**What is Web Crawling?**

Web crawler is a program that acts as an automated script which browses through the internet in a systematic way. The web crawler looks at the keywords in the pages, the kind of content each page has and the links, before returning the information to the search engine. This process is known as Web crawling.

**jsoup** is a Java based library to work with HTML based content.

*//get useful information*

Document doc = Jsoup.connect("<http://www.mit.edu/>").get();

**if**(doc.text().contains("research")){

System.out.println(URL);

}

*//get all links and recursively call the processPage method*

Elements questions = doc.select("a[href]");

**java.util.Scanner** class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters

**What is Guava?**

Guava is an open source, Java-based library and contains many core libraries of Google, which are being used in many of their projects. It facilitates best coding practices and helps reduce coding errors. It provides utility methods for collections, caching, primitives support, concurrency, common annotations, string processing, I/O, and validations.

**map and flatmap difference**

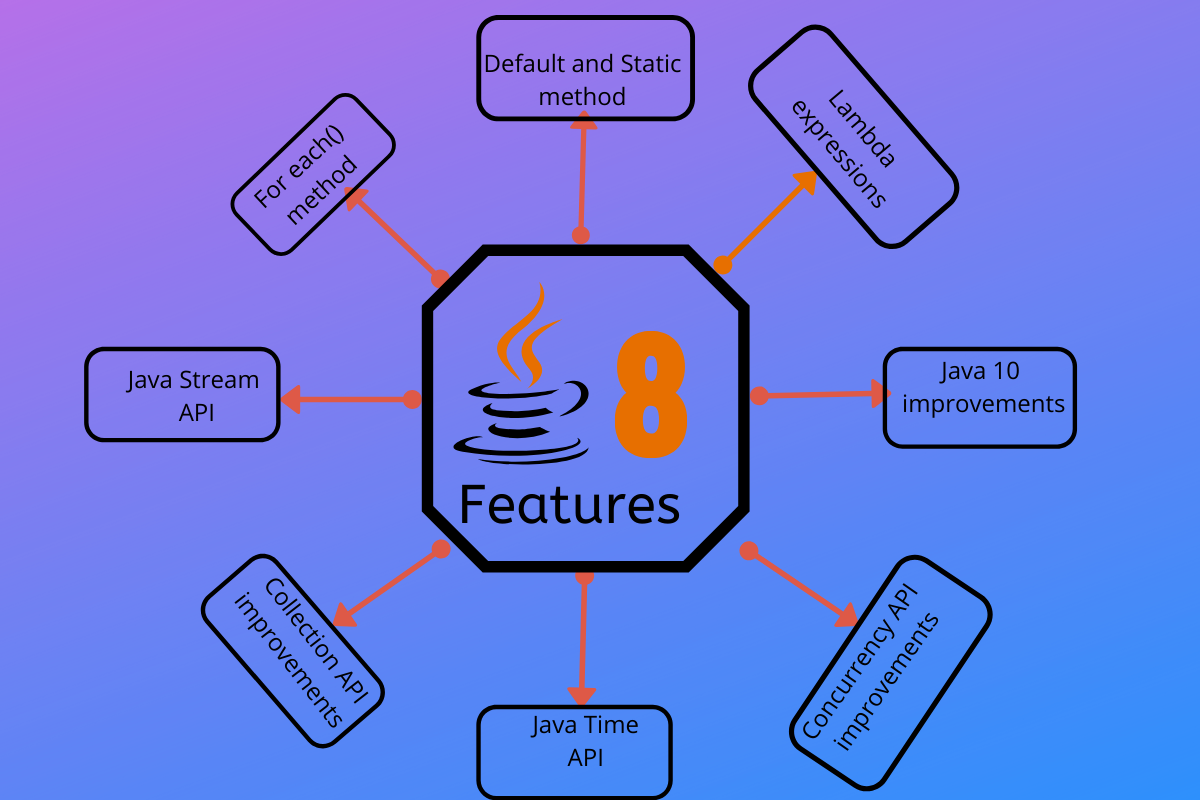
map: Stream T -> (T -> R) -> Stream R

flatmap: Stream T -> (T -> Stream R) -> Stream R

**Java Stream:**

Streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

**Java 8: (Features)**



The Stream API enables developers to create the parallel streams that can take advantage of multi-core architectures and enhance the performance of Java code. In a parallel stream, the operations are executed in parallel and there are two ways to create a parallel stream.

* Using the parallelStream() method on a collection
* Using the parallel() method on a stream

Optional<Integer> calcProd = list1.parallelStream().reduce((a,b) -> a\*b));

* A stateless operation is an operation in which the state of one element does not affect another element
* A non-interfering operation is an operation in which data source is not affected
* An associative operation is an operation in which the result is not affected by the order of operands

### When to use Parallel Streams?

* They should be used when the output of the operation is not needed to be dependent on the order of elements present in source collection (i.e. on which the stream is created)
* Parallel Streams can be used in case of aggregate functions
* Parallel Streams quickly iterate over the large-sized collections
* Parallel Streams can be used if developers have performance implications with the Sequential Streams
* If the environment is not multi-threaded, then Parallel Stream creates thread and can affect the new requests coming in

**Java 8 Streams is not Java I/O streams. Both are separate.**

**A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**

**private static Employee[] empArray= {**

**new Employee(1, "Jeff Bezos", 100000.0),**

**new Employee(2, "Bill Gates", 200000.0),**

**new Employee(3, "Mark Zuckerberg", 300000.0)**

**};**

**Stream.of(empArray);**

**private static List<Employee> empList = Arrays.asList(arrayOfEmps);**

**empList.stream();**

**OR**

**Stream.Builder<Employee> empStreamBuilder = Stream.builder();**

**empStreamBuilder.accept(arrayOfEmps[0]);**

**empStreamBuilder.accept(arrayOfEmps[1]);**

**empStreamBuilder.accept(arrayOfEmps[2]);**

**Stream<Employee> empStream = empStreamBuilder.build();**

**Stream.of(empArray [0], empArray [1], empArray [2]);**

#### **map**

map() produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

<R> Stream<R> map(Function<? super T, ? extends R> mapper);

**collect()** performs mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in the Stream instance.

#### **flatMap**

A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations:

flattens the List<List<String>> to a stream of List<String>

**Peek()**

**Stream<T> peek(Consumer<? super T> action);**

**Stream.peek() is an intermediate operation**, i.e. it does not end the processing of the stream.

Stream.peek() thus provides you the unique capability to consume a stream without ending the pipeline of operations when acting on the stream contents, by virtue of it being an intermediate operation. In addition, **Stream.peek() is a non-interfering Stream operation**. Non-interfering methods are those which guarantee that they will not modify the Stream’s data source during their execution. Non-interfering nature is required in multi-threaded environments where stream operations can be executed in parallel or concurrently. Concurrent execution makes it necessary to keep the stream’s data source unmodified until the terminal operation.

Some operations are deemed **short-circuiting operations**. Short-circuiting operations allow computations on infinite streams to complete in finite time:

**Skip**

skip() to skip first 3 elements.

List<Integer> collect = infiniteStream

.skip(3)

.limit(5)

.collect(Collectors.*toList*());

### Lazy Evaluation

One of the most important characteristics of streams is that they allow for significant optimizations through lazy evaluations.

all the operations are evaluated. Processing streams lazily allows avoiding examining all the data when that’s not necessary. This behavior becomes even more important when the input stream is infinite and not just very large.

### Comparison based stream

**sorted()**

#### min and max

#### **Collector method**

#### summarizingDouble

summaryStatistics

DoubleSummaryStatistics stats = empList.stream()

.mapToDouble(Employee::getSalary)

.summaryStatistics()

#### groupingBy

groupingBy() offers advanced partitioning – where we can partition the stream into more than just two groups

Map<Character, List<Employee>> groupByAlphabet = empList.stream().collect(

Collectors.groupingBy(e -> new Character(e.getName().charAt(0))));

#### partitioningBy

We can partition a stream into two – based on whether the elements satisfy certain criteria or not.

Map<Boolean, List<Integer>> isEven = intList.stream().collect(

Collectors.partitioningBy(i -> i % 2 == 0));

Partitioning is a special kind of grouping, in which the resultant map contains at most two different groups – one for true and one for false.

mapping()

Map<Character, List<Integer>> idGroupedByAlphabet = empList.stream().collect(

Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),

Collectors.mapping(Employee::getId, Collectors.toList())));

Employee emp1 = **new** Employee(1,"neeraj",40.0,1);//empid,empname,empsal,dept

Employee emp2 = **new** Employee(2,"sachan",60.0,1);

Employee emp3 = **new** Employee(3,"naveen",30.0,2);

**Sum of Emp salary Group by dept. 🡪**

Map<Integer,Double> empMap= empList.stream().collect(Collectors.groupingBy(Employee::getEmpDept,Collectors.summingDouble(Employee::getEmpSalary)));

# BiFunction

java.util.function.BiFunction is a functional interface whose functional method is R apply(T t, U u). The BiFunction is interface represents an operation that takes two arguments (T and U) and returns a result R.

BiFunction<Integer, Integer, Integer> function1 = (a, b) -> a + b;

Function<Integer, Integer> function2 = (n) -> n\*n;

BiFunction<T,T,T> implies binaryOperator

**Class loader:**

It is mainly responsible for three activities.

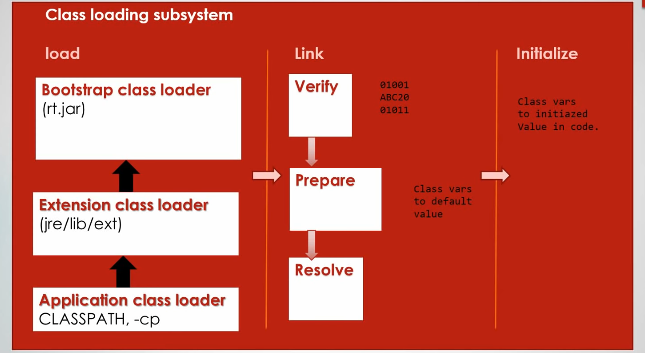
1. Loading: The Class loader reads the “.class” file, generate the corresponding binary data and save it in the method area.
2. Linking:

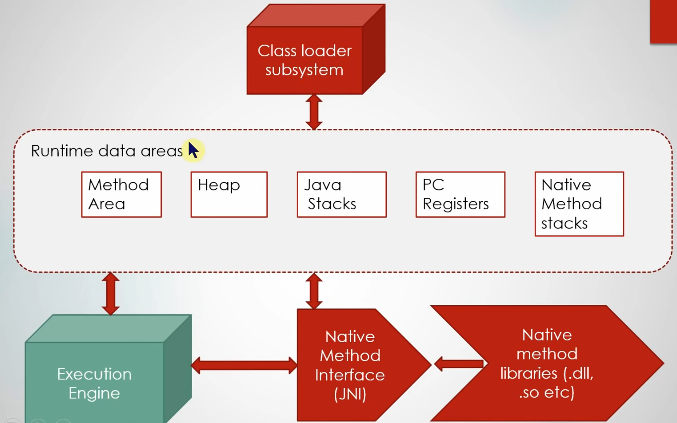
**Verification**: It ensures the correctness of the .class file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.

**Preparation**: JVM allocates memory for class static variables and initializing the memory to default values.

**Resolution**: It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

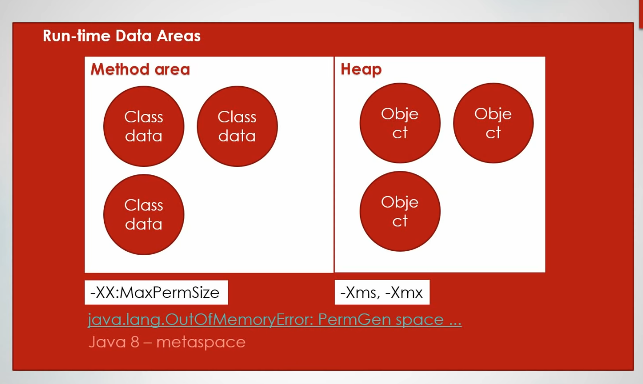
1. Initialization

****

****

**JVM Memory**

1. **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.
3. **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
4. **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
5. **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.

****

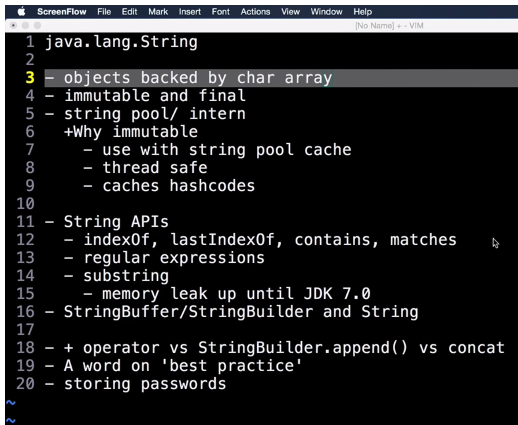
**Execution Engine**

Execution engine executes the “.class” (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

**Interpreter**: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.

**Just-In-Time Compiler(JIT)** : It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

**Garbage Collector**: It destroys un-referenced objects. For more on Garbage Collector, refer Garbage Collector.

****

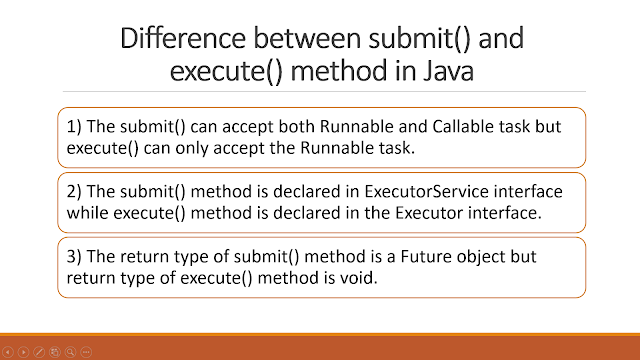
**The Future interface** was added in Java 5 to serve as a result of an asynchronous computation, but it did **not have any methods to combine these computations or handle possible errors.**

In Java 8, the **CompletableFuture** class was introduced. Along with the Future interface, it also implemented the **CompletionStage** interface. This interface defines the contract for an asynchronous computation step that can be combined with other steps. CompletableFuture is at the same time a building block and a framework with about 50 different methods for composing, combining, executing asynchronous computation steps and handling errors.

How does substring () inside String works? ([answer](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html))

Another good Java interview question, I think the answer is not sufficient, but here it is “Substring creates a new object out of source string by taking a portion of original string”. This question was mainly asked to see if the developer is familiar with the risk of memory leak, which sub-string can create. Until Java 1.7, **substring holds the reference of the original character array, which means even a sub-string of 5 character long, can prevent 1GB character array from garbage collection, by holding a strong reference.**

This issue is fixed in Java 1.7, where original character array is not referenced anymore, but that change also made the creation of substring bit costly in terms of time. Earlier it was in the range of O(1), which could be O(n) in worst case on Java 7.

[](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html)

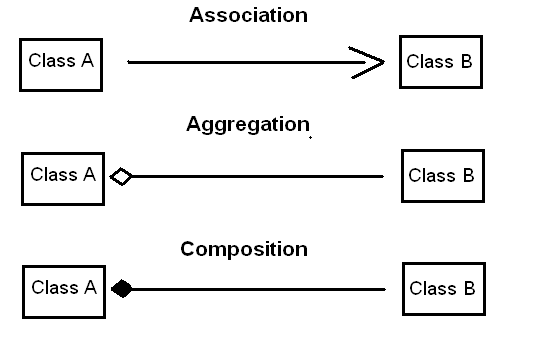
There is a difference when looking at exception handling. If your tasks throw an exception and **if it was submitted with execute() then the exception will go to the uncaught exception handler** (when you don't have provided one explicitly, the default one will just print the stack trace to System.err).

**If you submitted the task with submit()** than any thrown exception, checked exception or not, is then part of the task's return status. For a task that was submitted with submitting and that terminates with an exception, the Future.get() will re-throw this exception, wrapped in an ExecutionException.

**Can volatile make a non-atomic operation to atomic?**  
This another good question I love to ask on volatile, mostly as a follow-up of the previous question. This question is also not easy to answer because volatile is not about atomicity, but there are cases where you can use a volatile variable to make the operation atomic.  
  
One example I have seen is having a long field in your class. If you know that a long field is accessed by more than one thread e.g. a counter, a price field or anything, you better make it volatile. Why? because reading to a long variable is not atomic in Java and done in two steps, If one thread is writing or updating long value, it's possible for another thread to see half value (fist 32-bit). While reading/writing a volatile long or double (64 bit) is atomic.

**What guarantee volatile variable provides?** ([answer](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html))  
volatile variables provide the guarantee about ordering and visibility e.g. volatile assignment cannot be re-ordered with other statements but in the absence of any synchronization instruction compiler, JVM or JIT are free to reorder statements for better performance. volatile also provides the happens-before guarantee which ensures changes made in one thread is visible to others. In some cases volatile also provide atomicity e.g. reading 64-bit data types like long and double are not atomic but read of volatile double or long is atomic.

**Difference between Composition, Aggregation and Association in OOP?** ([answer](http://javarevisited.blogspot.sg/2014/02/ifference-between-association-vs-composition-vs-aggregation.html))  
If two objects are related to each other, they are said to be associated with each other. **Composition and Aggregation are two forms of association in object-oriented programming**. The composition is stronger association than Aggregation. In Composition, one object is OWNER of another object while in Aggregation one object is just USER of another object. If an object A is composed of object B then B doesn't exist if A ceased to exists, but if object A is just an aggregation of object B then B can exists even if A ceased to exist.



**Association**-> It represents a **HAS-A** relationship. It has **two** types.

**Aggregation**: Now, Aggregation in Java is a special type of association. Java Aggregation has the following characteristics –

1. A has-A relationship is represented here.
2. It is a one-way relationship, i.e. a unidirectional relationship.
3. Ending one entity won’t affect another, both can be present independently.

Students and college

**Composition**

This is a restricted form of Java aggregation that is the quantities are highly dependent on each other. Java composition represents a part-of relationship. One entity cannot exist without the other.

Example: book and author.

**The difference between Inheritance and Composition?** ([answer](http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html))  
Though both allows code reuse, Composition is more flexible than Inheritance because it allows you to switch to another implementation at run-time. Code written using Composition is also easier to test than code involving inheritance hierarchies.

**Which one is better constructor injection or setter dependency injection?** ([answer](http://javarevisited.blogspot.sg/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html))  
Each has their own advantage and disadvantage. Constructor injection guaranteed that class will be initialized with all its dependency, but setter injection provides flexibility to set an optional dependency. Setter injection is also more readable if you are using an XML file to describe dependency. Rule of thumb is to use constructor injection for mandatory dependency and use setter injection for optional dependency.

**A difference between WeakReference and SoftReference in Java?** ([answer](http://javarevisited.blogspot.sg/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html))  
Though both WeakReference and SoftReference helps garbage collector and memory efficient, WeakReference becomes eligible for garbage collection as soon as last strong reference is lost but SoftReference even though it cannot prevent GC, it can delay it until JVM absolutely need memory.

**Memory management** is done automatically in Java. The programmer doesn't need to worry about reference objects that have been released. One downside to this approach is that the programmer cannot know when a particular object will be collected. Moreover, the programmer has no control over memory management. However, the **java.lang.ref package** defines classes that provide a limited degree of interaction with the garbage collector. The concrete classes SoftReference, WeakReference and PhantomReference are subclasses of Reference that interact with the garbage collector in different ways.

**Reference Types**

we have different types of references: **strong, weak, soft, and phantom references**.

The difference between the types of references is that the objects on the heap they refer to are eligible for garbage collecting under the different criteria.

**Strong Reference:** StringBuilder builder = new StringBuilder();

**WeakReference :** WeakReference<StringBuilder> reference = new WeakReference<>(new StringBuilder());

A nice use case for weak references are caching scenarios. Imagine that you retrieve some data, and you want it to be stored in memory as well — the same data could be requested again. On the other hand, you are not sure when, or if, this data will be requested again. A nice implementation for caching scenarios is the collection WeakHashMap<K,V>. If we open the WeakHashMap class in the Java API, we see that its entries actually extend the WeakReference class and uses its ref field as the map’s key:

private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> {

V value;

Once a key from the WeakHashMap is garbage collected, the entire entry is removed from the map.

**Soft Reference**

These types of references are used for more memory-sensitive scenarios

since those are going to be garbage collected only when your application is running low on memory.

Java guarantees that all soft referenced objects are cleaned up before it throws an OutOfMemoryError.

SoftReference<StringBuilder> reference = new SoftReference<>(new StringBuilder());

**Phantom Reference**

Phantom Reference can be used in situations, where sometime using finalize() is not  sensible thing to do. This reference type differs from the other types defined in [java.lang.ref](http://java.sun.com/javase/6/docs/api/java/lang/ref/package-summary.html) Package because it isn't meant to be used to access the object, but as a signal that the object has already been finalized, and the garbage collector is ready to reclaim its memory.

These types of references are considered preferable to finalizers.

**We can’t get a referent of a phantom reference.** The referent is never accessible directly through the API and this is why we need a reference queue to work with this type of references.

The Garbage Collector adds a phantom reference to a reference queue **after the finalize method of its referent is executed**. It implies that the instance is still in the memory.

People usually attempt to use finalize() method to perform postmortem cleanup on objects which usually not advisable. As mentioned earlier, Finalizers have an impact on the performance of the garbage collector since Objects with finalizers are slow to garbage collect.

Phantom references are safe way to know an object has been removed from memory. For instance, consider an application that deals with large images. Suppose that we want to load a big image in to memory when large image is already in memory which is ready for garbage collected. In such case, we want to wait until the old image is collected before loading a new one. Here, the phantom reference is flexible and safely option to choose. The reference of the old image will be enqueued in the ReferenceQueue once the old image object is finalized. After receiving that reference, we can load the new image in to memory. Similarly, we can use Phantom References to implement a Connection Pool. We can easily gain control over the number of open connections, and can block until one becomes available.

**JVM Parameter**

* Initial heap size -Xms512m – set the initial heap size to 512 megabytes.
* Maximum heap size -Xmx1024m – set the maximum heap size to 1024 megabytes.
* Thread stack size -Xss128m – set the thread stack size to 128 megabytes.
* Young generation size -Xmn256m – set the young generation size to 256 megabytes.

 If a Java application crashes with an OutOfMemoryError and you need some extra info to detect the leak, run the process with the –XX:HeapDumpOnOutOfMemory parameter, which will create a heap dump file when this error happens next time.

 Use the -verbose:gc option to get the garbage collection output. Each time a garbage collection takes place, an output will be generated.

**Code Cache:** JVM has an interpreter to interpret the byte code and convert it into hardware dependent machine code. As part of JVM optimization, the Just In Time (JIT) compiler has been introduced. The frequently accessed code blocks will be compiled to native code by the JIT and stored it in code cache. The JIT compiled code will not be interpreted.

**Meta Space:** This memory is out of heap memory and part of the native memory. As per the document by default the meta space doesn’t have an upper limit. In earlier versions of Java we called this “***Perm Gen Space***". This space is used to store the class definitions loaded by class loaders. This is designed to grow in order to avoid 0ut of memory errors. However, if it grows more than the available physical memory, then the operating system will use virtual memory. This will have an adverse effect on application performance, as swapping the data from virtual memory to physical memory and vice versa is a costly operation. We have JVM options to limit the Meta Space used by the JVM. In that case, we may get out of memory errors.

**Programming paradigm:**

* [**Imperative programming**](https://en.wikipedia.org/wiki/Imperative_programming) – defines [computation](https://en.wikipedia.org/wiki/Computation) as [statements](https://en.wikipedia.org/wiki/Statement_(programming)) that change a program [state](https://en.wikipedia.org/wiki/State_(computer_science)).(Java)
* [**Procedural (structured) programming**](https://en.wikipedia.org/wiki/Procedural_programming)**,**– specifies the steps a program must take to reach a desired state.(C)
* [**Declarative programming**](https://en.wikipedia.org/wiki/Declarative_programming) – defines program logic, but not detailed [control flow](https://en.wikipedia.org/wiki/Control_flow).(SQL)
* [**Functional programming**](https://en.wikipedia.org/wiki/Functional_programming) – treats programs as evaluating [mathematical functions](https://en.wikipedia.org/wiki/Function_(mathematics)) and avoids [state](https://en.wikipedia.org/wiki/Program_state) and [mutable](https://en.wikipedia.org/wiki/Immutable_object) data.
* [**Object-oriented programming**](https://en.wikipedia.org/wiki/Object-oriented_programming) **(OOP)** – organizes programs as [objects](https://en.wikipedia.org/wiki/Object_(computer_science)): [data structures](https://en.wikipedia.org/wiki/Data_structure) consisting of [data fields](https://en.wikipedia.org/wiki/Field_(computer_science)) and [methods](https://en.wikipedia.org/wiki/Method_(computer_science)) together with their interactions.
* [**Event-driven programming**](https://en.wikipedia.org/wiki/Event-driven_programming) – program [control flow](https://en.wikipedia.org/wiki/Control_flow) is determined by [events](https://en.wikipedia.org/wiki/Event_(computing)), such as [sensor](https://en.wikipedia.org/wiki/Sensor) inputs or user actions ([mouse](https://en.wikipedia.org/wiki/Computer_mouse) clicks, key presses) or [messages](https://en.wikipedia.org/wiki/Message_passing) from other programs or [threads](https://en.wikipedia.org/wiki/Thread_(computer_science)).
* [**Automata-based programming**](https://en.wikipedia.org/wiki/Automata-based_programming)– a program, or part, is treated as a model of a finite state machine or any other formal automaton.

**Declarative programming** is a programming paradigm … that expresses the logic of a computation without describing its control flow.

**Imperative programming** is a programming paradigm that uses statements that change a program’s state.

**Functional Programming**

Functional programming is a programming paradigm in which we **try to bind everything in pure mathematical functions style**. It is a declarative type of programming style. Its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”. It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables. Those functions have some special features discussed below.

**Synchronous vs Asynchronous**

When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

**synchronous / asynchronous is to describe the relation between two modules.**

**blocking / non-blocking is to describe the situation of one module.**

Predicate<T>–Determine if the input of type T matches some criteria

Consumer<T>–Accept a single input argument of type T, and return no result

Function<T, R>–Apply a function to the input type T, generating a result of type R

Supplier<T>–A supplier of results

**ThreadGroup**

Java thread group is implemented by java.lang.ThreadGroup class. A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent. A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

1. ThreadGroup tg1 = new ThreadGroup("Group A");
2. Thread t1 = new Thread(tg1,new MyRunnable(),"one");
3. Thread t2 = new Thread(tg1,new MyRunnable(),"two");
4. Thread t3 = new Thread(tg1,new MyRunnable(),"three");

Now we can interrupt all threads by a single line of code only.

1. Thread.currentThread().getThreadGroup().interrupt();

# Shutdown Hook

the shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. So if you want to execute some code before JVM shuts down, use shutdown hook.

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine. Syntax:

1. public void addShutdownHook(Thread hook){}
2. class MyThread extends Thread{
3. public void run(){
4. System.out.println("shut down hook task completed..");
5. }
6. }
8. public class TestShutdown1{
9. public static void main(String[] args)throws Exception {
11. Runtime r=Runtime.getRuntime();
12. r.addShutdownHook(new MyThread());
14. System.out.println("Now main sleeping... press ctrl+c to exit");
15. try{Thread.sleep(3000);}catch (Exception e) {}
16. }
17. }