Core Java

-AMRUTA NADGONDE

Chapter 7

FUNCTIONAL PROGRAMMING AND STREAMS

Lambda Expression

- Streams and Lambdas are two key factors in functional programming
- Functional programming helps in creating certain programs that are simpler, faster and have fewer bugs
- Such programs are easier to parallelize providing user the advantage of multi-core architecture to enhance performance
- With lambda Java introduced a new syntax and an operator -> lambda operator
- Lambda is essentially an anonymous method, where left hand side is parameter list and right hand side looks like method body [single statement or block of code]
- Lambda expressions allow us to create methods that can be treated as data

Functional Interface

- A functional interface is an interface that contains exactly one abstract method [may contain default/static methods] , also called as Single Abstract Method[SAM] interface
- Functional interface contains a single abstract methods, which defines the function of the interface
- A function is said to be a pure function if it –
  - Depends only on its parameters
  - Has no side effects
  - Do not maintain any state
- In java, methods that implement functional interfaces are pure functions– typically defined as lambdas

Functional Interface and lambdas

- Lambda expressions can be used anywhere functional interfaces are expected
- Lambda can be specified only in the context where a Target type is defined
- The java compiler can infer the types of lambda parameters and the type returned by the lambda from the context in which lambda is used
- This is determined by the lambda's **target-type –** i.e. type of functional interface type that the lambda implements

```
NumTest ispositive = (n) -> n >= 0 ;
```

Here compiler determines, the target type is interface NumTest. Lambda is assigned to a variable that can later be used to invoke abstract method of NumTest

Passing Lambda as argument

- Lambda can be used in any context that provides a target type
- One of the context is when lambda is passed as an argument to another method.
- This is most common use of lambda as it gives us a way to pass executable code as an argument to a method
- The methods that accept lambdas should have parameter type as functional interface. This type should be compatible with the lambda expression passed

```
map((x) -> x*2)
```

Here lambda is passed as a argument to map method. The parameter type of map method is IntUnaryOperator. The applyAsInt method of IntUnaryOperator accepts an int and return an int which is compatible with the lambda expression

Method Reference

- Method reference : provides a way to refer to a method without executing it.
- It is Java's way of sending a method reference inside another method
- The method that accepts another method as reference must have functional interface as parameter
- The lambda expression is written in place of actual method code. The lambda must be compatible with function interface
- To create a static method reference

```
Classname :: method_name
```

- To create an instance method reference

```
objRef::method_name
```

Imperative vs Declarative Programming

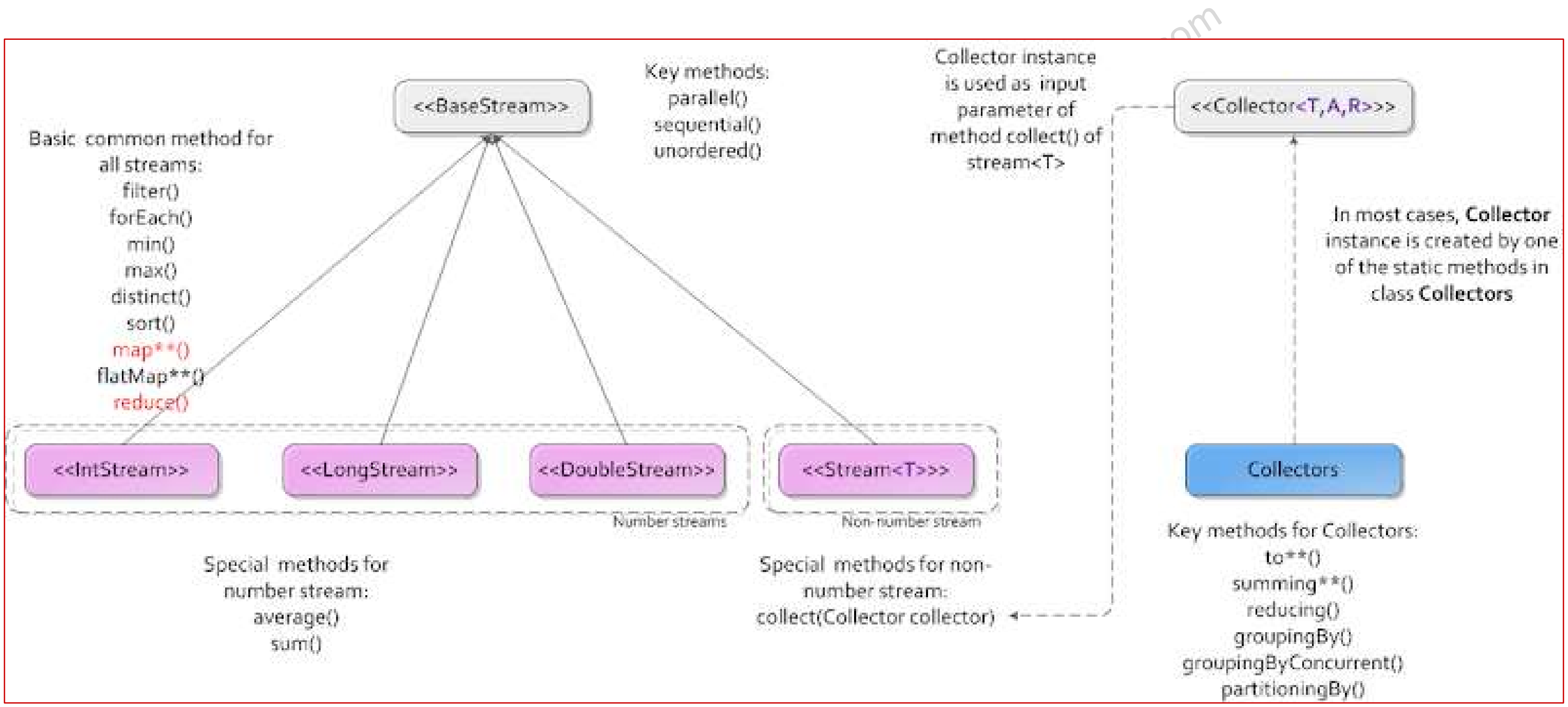
- Counter-controlled iteration : Operations where we typically specify 'what' do we want to accomplish and 'how' to accomplish the task using a loop (e.g. for)
- Task is performed in each iteration of the loop – external iteration
- Such loops are error prone leading to errors related to initialization , increment and bounds
- Imperative Programming : where we specify what do we want to achieve followed by how the task can be achieved
- Declarative Programming : We specify 'what' we want the task to accomplish (rather than how)
- Internal iteration – Iteration over the elements without specifying 'how' to iterate or declaring any mutable variable

Streams

- Introduced in Java 8, in java.util.stream package
- Stream : sequence of elements on which we perform a specific task
- A stream operates on data source such as an array or collection
- Stream classes support declarative programming – specify what to do rather than how to do it
- Stream pipelines - Chained method calls on streams create a stream pipeline
- Stream pipelines start with a method call that creates a stream [e.g. range()]
- We can perform many intermediate operations on the stream before a terminal operation is performed
- Terminal operation terminates the stream and produces final result

Stream API

- The stream API defines several stream interfaces, packaged in java.util.streams
- BaseStream interface is at the Base of all streams and defines basic functionalities for all streams
- Several types of streams are derived from the base stream like – most generic is Stream
- Stream is a stream of reference types [objects]
- Stream API also provides streams to operate on primitive data – IntStream, LongStrem and DoubleStream
- Collections are used in java as data structures to store objects, streams on the other hand are used to operate on the data stored in the collections or arrays


Basic common method for all streams:
filter()
forEach()
min()
max()
distinct()
sort()
map**()
flatMap**()
reduce()
<<BaseStream>>
Key methods:
parallel()
sequential()
unordered()
Collector instance is used as input parameter of method collect() of stream<T>
<<Collector<T,A,R>>>
In most cases, **Collector** instance is created by one of the static methods in class **Collectors**
<<IntStream>>
<<LongStream>>
<<DoubleStream>>
<<Stream<T>>>
Collectors
Number streams
Non-number stream
Special methods for number stream:
average()
sum()
Special methods for non-number stream:
collect(Collector collector)
Key methods for Collectors:
to**()
summing**()
reducing()
groupingBy()
groupingByConcurrent()
partitioningBy()

Obtaining a Stream

- Stream can be obtained for a collection as well as for an array
- Two new methods are added Collection interface which can obtain a stream from the collection [JDK 8]

```
collection_obj.stream()
```

- This obtains a sequential stream. A parallel stream can be obtained using parallelStream method

```
collection_obj.parallelStream()
```

- Streams can also be obtained for arrays using static method stream()

```
Arrays.stream(array_obj)
```

Collecting

- It is common to obtain a stream from collection, but it is equally desirable to obtain a collection from the stream
- This can be viewed as the terminal operation
- To perform this stream provides collect() method

    stream.collect(Collector<?super T,A,R > collectFunc)

- The Collector interface provides several method to perform the collection function
- Collectors [implements Collector] class methods toList() and toSet() are most commonly used

Performing Stream Operations

- Intermediate operations use lazy evaluation
- Lazy evaluation : each intermediate operation results into a new stream object but does not preform any operation on it immediately until terminal operation is performed to produce a result
- Terminal operations use eager evaluation
- Eager evaluation : Terminal operations perform the requested operation when they are called
- This enhances the performance to a great extend
- Also some intermediate operations are stateless while other are stateful
- In stateless operation each element is process independently of other, in stateful operation such processing might depend on other element values

Typical Operations

Intermidiate Operations	Terminal Operations
Filter	forEach
Map	reduce
Distinct	count
limit	min
sorted	max
	collect

Functional Interfaces

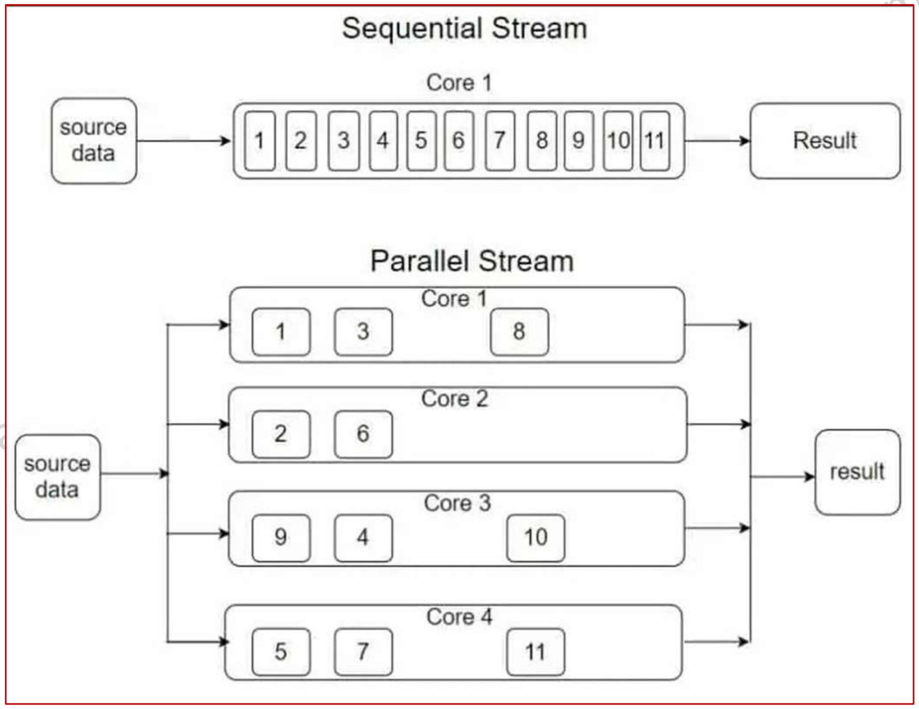
- java.util.functions contain several functional interfaces. Few of the basic ones are explained below:

Interface	Description
Binary Operator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands. E.g. the lambda passed to reduce method implements this interface
Consumer<T>	Represents an operation that accepts a single input argument and returns void. E.g. the lambda passed to forEach method implements this interface
Function <T, R>	Represents a function that accepts one argument and produces a result. E.g. the lambda passed to map method implements this interface
Predicate <T>	Represents a one parameter method that returns boolean value.E.g. the lambda passed to filter method implements this interface
Supplier<T>	Represents a supplier of results - a one parameter method that returns a result. Often used to create a collection object in which a stream operation's result are passed

Optional Class & Static Methods

- Various mathematical operations can be performed on streams – min, max, count
- The operations like min() and max() take Compotator as their argument.
- A lambda can be written as method reference to call compare method of Comparator
- Return type of these methods is 'Optional' instance which may contain a value of type T or may be empty
- The streams of primitives also support mathematical operations like min, max, average, count
- The streams of primitives obtain a stream using static method 'of' of respective streams
- These streams also have a method boxed() which returns stream consisting of the elements of this stream, each boxed to an Wrapper type

Parallel Stream



Chapter 8

JAVA DATETIME

Introduction

- Working with date and time is one of the challenges in programming language
- Java DateTime API helps to deal effectively with time zones, daylight saving time, and different written date formats
- The old Java Date API was not thread-safe, It depended on java.text classes to format the date
- The Objects of new DateTime API classes are immutable [hence thread safe]
- Computers count time from an instant called the Unix epoch [January 1, 1970, at 00:00:00 UTC]
- An ‘Instant’ is a point on the timeline
- Duration is difference between two instants

LocalDate Class

- Gives date and/or time of a day without any time zone information

  ```
  LocalDate today = LocalDate.now()
  LocalDate adate = LocalDate.of(2023, Month.AUGUST, 10)
  ```

- The difference between two dates is Period
- Using plus/minus methods we can add/subtract period
- plusDays(), plusWeeks(), plusMonths(), plusYears() methods can be used to add specific value to the date

  ```
  LocalDate nextweek = today.plusWeeks(1)
  ```

Method	Description
now(), of()	Construct a LocalDate from the current time/ from given year,month and day
plusDays(),plusWeeks(), plusMonths(), plusYears()	Adds a number of days, weeks, months or years to this LocalDate
minusDays(),minusWeeks() ,minusMonths(), minusYears()	Subtracts a number of days, weeks, months or years from this LocalDate
plus(), minus()	Adds or substrats a Duration of Period
getDayOfMonth()	Gets the day-of-month field
getDayOfWeek()	Gets the day-of-week field, which is an enum DayOfWeek.
getDayOfYear()	Gets the day-of-year field.
getMonth()	Gets the month-of-year field using the Month enum.
getYear()	Gets the year field.
unitl()	Gets the period, or the number of the given Chronounits, between the two dates
isBefore, isAfter	Compares the LocalDate with another

LocalTime Class

- A LocalTime represents a time of the day
- A time Instance can be created using methods now or of

```
LocalTime tcurrent = LocalTime.now();
LocalTime later = LocalTime.of(22, 00);
```

- Interval can be given using Duration

```
LocalTime plusone = wakeuptime.plus(Duration.ofHours(1))
```

- getHour, getMinute, getSecond can be used to retrieve time values

Method	Description
now(), of()	Construct a LocalTime from the current time/ from given hours, minutes and seconds
plusHours(),plusMinutes(), plusSeconds(), plusNanos()	Adds a number of hours, minutes, seconds or nanoseconds to this LocalTime
minusHours(),minusMinutes(), minusSeconds(), minusNanos()	Subtracts a number of hours, minutes, seconds or nanoseconds to this LocalTime
plus(), minus()	Adds or substrats a Duration of Period
getHour()	Gets the Hour
getMinute()	Gets the Minutes
getSeonds()	Gets the Seconds
getNano()	Gets the Nano Seconds
isBefore, isAfter	Compares the Local Time with another
toEpochSecond	Given a Local Date and Zone Offset, yields the number of seconds from the epoch to the specified point in time

Zoned Time

- Zone in which a date occurs is an important aspect of ensuring your code is correct
- IANA – Internet Assigned Number Authority keeps database of all known time zones around the world
- Each time zone has an ID, which can be found out by ZoneId.getAvailableIds()
- Using a static method ZoneId.of("Europe/Berlin") one can get a ZoneId object, which can be used to convert a LocalDate to ZonedDateTime

```
ZonedDateTime now = ZonedDateTime.of(LocalDate.now(),
                                     LocalTime.now(), ZoneId.of("Europe/Berlin"));
```

Method	Description
now(), of()	Construct a LocalDate from the current time/ from given year,month and day
plusDays(),plusWeeks(), plusMonths(), plusYears()	Adds a number of days, weeks, months or years to this LocalDate
minusDays(),minusWeeks(), minusMonths(), minusYears()	Subtracts a number of days, weeks, months or years to this LocalDate
plus(), minus()	Adds or substrats a Duration of Period
getDayOfMonth()	Gets the day-of-month field
getDayOfWeek()	Gets the day-of-week field, which is an enum DayOfWeek.
getDayOfYear()	Gets the day-of-year field.
getMonth()	Gets the month-of-year field using the Month enum.
getYear()	Gets the year field.
unitl()	Gets the period, or the number of the given Chronounits, between the two dates
isBefore, isAfter	Compares the LocalDate with another

Formatting and Parsing

- The DateTimeFormatter class provides three kinds of formatters to print a date/time value
  - Predefined standard formatters
  - Local-specific formatters
  - Formatters with custom patterns
- The format() method can be used to give standard formatter
- The standard formatters are mostly intended for machine readable timestamps, For human readable dates locale specific formatters are used
- Custom patterns can be given using a user defined pattern with the help of ofPattern() method

Symbol	Meaning	Example	Symbol	Meaning	Example
G	Era Designation	AD	S	Millisecond	968
Y	Year	2022	E	Day in week	Monday
M	Month In A Year	March; Mar; 03	D	Day in year	180
d	Day in a Month	5	w	Week in year	25
h	hour in am/pm (1-12)	10	W	Week in month	3
H	hour in day (0-23)	March; Mar; 03	a	Am/Pm marker	PM
m	Minute in hour	40	k	Hour in day (1-24)	20
s	Second in minute	50	K	Hour in am/pm (0-11)	2

Chapter 9

MULTITHREADING

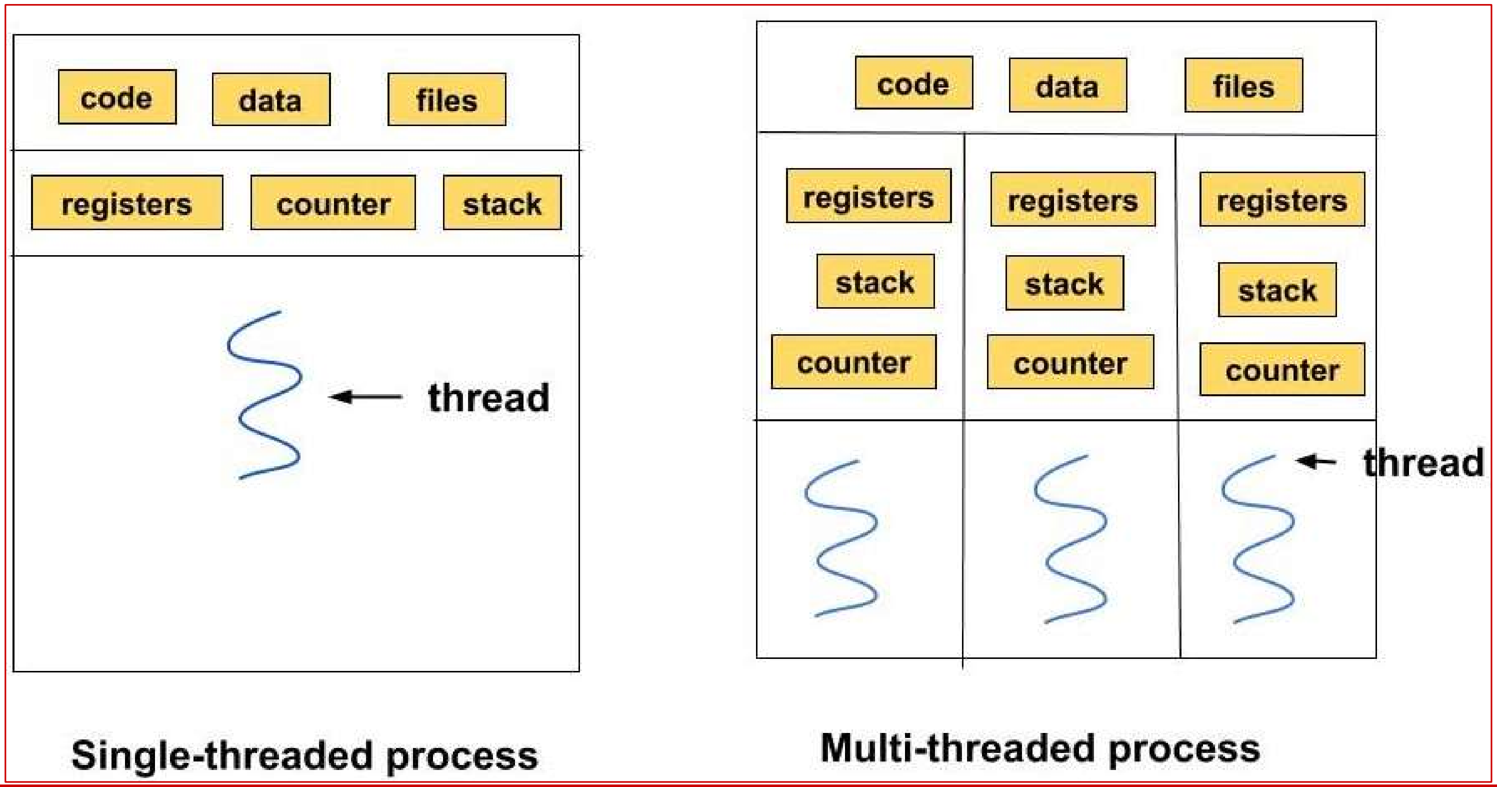
Multithreading

- In built support for Multithreading programming
- Multithreaded program - two or more parts of the program can run concurrently, each part is called a thread
- Enables programmers to write efficient programs that can make maximum use of processing power
- Most important benefit is minimum CPU idle time
- In single threaded environment program has to wait for each task to get processed (e.g reading from a file), most of the time program is idle waiting for input
- In multi threaded environment idle time is minimized because another thread can run when one is waiting

Multi processing vs Multi Threading

Multi Processing	Multi Threading
Alllows multiple programs to run concurrently	Multiple parts of a program can be run concurrently
Each process requires its own separate address space	Threads share same address space
Context switching from one process to another is costly	Context switching from one thread to another is not costly
Inter-process communication is expensive	Inter thread communication is inexpensive

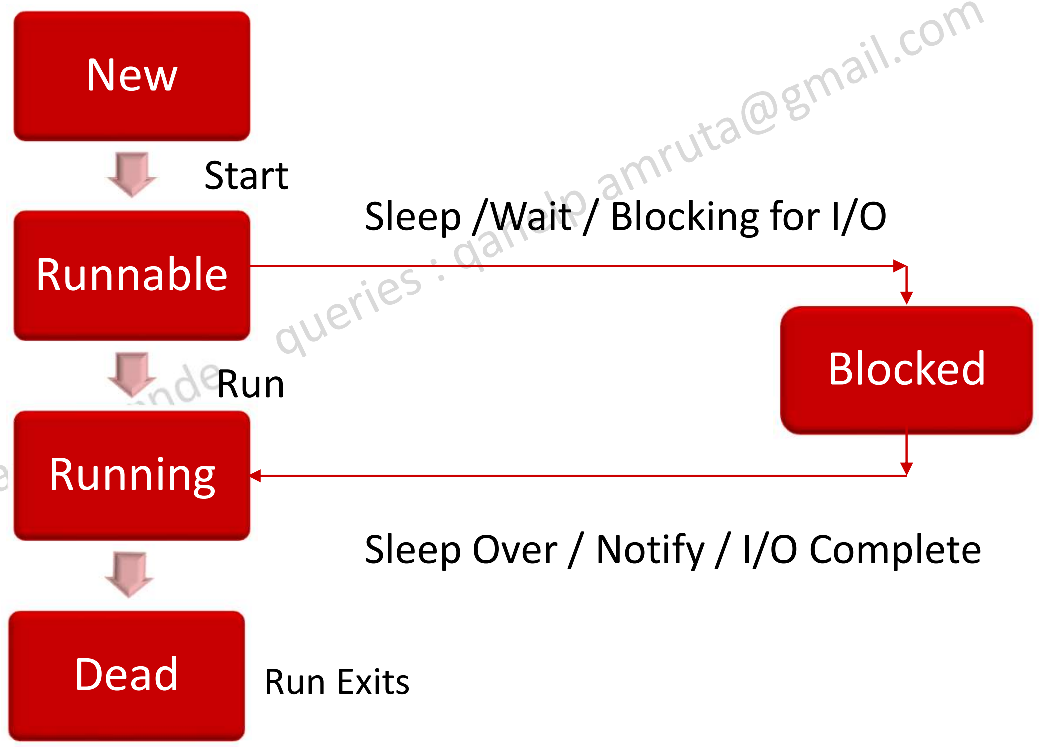
Single threaded vs Multithreaded



Java Thread Model

- Single threaded System – single thread of control runs in infinite loop, polling a single event queue to decide the next task
- Once polling mechanism returns the event loop dispatches control to the appropriate event handler
- In multithreaded environment the main loop is eliminated
- One thread can pause without stopping other parts of the program
- In multi core systems, it is actually possible to run multiple threads simultaneously , unlike the single core system where each thread just gets a slice of CPU time

Java Thread Model


New
Start
Runnable
Sleep /Wait / Blocking for I/O
Blocked
Run
Running
Sleep Over / Notify / I/O Complete
Dead
Run Exits

CPU Scheduling

- CPU Scheduler : Whenever the CPU becomes idle the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the Short-Term scheduler (CPU scheduler)
- Scheduler selects the process from the processes that are ready to execute and allocates CPU to it
- Dispatcher : Module that gives control of the CPU to the process selected by the CPU scheduler
- CPU scheduling can take place when:
  - Process switches from running state to waiting state (for I/O complete)
  - Process switches from running to ready (e.g. when interrupt occurs)
  - Process switches from waiting to ready (on I/O complete)
  - Process terminates

Non -preemptive and preemptive scheduling

- When the process is in waiting state (#1) or it has terminated (#4) , the CPU must be given to another process, there is no choice
- When scheduling takes place under only situation 1 & 4, it is called as non- preemptive (cooperative) scheduling
- In such scheduling scheme the CPU is assigned to another process only when the current process has either terminated or blocked [waiting]
- In other two situation the process was in running state when the CPU can be given to another ready process – preemptive scheduling
- In preemptive scheduling a lower priority thread that does not yield the processor can be preempted by higher priority thread

Thread Methods

sleep	Causes currently executing thread to sleep for specified number of milliseconds
yeild	Causes currently executing thread to pause and allow other thread to execute
join	Waits for a thread to die
Wait (From Object)	Allows a thread to wait inside a synchronized method
Notify (From Object)	Wakes up a single thread that is waiting
notifyAll (From Object)	Wakes up all threads those are waiting

Synchronization

- Synchronization in computer program means making one instruction run before another
- When Threads share a common resource, they need to make sure that resources will be used by one thread at a time
- Synchronization is implemented using concept of Monitor – An object used as a mutually exclusive lock
- Only one thread can own a monitor at a time, when one thread enters the monitor all other threads are suspended (said to be waiting for the monitor)
- Synchronized method – When one thread is inside the synchronized method, all other threads trying to call it on the same instance have to wait for the first one to return
- Another way is to put the calls to any method inside a synchronized block, and pass the object reference of the object being synchronized

Producer Consumer Problem

- Multithreaded programs have a division of labor between threads, some threads produce while others consume
- For instance, Event-driven programming

  An “event” is something that happens that requires the program to respond. Whenever event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them
- Here when an item is added/removed from the buffer the buffer is in inconsistent state
- Threads must have exclusive access to the buffer.
- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Reader Writer Problem

- Pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads
- When data is being written/modified it is essential to bar other threads from reading the data storage
- Readers and writers should execute different code before entering the critical section
- Synchronization constrains would be:
  - Any number of readers can be in the critical section simultaneously
  - Writers must have exclusive access to the critical section

Interthread Communication

- Implicit monitors are powerful, but for better control inter-process communication is useful
- To avoid polling, Java implements inter process communication using wait, notify and notifyAll methods
- These methods can be called only from within a synchronized context
- **wait** – Tells the calling thread to give up monitor and go to sleep until another thread enters the same monitor and calls notify or notifyAll method
- **notify** – wakes up the thread that called wait on the same object
- **notifyAll** - wakes up all the threads that called wait on the same object. One of the threads will be granted access

Java Concurrency Package

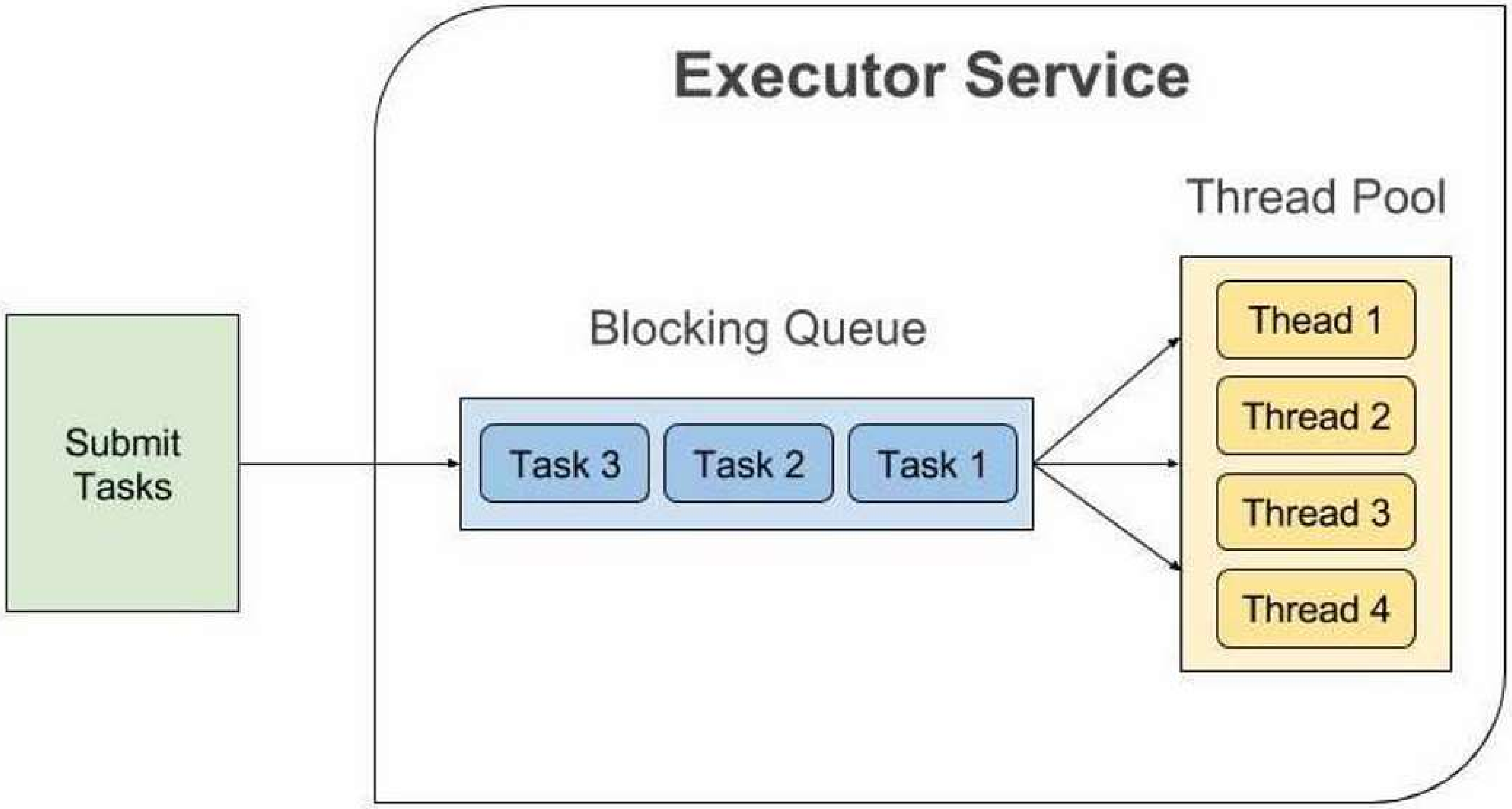
- Although java supports concurrent application development in a thread safe way using synchronization keyword and interthread communication, it is not ideal for application that extensively use multiple threads
- In JDK 5, java introduced util.concurrent package, commonly referred as concurrent API
- This supplies many features to support development of concurrent applications
- The key features include :
  - Synchronizers
  - Executors
  - Concurrent Collections
  - The fork/join framework

Concurrent Collections

- Concurrent collection represent set of classes that allow multiple threads to access and modify a collection concurrently
- For concurrent modifications no explicit synchronization is needed
- These collections provide thread-safe implementations of the traditional collection interfaces like List, Set, and Map
- Include following interface and classes:
  - Blocking queue implemented by ArrayBlockingQueue, LinkedBlockingQueue
  - ConcurrentMap implemented by ConcurrentHashMap
  - Classes like CopyOnWriteArrayList, CopyOnWriteArraySet

Executors

- The executors manage thread execution
- ExecutorService extends Executor, and it provides methods that manage the execution of threads
- Creating a java thread is much like creating a OS thread, which is an expensive operations
- When we need to run multiple tasks concurrently, an alternative is to use an Executor service
- The executor framework creates a thread pool with fixed no of thread which will execute the tasks concurrently
- So instead of creating a new thread every time, we submit multiple tasks which will be executed by the threads in the fixed thread pool
- The thread pool uses a thread safe data structure - blocked queue to store the tasks,


Executor Service
Submit Tasks
Blocking Queue
Task 3
Task 2
Task 1
Thread Pool
Thead 1
Thread 2
Thread 3
Thread 4

Executor Framework

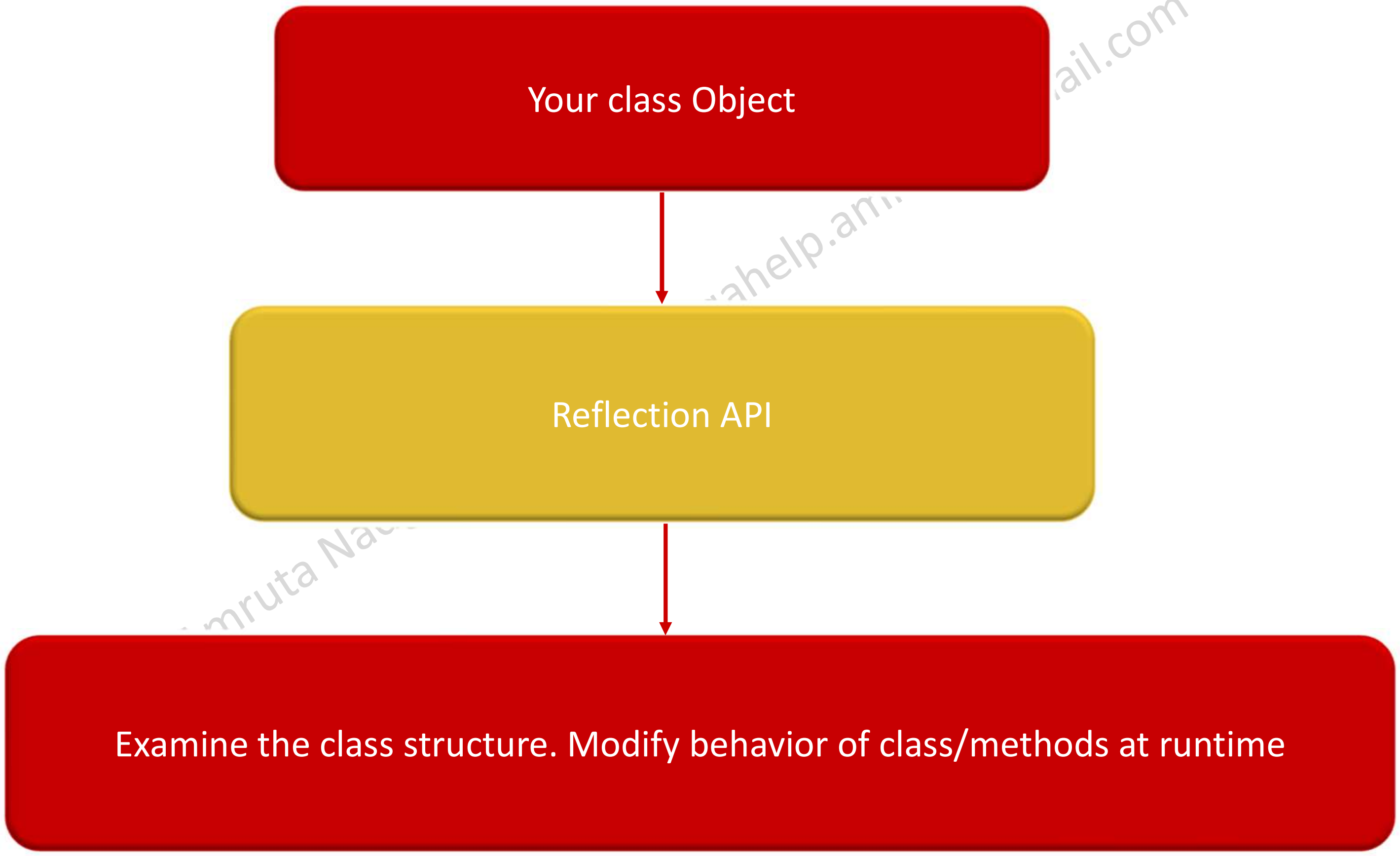
- The executor framework support different types of thread pools
  - Fixed Thread Pool
  - Cached Thread Pool
  - Scheduled Thread Pool
  - Single Thread Executor
- A cached thread pool uses a synchronous queue to hold only 1 task. It searches for a thread that can execute the task, and if all threads are busy it creates a new thread for execution
- Since a new thread is created for each task , if all threads are busy, that may lead to too many threads
- To avoid this cached thread pool kills the threads after 60 seconds of idle time

Chapter 10

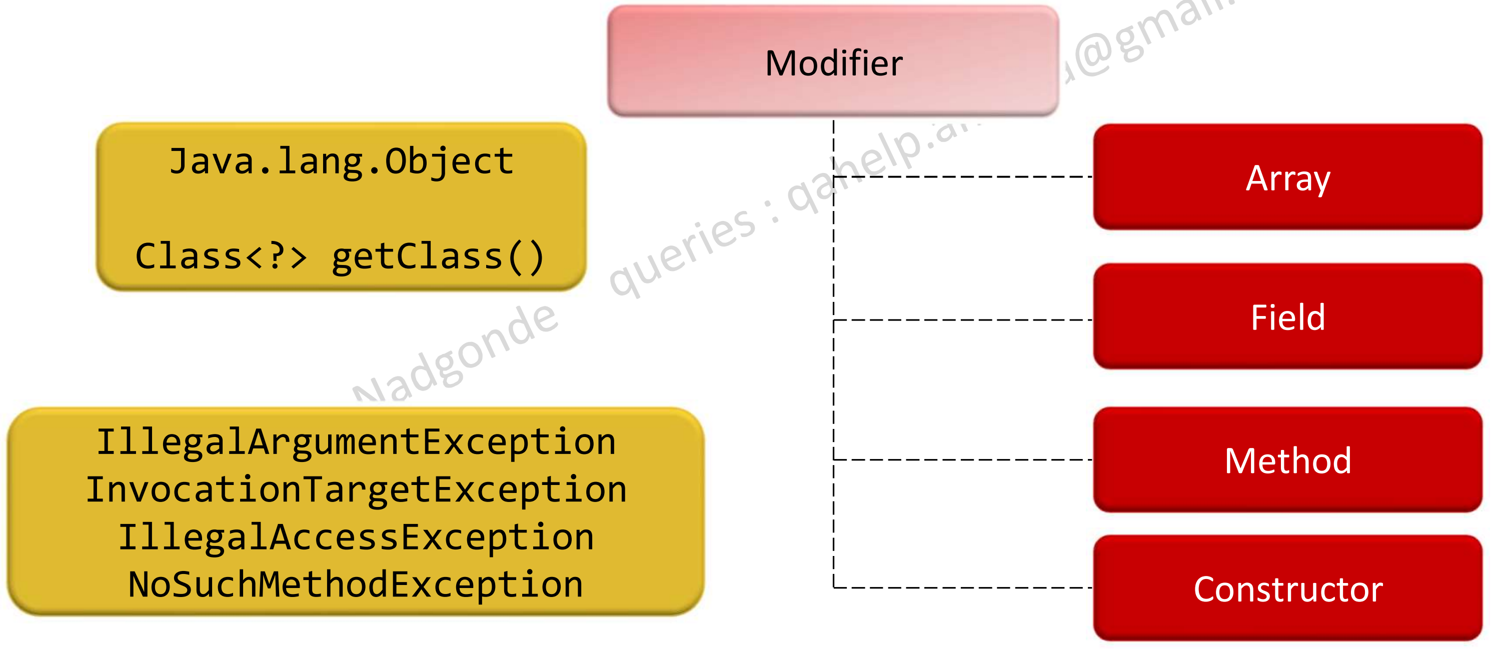
REFLECTION AND ANNOTATIONS

Java Reflection

- Reflection is a java API that allows us examine and /or modify behavior of methods, classes and interfaces at run time
- The classes that support reflection are available in java.lang.reflect package
- Using reflection we can get more information about fields, constructors, methods as well as annotations used in a class
- Reflection gives us information about the class to which an object belongs and also the methods of that class that can be executed by using the object.
- Through reflection, we can invoke methods at runtime irrespective of the access specifier used with them.
- The getClass() method of Object class is used to return the runtime class of any Object


Your class Object
Reflection API
Examine the class structure. Modify behavior of class/methods at runtime

The Reflection API



Runtime Class of an object

- The getClass() method of Object class returns the runtime Class of given object

```
Employee emp = new Employee();
Class c = emp.getClass()
```

- The static method of 'Class' can also be used to get the runtime class of the given object

```
Class c = Class.forName("Employee");
```

- A new instance of a class can be created using newInstance() method

```
Class c = Class.forName("Employee");
Employee e = (Employee)c.newInstance();
```

Class class

- Class objects are constructed automatically by the Java Virtual Machine as classes are loaded

Annotation[] getDeclaredAnnotations()	Returns annotations that are directly present on this element.
Constructor<?>[] getDeclaredConstructors	Returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
Field[] getDeclaredFields()	Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
Method [] getDeclaredMethods()	Returns an array containing Method objects reflecting all the declared methods of the class or interface represented by this Class object
Type getGenericSuperclass()	Returns the Type representing the direct superclass of the entity represented by this Class
Class <?> [] getInterfaces()	Determines the interfaces implemented by the class or interface represented by this object.
String getName()	Returns the name of the entity epresented by this Class object
boolean isAnnotationPresent()	Returns true if an annotation for the specified type is present on this element, else false.

Runtime Method invocation

- To invoke a method through reflection we should know its name and parameter types

```
Method method = emp.getClass().getDeclaredMethod("setEmpid",
        int.class);
 method.invoke(emp, 112);


Method methods [] = emp.getClass().getDeclaredMethods();

 if(method.getName().equals("simplePrivateMethod"))
  {
     method.setAccessible(true);
     method.invoke(emp);
  }
```

Array Class

- The Array class provides static methods to dynamically create and access Java arrays
- The newInstance() method creates a new array with the specified component type and length

```
T[] arrayOfTType = (T[]) Array.newInstance(T.class, size);
```

- The setXXX() methods are used to add an element in the array

```
Array.setT(T[], indexOfInsertion, elementToBeInserted);
```

- The getXXX() method is used to retrieve an element from the array

```
Array.getT(T[], indexOfRetrieval);
```

Annotations

- Introduced in JDK 1.5, a feature that allows to add supplemental information into a source file
- This information is used by various tools during development and deployment [e.g. by a source code generator]
- Annotations are created using a @ that precedes a keyword interface
- Annotations consist only method declaration, no body
- The method implementation is provided by Java
- Marker annotations are annotations without any methods

Annotations at Runtime

- The annotations which are retained till runtime [@Retention(RetentionPolicy.*RUNTIME*)]
- The annotations can be used to modify behaviour of the code at runtime
- The runtime annotations can be obtained by

```
method.isAnnotationPresent(Test.class)
```

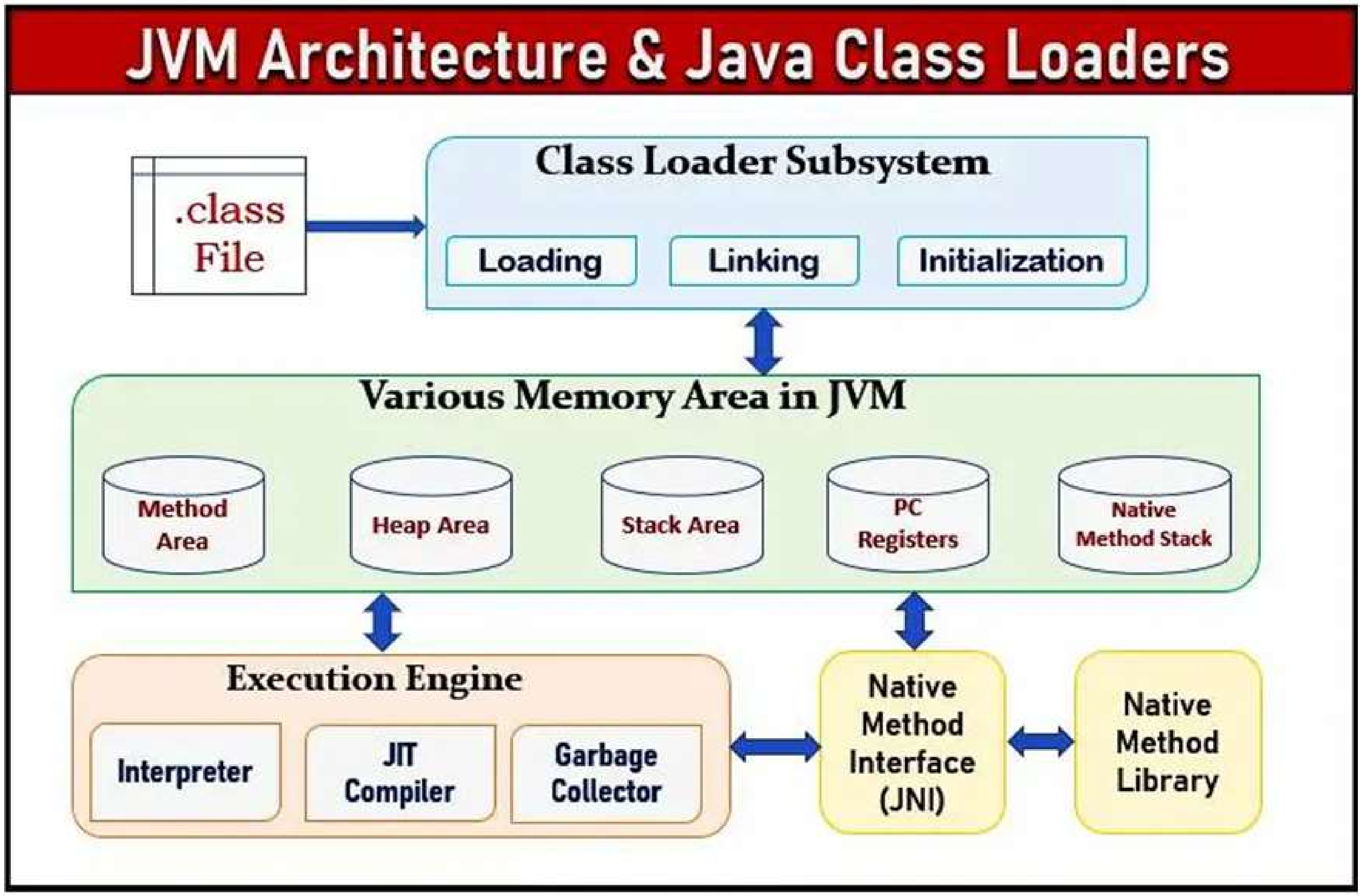
- The Annotation object is returned by using getAnnotation() method

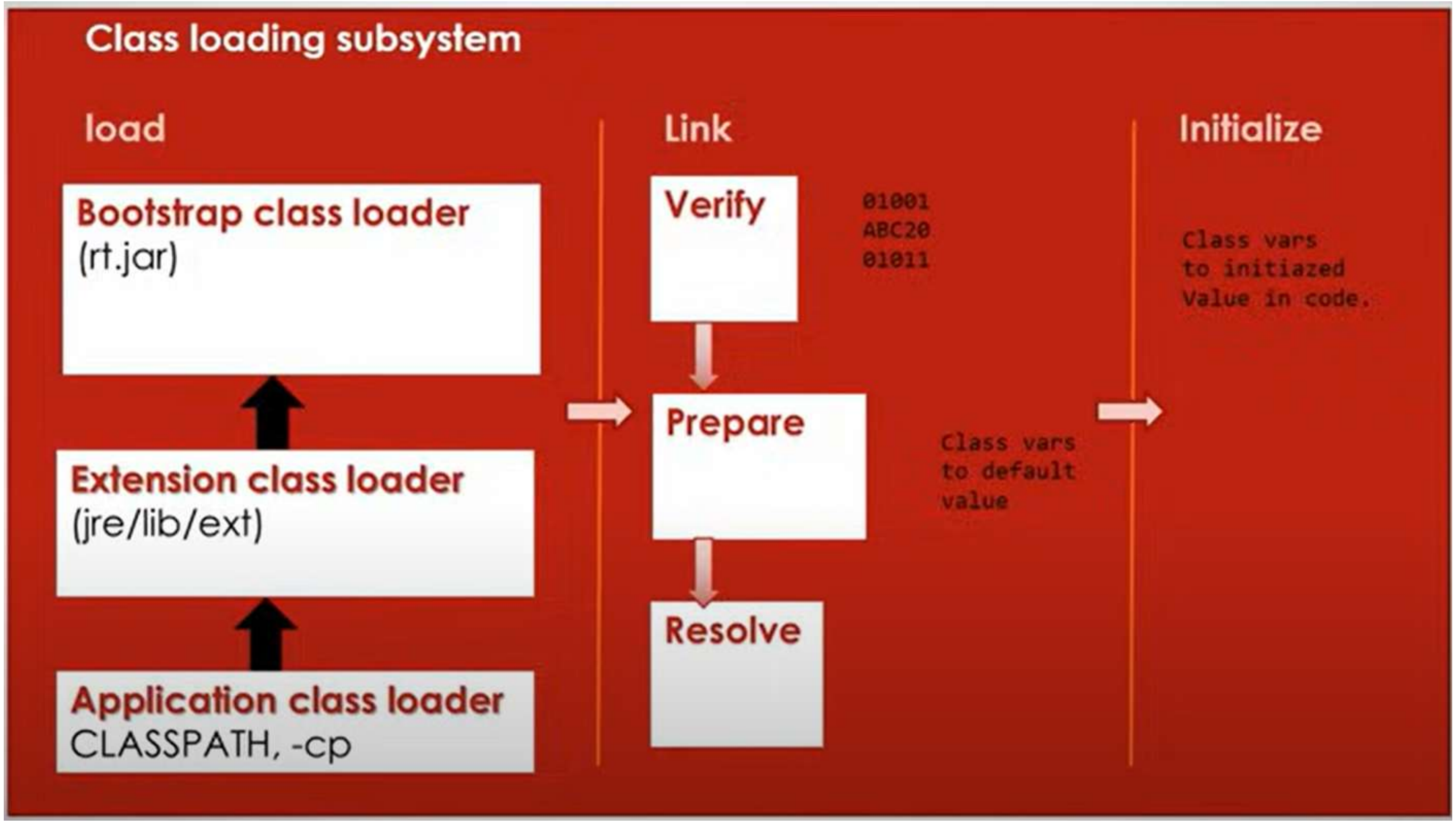
```
Test anno = method.getAnnotation(Test.class);
```

- Once the annotation object is obtained , it can be used to check annotation information and invoke methods / set field values etc

JVM

- JVM is responsible for loading and executing class file
- JVM instance is created when run command is given
- JVM contains class loader sub system and execution engine
- Class loader is responsible for loading the application .class file and in-built java classes
- The loaded classes [bytecode instructions] are fed to execution engine


JVM Architecture & Java Class Loaders
.class File
Class Loader Subsystem
Loading
Linking
Initialization
Various Memory Area in JVM
Method Area
Heap Area
Stack Area
PC Registers
Native Method Stack
Execution Engine
Interpreter
JIT Compiler
Garbage Collector
Native Method Interface (JNI)
Native Method Library


Class loading subsystem
load
Bootstrap class loader
(rt.jar)
Extension class loader
(jre/lib/ext)
Application class loader
CLASSPATH, -cp
Link
Verify
01001
ABC20
01011
Prepare
Class vars
to default
value
Resolve
Initialize
Class vars
to initiazed
Value in code.

Class Loader

- Consists of three phases : Load, Link, Initialize
- Load phase is responsible for loading the class file from physical memory / network
- Class loader types :
  - Bootstrap class loader : Loads java's internal classes residing in rt.jar which is distributed with JVM
  - Extension class loader : Loads class files for additional jars present in jre/lib/ext folder
  - Application class loader : Loads application class file from the value specified in the classpath variable

Class Loader

- The Link phase of class loader is further divided as verify, prepare and resolve
- The verify phase checks if the given byte code is as per the JVM specification [valid byte code]
- In the prepare phase static variables are assigned the memory [with default value]
- In the resolve phase all symbolic references in the class are resolved
- References to other classes, constant pool are changed from symbolic names to actual reference
- In the initialize phase the static initialization blocks get executed
- The static variables which were assigned with default values are now assigned with their actual values

Runtime Data [Memory] Area

- Method area : Method data corresponding to a class is stored here
- Many API in Java reflection require the data in method area. Static variables, class level constant pool etc. are saved here
- Heap: Heap is used to store the Java objects
- PC register : Contains the program counter for the next instruction per thread
- Java Stack : Contains method stack for the currently running threads. Each thread maintains its own method stack
- Native Method Stack : When native methods are invoked from one of the methods , native method stack is created

Execution Engine

- Interpreter is responsible for converting the byte code instruction into the native machine language
- Interpreter finds which native operation needs to be performed and executes that using native method interface (JNI)
- The JNI interface, interfaces the native method libraries present in the JVM
- JIT Compiler : Responsible for compiling the repeated instructions on the fly and keep them ready as target code. Such code is executed directly without any repeated interpretation
- Hotspot Profiler : Helps the JIT to locate the repeated code (Hotspots)
- Garbage Collector : Responsible for memory management of dead objects