Core Java Document

Why java is platform independent?

When Java Code is compiled, a byte code is generated which is independent of the system. This byte code is fed to the JVM (Java Virtual Machine) which resides in the system. Since every system has its own JVM (But JVM is platform dependent. Different OS will have different JVM's and hence different GC algorithms), it does not matter where you compile the source code. The byte code generated by the compiler can be interpreted by any JVM of any machine. Hence, it is called Platform independent Language. Java's byte-codes are designed to be read and interpreted in exactly same manner on any computer hardware or operating system that supports Java Runtime Environment.

OOPS Concepts in Java?

Inheritance - This is the mechanism of organizing and structuring software program. Though objects are distinguished from each other by some additional features but there are objects that share certain things common. In object-oriented programming, classes can inherit some common behavior and state from others. Inheritance in OOP allows to define a general class and later to organize some other classes simply adding some details with the old class definition. This saves work as the special class inherits all the properties of the old general class and as a programmer, you only require the new features. This helps in a better data analysis, accurate coding and reduces development time.

Abstraction - The process of abstraction in Java is used to hide certain details and only show the essential features of the object. In other words, it deals with the outside view of an object (interface).

Encapsulation - This is an important programming concept that assists in separating an object's state from its behavior. This helps in hiding an object's data describing its state from any further modification by external component. In Java there are four different terms used for hiding data constructs and these are PUBLIC, PRIVATE, PROTECTED and PACKAGE. It can also be termed as information hiding that prohibits outsiders in seeing the inside of an object in which abstraction is implemented.

Polymorphism - It describes the ability of the object in belonging to different types with specific behavior of each type. Therefore, by using this, one object can be treated like another and in this way, it can create and define multiple level of interface. Here the programmers need not have to know the exact type of object in advance and this is being implemented at runtime.
Overloading -> Same function name different signature, return type does not matter.
Overriding using classes -> Same name and signature but different classes.
Overriding using interfaces -> Same name and signature in all classes
We cannot reduce the visibility of methods
We cannot override private methods
We cannot override static methods

The big look at Interfaces Vs Abstract Classes

When to use an Interface: It asks you to start everything from scratch. You need to provide implementation of all the methods. Therefore, you should use it to define the contract, which you are unsure of how the different vendors/producers will implement. Therefore, you can say that Interfaces can be used to enforce certain standards.

When to use an Abstract Class: It is used mostly when you have partial implementation ready with you, but not the complete. Therefore, you may declare the incomplete methods as 'abstract' and leave it to the clients to implement it the way they actually want. Not all the details can be concrete at the base class level or different clients may like to implement the method differently.

- Abstract classes are used only when there is a "is-a" type of relationship between the classes. Interfaces can be implemented by classes that are not related to one another.
- You cannot extend more than one abstract class. You can implement more than one interface.
- Abstract class can implement some methods also. Interfaces cannot implement methods.
- With abstract classes, you are grabbing away each class's individuality. With Interfaces, you are merely extending each class's functionality.
- An Interface can only have public members whereas an abstract class can contain private as well as protected members.
- A class implementing an interface must implement all of the methods defined in the interface, while a class extending an abstract class need not implement any/most of the methods defined in the abstract class.
- The problem with an interface is, if you want to add a new feature (method) in its contract, then you MUST implement those methods in all of the classes, which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass

Big Difference between Singleton and Class with all Static Methods:

Although a singleton can be implemented as a static instance, it can also be lazily constructed, requiring no memory or resources until needed. Another notable difference is that static member classes cannot implement an interface, unless that interface is simply a marker. Therefore, if the class has to realize a contract expressed by an interface, it really has to be a singleton.

What is the difference between object oriented programming language and object based programming language?

Object based programming languages follow all the features of OOPs except Inheritance. Examples of object based programming languages are JavaScript, VBScript etc.

Difference between aggregation and Composition

Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

Composition implies a relationship where the child cannot exist independent of the parent.

What is blank final variable? Can we initialize blank final variable?
A final variable, not initialized at the time of declaration, is known as blank final variable. Yes we can initialize, only in constructor if it is non-static. If it is static blank final variable, it can be initialized only in the static block.

What is nested interface?
Any interface i.e. declared inside the interface or class, is known as nested interface. It is static by default.

Can an Interface have a class?
Yes, they are static implicitly.

- **Serializable** ➔ just to say to JVM that this class objects can be Serialized.
- **Prepare Statement** ➔ can take arguments dynamically; complied only once.
- **Statement** ➔ cannot take arguments dynamically; complied every-time called.
- **Collections.sort()** ➔ Modified MergeSort; n*log(n). But Depends.
- **Arrays.sort()** ➔ Modified QuickSort; n*log(n). But Depends.
- **Clone()** ➔ Must implement Clonable interface; protected Method; does Shallow Copy.
- **DeepCopy** ➔ Everything will be a new Copy; separate Memory.
- **ShallowCopy** ➔ Reference variables will be pointing to same reference; First level copy.
- **java className command** ➔ looks for the main(); allocates the memory; executes the Static block.
- **To Create your own checked Exception** ➔ extends Exception
- **To Create your own un-Checked Exception** ➔ extends RuntimeException
- **Comparison with double & float** ➔ Do not use "==" or "! =" because of rounding of problem.
- **Final primitive variable** ➔ Memory not allocated per instance basis.
- **Final reference variable** ➔ Memory allocated per instance basis.
- **Synchronization** ➔ Object lock; Class Object lock; If Static Method then thread acquires Class level Object Lock.
- Immutable objects are always thread-safe. Ex: - String, Wrapper Classes.
- If wait() method is invoked without holding the lock, it throws IllegalMonitorStateException.
- StringBuffer and StringBuilder Classes doesn't override the equals() method. So only objects with same reference are treated equals.
- The hash value of the empty string is zero.
- For any two strings str1 and str2, str1.intern() == str2.intern() is true if and only if str1.equals(str2) is true.
- **Serialization Is Not for Statics** ➔ Static variables are NEVER saved as part of the object's state, because they do not belong to the object. So transient also as no effect but still it can be declared.
- **Externalizable** ➔ is an Interface that extends Serializable Interface. And sends data into Streams in Compressed Format. It has two methods, writeExternal (ObjectOuput out) and readExternal (ObjectInput in). Mainly used when we need more control over the object that are serialized.

- There is one major difference between serialization and externalization: When you serialize an Externalizable object, a default constructor will be called automatically; only after that will the readExternal() method be called.
- **SerialVersionUID ➔** Each time an object is serialized, the object including every object in its graph is stamped with a version ID. Computed based on information about class structure. If the class is changed and the version ID is different, then deserialization will fail!
- **How to Synchronize an ArrayList?**
  ArrayList al = (ArrayList) Collections.synchronizedList(new ArrayList());
- **How to Synchronize a HashSet?**
  HashSet set = (HashSet) Collections.synchronizedSet(new HashSet());
- **How to Synchronize a HashMap?**
  HashMap hm = (HashMap) Collections.synchronizedMap(new HashMap());

Method Signature ➔ A method signature consists only the name of the method and the parameter types, their number and their order. Modifiers, return type and throws clause are not part of the signature.

String.intern() ➔ When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

String s1 = "abc"; ➔ Allocates memory in the String pool.
String s1 = new String("abc"); ➔ Allocates memory in the Heap & one String literal in String Pool.
String s1 = new String("abc").intern(); ➔ Checks in the String pool for the same String ("abc") if not present then allocates the memory on the Heap.

If,

```
String s1 = "abc";
String s2 = new String("abc");                  == will return false;

String s1 = new String("abc");
String s2 = new String("abc").intern();         == will return false;

String s1 = "abc";
String s2 = "abc";                              == will return true;

String s1 = "abc";
String s2 = new String("abc").intern();         == will return true;

String s1 = new String("abc");
String s2 = new String("abc");                  == will return false;

String a = "abc";
StringBuffer sb = new StringBuffer("xyz");
If (a == sb);                                   == Complier Error
If (a.equals(sb));                              == false
```

String str1 = “abc”;
String str2 = str1 + “def”;
String str3 = str1 + “d” + “e” + “f”;
In this case, only 2 objects are created in the string constant pool.

equals() method in String Class

```
public boolean equals(Object anObject) {
      if (this == anObject) {
       return true;
   }
   if (anObject instanceof String) {
       String anotherString = (String)anObject; int n = count;
       if (n == anotherString.count) {
         char v1[] = value;
         char v2[] = anotherString.value;
         int i = offset;
         int j = anotherString.offset;
         while (n-- != 0) {
             if (v1[i++] != v2[j++])
               return false;
         }
         return true;
       }
   }
      return false;
}
```

Classes

- Class can be PUBLIC, DEFAULT, ABSTRACT and FINAL, but not PROTECTED.
- Class that is not PUBLIC is not visible in another source file.
- Class can extend only one class but can implement any number of interfaces.
- You cannot instantiate an ABSTRACT class.
- You cannot extend a FINAL class.
- Only inner-classes or nested classes can be PRIVATE.

Static Methods

- They are class level methods.
- They cannot access non-static variables.
- They do not have access to THIS keyword as they don’t operate on objects.
- They should be always accessed using class name instead of an object reference.

Access Specifiers

- PUBLIC -> anywhere via object reference.
- PROTECTED -> package and subclasses via object reference.
- PRIVATE -> only in same class via methods.

Interfaces

- All variables are PUBLIC, STATIC and FINAL.
- Interface can extend any number of interfaces.
- We cannot always use ABSTRACT classes instead of interfaces, as there is a limitation that a class can extend only one class.

Does Java pass method arguments by value or by reference?
Java passes all arguments by value (copy of value), not by reference.

If no arguments on the command line, String[] of Main method will be empty or null?
It is empty. But not null.

Final variable
Once you declare a variable as FINAL it cannot occupy memory per instance basis.

Static block
Static block is executed exactly once when the class is first loaded into JVM.
Before going to the MAIN method the static block will execute.

Different ways to create objects in Java

1. Using new keyword

This is the most common way to create an object in java.

MyObject object = new MyObject();
A.class.getClassLoader().LoadClass("B");

2. Using Class.forName()

If we know the name of the class & if it has a public default constructor:

MyObject object = (MyObject)
Class.forName("subin.rnd.MyObject").newInstance();

3. Using clone()

The *clone()* can be used to create a copy of an existing object.

MyObject anotherObject = new MyObject();
*MyObject object = anotherObject.**clone()**;*

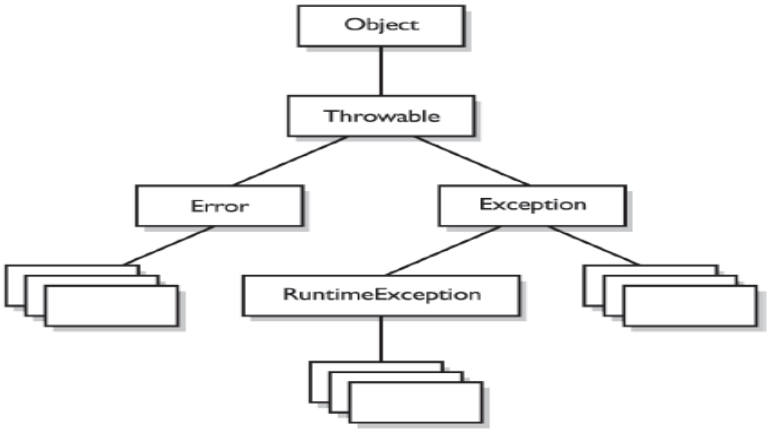
4. Using object deserialization

Object deserialization is nothing but creating an object from its serialized form.

ObjectInputStream inStream = new ObjectInputStream(anInputStream);
MyObject object = (MyObject) inStream.readObject();

Errors and Exceptions:
Errors: StackOverFlowError, NoClassDefFoundError, AssertionError

Checked Exception	UnChecked Exception
ClassNotFoundException	NullPointerException
CloneNotSupportedException	ArrayIndexOutOfBoundException
InstantiationException	StringIndexOutOfBoundException
InterruptedException	
NoSuchFieldException	
NoSuchMethodException	
IllegalAccessException	
RemoteException	



- Instance variables are initialized when the class is instantiated.
- It is possible to declare a local variable with the same name as an instance variable. It is known as ***shadowing***.
- Declaring a variable with the FINAL keyword makes it impossible to reinitialize that variable once it has been initialized with an explicit value (notice we said explicit rather than default). For primitives, this means that once the variable is assigned a value, the value cannot be altered.
- Static members exist before you ever make a new instance of a class, and there will be only one copy of the static member regardless of the number of instances of that class.
- Method invocations allowed by the compiler are based solely on the declared type of the reference, regardless of the object type.
- Which **over*ridden*** version of the method to call (in other words, from which class in the inheritance tree) is decided at ***runtime*** based on *object* type, but which **over*loaded*** version of the method to call is based on the *reference* type of the argument passed at ***compile* time**.
- The rule about INT-or-smaller expressions always resulting in an INT.
- The rule is that you can assign a subclass of the declared type, but not a superclass of the declared type.
- Java does not give local variables a default value; you must explicitly initialize them with a value.
- It is never legal to include the size of the array in your declaration. The JVM does not allocate space until you actually instantiate the array object. That is when the size matters.
- In order to save memory, two instances of the following wrapper objects will always be == when their primitive values are the same: **Boolean, Byte, Character, Short and Integer**.
- In the finalize() method you could write code that passes a reference to the object in question back to another object, effectively un-eligiblizing the object for garbage collection. If at some point later on this same object becomes eligible for garbage collection again, the garbage collector can still process this object and delete it. The garbage collector, however, will remember that, for this object, finalize() already ran, and it will not run finalize() again.
- Since the switch-case argument has to be resolved at compile time that means you can use only a constant or FINAL variable that is assigned a literal value. It is not enough to be FINAL, it must be a compile time constant.

- It is illegal to have more than one switch-case label using the same value.
- Even if there is a RETURN statement in the TRY block, the FINALLY block executes right after the RETURN statement is encountered, and **before the RETURN executes!**
- When an instance of a serializable class is de-serialized, the constructor does not run, and instance variables are NOT given their initially assigned values!
- Object references marked TRANSIENT will always be reset to NULL, regardless of whether they were initialized at the time of declaration in the class.
- If you are a serializable class, but your superclass is NOT serializable, then any instance variables you INHERIT from that superclass will be reset to the values they were given during the original construction of the object. **This is because the non-serializable class constructor WILL run!**
- When using HashSet or LinkedHashSet, the objects you add to them must override hashCode(). If they don't override hashCode(), the default Object's hashCode() method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.
- When you use a class that implements Map, any classes that you use as a part of the keys for that map must override the hashCode() and equals() methods.
- Runnable r = new Runnable(); r.run(); *Legal, but does not start a separate thread*
- int mid = (low + high) / 2; and int mid = (low + high) >>> 1; ➔ **SAME**

ArrayList

- Extends AbstractList. Implements List, Cloneable, RandomAccess, Serializable.
- Initial capacity is 10.
- Good choice for iteration.
- Order guaranteed. Dynamically expands and contracts.
- The get(), set(), iterator() operations run in constant time. The add() operation runs in amortized constant time, that is, adding N elements requires O(N) time.

LinkedList

- Extends AbstractSequentialList. Implements List, Deque, Cloneable, Serializable.
- Order guaranteed. Dynamically expands and contracts.
- Good choice for insertion and deletion.

HashMap

- Extends AbstractMap. Implements Map, Cloneable, Serializable.
- Hash table based implementation of the Map interface.
- It permits one NULL key and multiple NULL values.
- This implementation provides constant-time performance for the basic operations get() and put().
- Its performance is affect by initial capacity(16) and load factor(.75).
- Iterating over this set requires time proportional to the sum of the HashMap instance's size (the number of elements) plus its "capacity".

LinkedHashMap

- Extends HashMap. Implements Map.
- Hash table based implementation of the Map interface.
- It permits one NULL key and multiple NULL values.

- LinkedHashMap is well-suited for building LRU caches.
- Its performance is affect by initial capacity(16) and load factor(.75).
- Iteration times for this class is unaffected by capacity.
- LinkedHashMap defines the iteration ordering, which is normally the order in which keys were inserted.
- Iteration is fast. Addition and deletion is slow.

TreeMap

- Extends AbstractMap. Implements NavigableMap, Cloneable, Serializable.
- A Red-Black tree based implementation. The map is sorted according to the order of its keys.
- This implementation provides guaranteed log(n) time cost for the containsKey(), get(), put() and remove() operations.
- Mutually comparable.

HashSet

- Extends AbstractSet. Implements Set, Cloneable, Serializable.
- It makes no guarantees as to the iteration order of the set,
- This class permits the NULL elements.
- Duplicates not allowed.
- Its performance is affect by initial capacity(16) and load factor(.75).
- This implementation provides constant-time performance for the basic operations get() and put().
- Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance.

LinkedHashSet

- Extends HashSet. Implements Set, Cloneable, Serializable.
- LinkedHashSet defines the iteration ordering, which is normally the order in which keys were inserted.
- This class permits the NULL elements.
- Duplicates not allowed.
- Its performance is affect by initial capacity(16) and load factor(.75).
- Iteration times for this class are unaffected by capacity.
- Iteration is fast. Addition and Deletion is slow.

TreeSet

- Extends AbstractSet. Implements NavigableSet, Cloneable, Serializable.
- Ordered and Sorted. Duplicates not allowed.
- Mutually comparable.
- This implementation provides guaranteed log(n) time cost for the basic operations add(), remove() and contains().

Arrays.deepEquals(Object[], Object[])
Returns TRUE if the two specified arrays are deeply equal to one another. Unlike the equals(Object[], Object[]) method, this method is appropriate for use with nested arrays of arbitrary depth. Two array references are considered deeply equal if both are NULL or if they refer to arrays that contain the same number of elements and all corresponding pairs of elements in the two arrays are deeply equal.

Collections.sort()
Internally uses Arrays.sort().
This sort is guaranteed to be stable: equal elements will not be re-ordered as a result of the sort.

Parameter	Linked list	Array	Dynamic array
Indexing	O(*n*)	O(1)	O(1)
Insertion/deletion at beginning	O(1)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Insertion at ending	O(*n*)	O(1), if array is not full	O(1), if array is not full O(*n*), if array is full
Deletion at ending	O(*n*)	O(1)	O(*n*)
Insertion in middle	O(*n*)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Deletion in middle	O(*n*)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Wasted space	O(*n*)	0	O(*n*)

Condition	Required	Not Required (But Allowed)
`x.equals(y) == true`	`x.hashCode() == y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y) == true`
`x.equals(y) == false`		`No hashCode() requirements`
`x.hashCode() != y.hashCode()`	`x.equals(y) == false`	

java.lang.Comparable	java.util.Comparator
`int objOne.compareTo(objTwo)`	`int compare(objOne, objTwo)`
Returns negative if `objOne < objTwo` zero if `objOne == objTwo` positive if `objOne > objTwo`	Same as Comparable
You must modify the class whose instances you want to sort.	You build a class separate from the class whose instances you want to sort.
Only one sort sequence can be created	Many sort sequences can be created
Implemented frequently in the API by: String, Wrapper classes, Date, Calendar...	Meant to be implemented to sort instances of third-party classes.

Methods from the java.lang.Thread Class Some of the methods that can help us influence thread scheduling are as follows:

```
public static void sleep(long millis) throws InterruptedException
public static void yield()
public final void join() throws InterruptedException
public final void setPriority(int newPriority)
```

Methods from the java.lang.Object Class Every class in Java inherits the following three thread-related methods:

```
public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()
```

Give Up Locks	Keep Locks	Class Defining the Method
`wait ()`	`notify()` (Although the thread will probably exit the synchronized code shortly after this call, and thus give up its locks.)	`java.lang.Object`
	`join()`	`java.lang.Thread`
	`sleep()`	`java.lang.Thread`
	`yield()`	`java.lang.Thread`

Threading in Java

The most important properties of a thread are:

- runnable: the object whose `run()` method is run on the thread
- name: the name of the thread (used mainly for logging or other diagnostics)
- id: the thread's id (a unique, positive long generated by the system when the thread was created)
- threadGroup: the group to which this thread belongs
- daemon: the thread's daemon status. A daemon thread is one that performs services for other threads, or periodically runs some task, and is not expected to run to completion.
- contextClassLoader: the classloader used by the thread
- priority: a small integer between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY inclusive
- state: the current state of the thread
- interrupted: the thread's interruption status

Implicit Locks	Explicit Locks
• Are acquired only via synchronized statements (or methods) • Can only be used lexically • Can only be acquired and released in a LIFO way • Prevents common mistakes like forgetting to unlock • Are always blocking (and not iterruptibly so) • Are always reentrant • Cannot queue waiters fairly • Are pretty limited and inflexible, but easy and safe	• Very flexible • Can lock and unlock pretty much at any time (for example the lock and unlock calls can be in different methods) • Can be associated with multiple condition variables • Can be programmed to allow exlusive access to a shared resource **or** concurrent access to a shared resource • Can participate in hand-over-hand (chain) locking • Support non-blocking conditional acquisition • Support a time-out acquisition attempt • Support an interruptable acquisition attempt • Must be used responsibly • Can be made to be non-reentrant • Can be made fair • Can be made to support deadlock detection

Intrinsic synchronization	Explicit Locking using Lock and Condition
• Its easy to use technique with code more readable and compact • It is not possible to interrupt a thread waiting to acquire a lock, or attempt to acquire a lock without being willing to wait for it forever. • Responsibility of releasing a lock is handled by JVM even in case a exception occurs • JVM can do some performance optimization if synchronized keyword is used like lock elision & lock coarsening • As per Java Documentation *"The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired."*	• Provides same mutual exclusion & memory visibility guarantee as the synchronized block • Provides option for timed, polled or Interruptible locks helping avoid probabilistic deadlock. lock.tryLock() used for polling tryLock(long time, TimeUnit unit) for timed locking lockInterruptibly() for interruptible locking Interruptible lock acquisition allows locking to be used within cancelable activities. • Offers choice of fairness to the lock acquisition when multiple threads try to acquire the shared lock setting fairness flag to true as shown below. Lock lock = new ReentrantLock(true); This has significant performance cost when sued. • Ability to implement non-block-structured locking. Lock doesn't have to be released in the same block of code, unlike synchronized locks. • Lock has to be released manually in a finally block once we have modified the protected state

Introduction

In many cases, using implicit locking is enough. Other times, we will need functionalities that are more complex. In such cases, ***java.util.concurrent.locks*** package provides us with lock objects. When it comes to memory synchronization, the internal mechanism of these locks is the same as with implicit locks. The main advantages or improvements over implicit synchronization are:

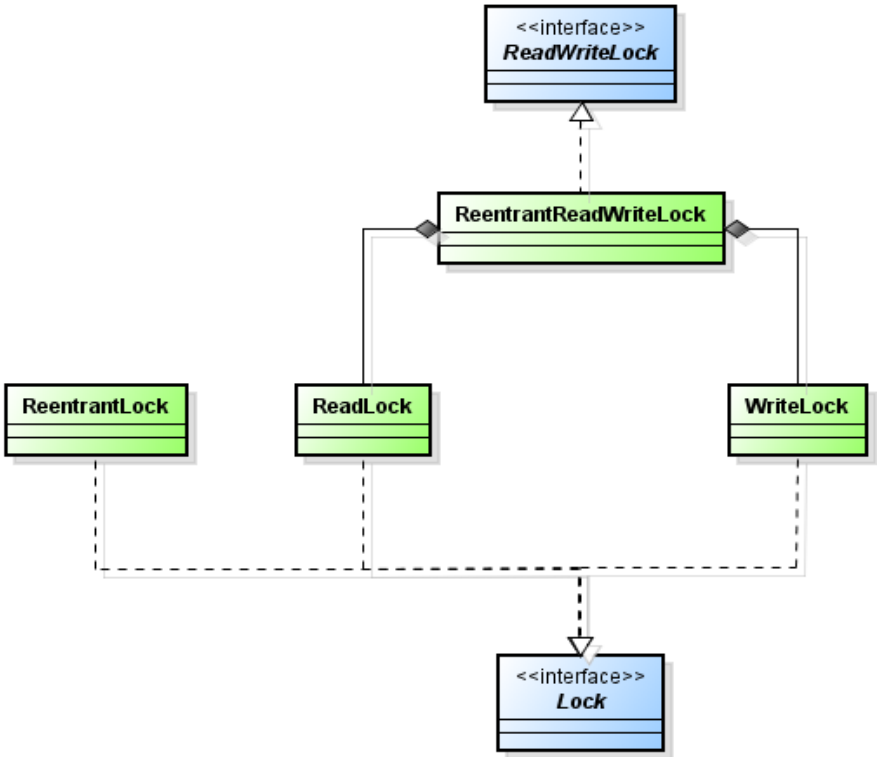
- Separation of locks by read or write.
- Some locks allow concurrent access to a shared resource (ReadWriteLock).
- Different ways of acquiring a lock:
  - Blocking: lock()
  - Non-blocking: tryLock()
  - Interruptible: lockInterruptibly()

Classification of lock objects

Lock objects implement one of the following two interfaces:

- **Lock**: Defines the basic functionalities that a lock object must implement. This means acquiring and releasing the lock. In contrast to implicit locks, this one allows the acquisition of a lock in a non-blocking or interruptible way (additionally to the blocking way). Main implementations:
  - ReentrantLock
  - ReadLock (used by ReentrantReadWriteLock)
  - WriteLock (used by ReentrantReadWriteLock)

- **ReadWriteLock**: It keeps a pair of locks, one for read-only operations and another one for writing. The read lock can be acquired simultaneously by different reader threads (as long as the resource isn't already acquired by a write lock), while the write lock is exclusive. In this way, we can have several threads reading the resource concurrently as long as there is not a writing operation. Main implementations:
  - ReentrantReadWriteLock



Lock Methods

The Lock interface has the following primary methods:

1. **The lock()** method locks the Lock instance if possible. If the Lock instance is already locked, the thread calling lock() is blocked until the Lock is unlocked.
2. **The lockInterruptibly() throws InterruptedException** method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.
3. **The tryLock()** method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.
4. **The tryLock(long time, TimeUnit unit) throws InterruptedException** works like the tryLock() method, except it waits up the given timeout before giving up trying to lock the Lock.
5. **The unlock()** method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException).

ReentrantLock

This lock works the same way as the synchronized block; one thread acquires the lock as long as it is not already acquired by another thread, and it does not release it until unlock is invoked. If the lock is already acquired by another thread, then the thread trying to acquire it becomes blocked until the other thread releases it. We are going to start with a simple example without locking, and then we will add a reentrant lock to see how it works.

```java
public class NoLocking {
    public static void main(String[] args) {
        Worker worker = new Worker();

        Thread t1 = new Thread(worker, "Thread-1");
        Thread t2 = new Thread(worker, "Thread-2");
        t1.start();
        t2.start();
    }

    private static class Worker implements Runnable {
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + " - 1");
            System.out.println(Thread.currentThread().getName() + " - 2");
            System.out.println(Thread.currentThread().getName() + " - 3");
        }
    }
}
```

Since the code above is not synchronized, threads will be interleaved.

```
Thread-2 - 1
Thread-1 - 1
Thread-1 - 2
Thread-1 - 3
Thread-2 - 2
Thread-2 - 3
```

Now, we will add a reentrant lock in order to serialize the access to the run method:

```java
public class ReentrantLockExample {
    public static void main(String[] args) {
        Worker worker = new Worker();

        Thread t1 = new Thread(worker, "Thread-1");
        Thread t2 = new Thread(worker, "Thread-2");
        t1.start();
        t2.start();
    }

    private static class Worker implements Runnable {
        private final ReentrantLock lock = new ReentrantLock();

        @Override
        public void run() {
            lock.lock();
            try {
                System.out.println(Thread.currentThread().getName() + " - 1");
                System.out.println(Thread.currentThread().getName() + " - 2");
                System.out.println(Thread.currentThread().getName() + " - 3");
            } finally {
                lock.unlock();
            }
        }
    }
}
```

The above code will safely be executed without threads being interleaved.

Additional ways of acquiring the lock are provided by implementing Lock interface:

- **lockInterruptibly**: The current thread will try to acquire the lock and become blocked if another thread owns the lock, like with the lock() method. However,

if another thread interrupts the current thread, the acquisition will be cancelled.

- **tryLock**: It will try to acquire the lock and return immediately, regardless of the lock status. This will prevent the current thread from being blocked if the lock is already acquired by another thread.
- **newCondition**: Allows the thread which owns the lock to wait for a specified condition.

Trying lock acquisition

In the following example, we have two threads, trying to acquire the same two locks. One thread acquires *lock2* and then it blocks trying to acquire *lock1*:

```
01 public void lockBlocking() {
02     LOGGER.info("{}|Trying to acquire lock2...", Thread.currentThread().getName());
03     lock2.lock();
04     try {
05         LOGGER.info("{}|Lock2 acquired. Trying to acquire lock1...",
   Thread.currentThread().getName());
06         lock1.lock();
07         LOGGER.info("{}|Both locks acquired", Thread.currentThread().getName());
08     } finally {
09         lock1.unlock();
10         lock2.unlock();
11     }
12 }
```

Another thread, acquires *lock1* and then it tries to acquire *lock2*.

```
01 public void lockWithTry() {
02     LOGGER.info("{}|Trying to acquire lock1...", Thread.currentThread().getName());
03     lock1.lock();
04     try {
05         LOGGER.info("{}|Lock1 acquired. Trying to acquire lock2...",
   Thread.currentThread().getName());
06         boolean acquired = lock2.tryLock(4, TimeUnit.SECONDS);
07         if (acquired) {
08             try {
09                 LOGGER.info("{}|Both locks acquired", Thread.currentThread().getName());
10             } finally {
11                 lock2.unlock();
12             }
13         }
14         else {
15             LOGGER.info("{}|Failed acquiring lock2. Releasing lock1",
   Thread.currentThread().getName());
16         }
17     } catch (InterruptedException e) {
18         //handle interrupted exception
19     } finally {
20         lock1.unlock();
21     }
22 }
```

Using the standard lock method, this would cause a dead lock, since each thread would be waiting forever for the other to release the lock. However, this time we are trying to acquire it with *tryLock* specifying a timeout. If it does not succeed after four seconds, it will cancel the action and release the first lock. This will allow the other thread to unblock and acquire both locks. Let us see the full example:

```java
public class TryLock {
    private static final Logger LOGGER = LoggerFactory.getLogger(TryLock.class);
    private final ReentrantLock lock1 = new ReentrantLock();
    private final ReentrantLock lock2 = new ReentrantLock();

    public static void main(String[] args) {
        TryLock app = new TryLock();
        Thread t1 = new Thread(new Worker1(app), "Thread-1");
        Thread t2 = new Thread(new Worker2(app), "Thread-2");
        t1.start();
        t2.start();
    }

    public void lockWithTry() {
        LOGGER.info("{}|Trying to acquire lock1...", Thread.currentThread().getName());
        lock1.lock();
        try {
            LOGGER.info("{}|Lock1 acquired. Trying to acquire lock2...",
Thread.currentThread().getName());
            boolean acquired = lock2.tryLock(4, TimeUnit.SECONDS);
            if (acquired) {
                try {
                    LOGGER.info("{}|Both locks acquired", Thread.currentThread().getName());
                } finally {
                    lock2.unlock();
                }
            }
            else {
                LOGGER.info("{}|Failed acquiring lock2. Releasing lock1",
Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
            //handle interrupted exception
        } finally {
            lock1.unlock();
        }
    }
```

```
36
37      public void lockBlocking() {
38          LOGGER.info("{}|Trying to acquire lock2...", Thread.currentThread().getName());
39          lock2.lock();
40          try {
41              LOGGER.info("{}|Lock2 acquired. Trying to acquire lock1...",
    Thread.currentThread().getName());
42              lock1.lock();
43              LOGGER.info("{}|Both locks acquired", Thread.currentThread().getName());
44          } finally {
45              lock1.unlock();
46              lock2.unlock();
47          }
48      }
49
50      private static class Worker1 implements Runnable {
51          private final TryLock app;
52
53          public Worker1(TryLock app) {
54              this.app = app;
55          }
56
57          @Override
58          public void run() {
59              app.lockWithTry();
60          }
61      }
62
63      private static class Worker2 implements Runnable {
64          private final TryLock app;
65
66          public Worker2(TryLock app) {
67              this.app = app;
68          }
69
70          @Override
71          public void run() {
72              app.lockBlocking();
73          }
74      }
75  }
```

If we execute the code, it will result in the following output:

```
1  13:06:38,654|Thread-2|Trying to acquire lock2...
2  13:06:38,654|Thread-1|Trying to acquire lock1...
3  13:06:38,655|Thread-2|Lock2 acquired. Trying to acquire lock1...
4  13:06:38,655|Thread-1|Lock1 acquired. Trying to acquire lock2...
5  13:06:42,658|Thread-1|Failed acquiring lock2. Releasing lock1
6  13:06:42,658|Thread-2|Both locks acquired
```

After the fourth line, each thread has acquired one lock and is blocked trying to acquire the other lock. At the next line, you can notice the four second lapse. Since we reached the timeout, the first thread fails to acquire the lock and releases the one it had already acquired, allowing the second thread to continue.

ReentrantReadWriteLock

This type of lock keeps a pair of internal locks (a *ReadLock* and a *WriteLock*). As explained with the interface, this lock allows several threads to read from the resource concurrently. This is especially convenient when having a resource that has frequent reads but few writes. As long as there isn't a thread that needs to write, the resource will be concurrently accessed.

The following example shows three threads concurrently reading from a shared resource. When a fourth thread needs to write, it will exclusively lock the resource, preventing reading threads from accessing it while it is writing. Once the write finishes and the lock is released, all reader threads will continue to access the resource concurrently:

```java
public class ReadWriteLockExample {
    private static final Logger LOGGER = LoggerFactory.getLogger(ReadWriteLockExample.class);
    final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    private Data data = new Data("default value");

    public static void main(String[] args) {
        ReadWriteLockExample example = new ReadWriteLockExample();
        example.start();
    }

    private void start() {
        ExecutorService service = Executors.newFixedThreadPool(4);
        for (int i=0; i<3; i++) service.execute(new ReadWorker());
        service.execute(new WriteWorker());
        service.shutdown();
    }

    class ReadWorker implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 2; i++) {
                readWriteLock.readLock().lock();
                try {
                    LOGGER.info("{}|Read lock acquired", Thread.currentThread().getName());
                    Thread.sleep(3000);
                    LOGGER.info("{}|Reading data: {}", Thread.currentThread().getName(),
data.getValue());
                } catch (InterruptedException e) {
                    //handle interrupted
                } finally {
                    readWriteLock.readLock().unlock();
                }
            }
        }
    }

    class WriteWorker implements Runnable {
        @Override
        public void run() {
            readWriteLock.writeLock().lock();
            try {
                LOGGER.info("{}|Write lock acquired", Thread.currentThread().getName());
                Thread.sleep(3000);
                data.setValue("changed value");
                LOGGER.info("{}|Writing data: changed value", Thread.currentThread().getName());
            } catch (InterruptedException e) {
                //handle interrupted
            } finally {
                readWriteLock.writeLock().unlock();
            }
        }
    }
}
```

The console output shows the result:

```
11:55:01,632|pool-1-thread-1|Read lock acquired
11:55:01,632|pool-1-thread-2|Read lock acquired
11:55:01,632|pool-1-thread-3|Read lock acquired
11:55:04,633|pool-1-thread-3|Reading data: default value
11:55:04,633|pool-1-thread-1|Reading data: default value
11:55:04,633|pool-1-thread-2|Reading data: default value
11:55:04,634|pool-1-thread-4|Write lock acquired
11:55:07,634|pool-1-thread-4|Writing data: changed value
11:55:07,634|pool-1-thread-3|Read lock acquired
11:55:07,635|pool-1-thread-1|Read lock acquired
11:55:07,635|pool-1-thread-2|Read lock acquired
11:55:10,636|pool-1-thread-3|Reading data: changed value
11:55:10,636|pool-1-thread-1|Reading data: changed value
11:55:10,636|pool-1-thread-2|Reading data: changed value
```

As you can see, when writer thread acquires the write lock (thread-4), no other threads can access the resource.

Threads

1. Threads calling non-static synchronized methods in the same class will only block each other if they are invoked using the same instance. That is because they each lock on **THIS** instance, and if they are called using two different instances, they get two locks, which do not interfere with each other.
2. Threads calling static synchronized methods in the same class will always block each other—they all lock on the same Class instance.
3. A static synchronized method and a non-static synchronized method will not block each other ever. The static method locks on a Class instance while the non-static method locks on the **THIS** instance—these actions do not interfere with each other at all.
4. Threads that synchronize on the same object will block each other. Threads that synchronize on different objects will not.
5. If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method.
6. If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods.
7. If a thread goes to sleep, it holds any locks it has—it does not release them.
8. A thread can acquire more than one lock.

ReentrantLock is dangerous because it doesn't automatically clean up the lock when control leaves the guarded block. Read-Write locks allow a resource can be accessed by multiple readers or a single writer at a time, but not both.

Java Thread Scheduler

The JVM is based on pre-emptive, and priority based scheduling algorithm. The thread with more priority is given first preference than the thread with less priority. The thread with more priority relinquishes (empties) the thread with less priority that is being executed. If the threads of equal priority are in the pool, the waiting time is taken in consideration. Nature of threads sometimes affects. The daemon threads are given less importance and are executed only when no other thread is available for execution.

When JVM starts executing a Java program, it creates few threads for the execution.

- **Main** is a method for us, but main is a thread for JVM. The execution starts with main thread.
- **Garbage collector** is a daemon thread that comes into action before every thread.
- **Event Dispatcher** is a thread which will take care of events raised by the components like click of a button etc.
- There is one more, **Timer thread**, which maintains the time for methods like sleep() etc.

What is the difference between pre-emptive scheduling and time slicing?

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states, or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Executor is based on the producer-consumer pattern, where activities that submit tasks are the producers and the threads that execute tasks are the consumers. The life-cycle implied by ExecutorService has 3 states - **running, shutting down (shutdown/shutdownNow) and terminated**. The life-cycle of a task executed by an Executor has 4 phases: **Created, Submitted, Started and Completed.**

Qns-1: What do you understand by Executor Framework in Java?
Ans: Executor Framework in java has been introduced in JDK 5. Executor Framework handles creation of thread, creating the thread pool and checking health while running and terminates if needed.

Qns-2: What is the role of the interface ExecutorService in Java?
Ans: ExecutorService provides different methods to start and terminate thread. There are two methods **execute()** and **submit()** in ExecutorService.
Execute() method is used for threads which are Runnable.
Submit() method is used for Callable threads.

- **void shutdown()**
  Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Invocation has no additional effect if already shut down. This method does not wait for previously submitted tasks to complete execution. Use awaitTermination() to do that.
- **List<Runnable> shutdownNow()**
  Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. This method does not wait for actively executing tasks to terminate. Use awaitTermination() to do that.
- **boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException**
  Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first. After the specified time thread pool is terminated. Suppose we need to terminate a task just now, then we can do as
  ExecutorService.awaitTermination(0, TimeUnit.SECONDS)
- **Future<T> submit(Callable<T> task)**
  Submits a value-returning task for execution and returns a Future representing the pending results of the task. The Future's get() method will return the task's result upon successful completion.
- **Future<T> submit(Runnable task, T result)**
  Submits a Runnable task for execution and returns a Future representing that task. The Future's get() method will return the given result upon successful completion.
- **Future<?> submit(Runnable task)**
  Submits a Runnable task for execution and returns a Future representing that task. The Future's get() method will return null upon successful completion.
- **void execute(Runnable command)**
  From the parent Executor interface. Executes the given command at some time in the future. The command may execute in a new thread, in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.

Qns-3: What is Executors in java Executor Framework?

Ans: Factory and utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory & Callable classes defined in this package.

- **ExecutorService newSingleThreadExecutor()**
  Creates an Executor that uses a single worker thread operating off an unbounded queue.
- **ExecutorService newFixedThreadPool()**
  It returns the pool with fixed number of size. We need to pass the number of threads to this method. If concurrently tasks are submitted more than the pool size, then rest of task needs to wait in the queue.
- **ScheduledExecutorService newScheduledThreadPool()**
  This also creates a fixed size pool but it can schedule the thread to run after some defined delay. It is useful to schedule the task.
- **ExecutorService newCachedThreadPool()**
  Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to <tt>execute</tt> will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache.
- **ThreadFactory privilegedThreadFactory()**
  Returns a thread factory used to create new threads that have the same permissions as the current thread.
- **ExecutorService unconfigurableExecutorService(ExecutorService executor)**
  Returns an object that delegates all defined {ExecutorService} methods to the given executor, but not any other methods that might otherwise be accessible using casts. This provides a way to safely "freeze" configuration and disallow tuning of a given concrete implementation.

Qns-4: What is the role of FutureTask and Future in java?

Ans: FutureTask (Class) is a cancellable asynchronous computation in java. It can cancel the task which is running. Once the FutureTask will be cancelled, it cannot be restarted.

Future (Interface) is result of asynchronous computation. Future checks if task is complete and if completed it gets the output.

Qns-5: What are the different policy in Executor Framework?

Ans: There are different policy (static classes) within ThreadPoolExecutor in java.

- **AbortPolicy**: AbortPolicy is a handler for rejected task.
- **CallerRunsPolicy**: A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute() method, unless the executor has been shutdown, in which case the task is discarded.
- **DiscardOldestPolicy**: A handler for rejected tasks that discards the oldest unhandled request and then retries {execute}, unless the executor is shutdown, in which case the task is discarded.
- **DiscardPolicy**: A handler for rejected tasks that silently discards the rejected task.

The java.util.concurrent.ThreadPoolExecutor is an implementation of the ExecutorService interface. The ThreadPoolExecutor executes the given task (Callable or Runnable) using one of its internally pooled threads. The thread pool contained inside the ThreadPoolExecutor can contain a varying amount of threads. The number of threads in the pool is determined by these variables: ***corePoolSize & maximumPoolSize.***

If less than corePoolSize threads are created in the thread pool when a task is delegated to the thread pool, then a new thread is created, even if idle threads exist in the pool.

If the internal queue of tasks is full, and corePoolSize threads or more are running, but less than maximumPoolSize threads are running, then a new thread is created to execute the task.

Qns-6: How to get return value of a callable thread in java Executor Framework?

Ans: Using Future, we can get the return value of callable thread.

```
ExecutorService exService = Executors.newCachedThreadPool();
Future<Integer> future = exService.submit(new CallableThread());
int val = future.get();
```

ClassLoaders in Java

ClassLoader Mechanism

The ClassLoader class uses a delegation model to search for classes and resources. Each instance of ClassLoader has an associated parent ClassLoader. When requested to find a class or resource, a ClassLoader instance will delegate the search for the class or resource to its parent ClassLoader before attempting to find the class or resource itself. The virtual machine's built-in ClassLoader, called the "bootstrap class loader", does not itself have a parent but may serve as the parent for a ClassLoader instance.

Normally, the Java virtual machine loads classes from the local file system in a platform-dependent manner. However, some classes may not originate from a file, they may originate from other sources, such as the network, or they could be constructed by an application. The method defineClass() converts an array of bytes into an instance of class CLASS. Instances of this newly defined class can be created using Class.newInstance.

The methods and constructors of objects created by a ClassLoader may reference other classes. To determine the class(es) referred to, the Java virtual machine invokes the loadClass() method of the ClassLoader that originally created the class.

Class loading proceeds according to the following general algorithm:

- Determine whether the class has been loaded before. If so, return the previously loaded class.
- Consult the Bootstrap ClassLoader to attempt to load the class from the CLASSPATH. This prevents external classes from spoofing trusted Java classes.
- See whether the ClassLoader is allowed to create the class being loaded. The Security Manager makes this decision. If not, throw a security exception.
- Read the class file into an array of bytes. The way this happens differs according to particular ClassLoader. Some ClassLoaders may load classes from a local database. Others may load classes across the network.
- Construct a Class object and its methods from the class file.
- Resolve classes immediately referenced by the class before it is used. These classes include classes used by static initializes of the class and any classes that the class extends.
- Check the class file with the Verifier.

There are three default class loader used in Java, **Bootstrap**, **Extension** and **System or Application ClassLoader**. Every ClassLoader has a predefined location, from where they loads class files.

- **Bootstrap ClassLoader** is responsible for loading standard JDK class files from rt.jar and it is parent of all ClassLoaders in Java. Bootstrap ClassLoader don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw NullPointerException in Java. Bootstrap ClassLoader is also known as **Primordial ClassLoader** in Java.

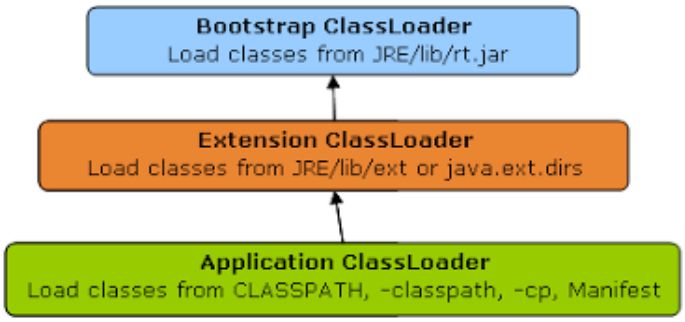
- **Extension ClassLoader** delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class from jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by sun.misc.Launcher$ExtClassLoader.

- Third default ClassLoader used by JVM to load Java classes is called **System or Application ClassLoader** and it is responsible for loading application specific classes from CLASSPATH environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application ClassLoader is a child of Extension ClassLoader and it is implemented by sun.misc.Launcher$AppClassLoader class.

Except Bootstrap ClassLoader, which is implemented in native language mostly in C, all Java ClassLoaders are implemented using java.lang.ClassLoader.

Location from which Bootstrap, Extension & Application ClassLoader load Class files:

- Bootstrap ClassLoader - JRE/lib/rt.jar
- Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs
- Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.



Delegation principles

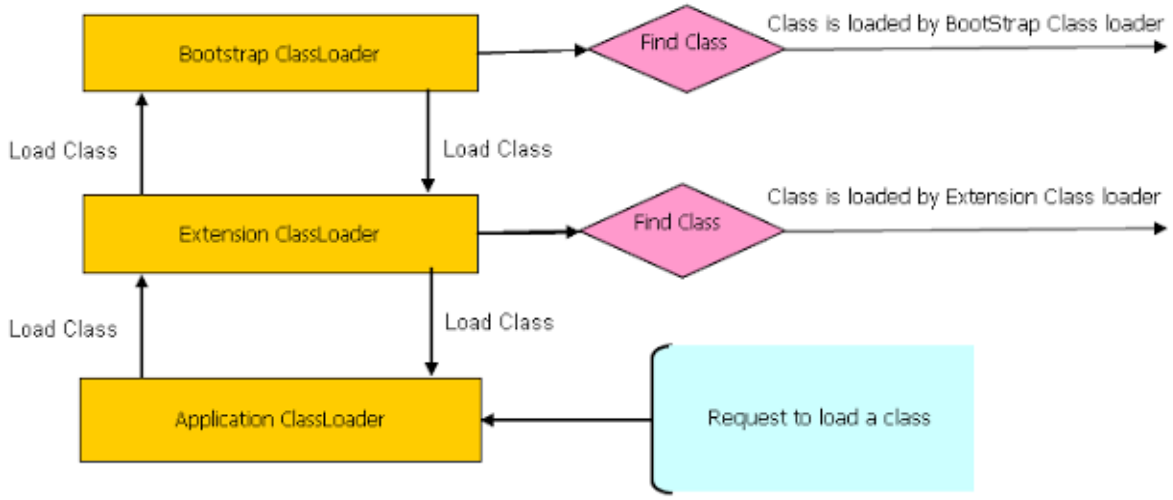
A class is loaded in Java, when it is needed. Suppose you have an application specific class called ABC.class, first request of loading this class will come to Application ClassLoader, which will delegate to its parent Extension ClassLoader that further delegates to Bootstrap ClassLoader. Bootstrap will look for that class in rt.jar and since that class is not there, request comes to Extension ClassLoader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there then Extension ClassLoader will load that class and Application ClassLoader will never load that class but if it's not loaded by extension ClassLoader then Application ClassLoader loads it from Classpath in Java. **The Classpath is used to load class files while PATH is used to locate executable like javac or java command.**

Visibility Principle

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class ABC is loaded by Application ClassLoader, then trying to load class ABC explicitly using extension ClassLoader will throw java.lang.ClassNotFoundException.

Uniqueness Principle

According to this principle, Child ClassLoader should not load a class loaded by Parent again. Though it is completely possible to write ClassLoader, which violates Delegation and Uniqueness principles and loads class by itself, it is not something, which is beneficial. You should follow all class loader principle while writing your own ClassLoader.



How to load class explicitly in Java?

Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drivers.

You can pass name of a ClassLoader, which should be used to load that particular class along with binary name of class. Class is loaded by calling loadClass() method of java.lang.ClassLoader class which calls findClass() method to locate bytecodes for corresponding class.

Extension ClassLoader uses java.net.URLClassLoader, which search for class files and resources in JAR and directories. Any search path which is ended using "/" is considered directory. If findClass() does not find the class then it throws java.lang.ClassNotFoundException and if it finds, it calls defineClass() to convert bytecodes into a *.class instance which is returned to the caller.

No class found

Variants
- ClassNotFoundException
- NoClassDefFoundError

Helpful
- IDE class lookup (Ctrl+Shift+T in Eclipse)
- find *.jar -exec jar -tf '{}'\; | grep MyClass
- URLClassLoader.getUrls()
- Container specific logs

Wrong class found

Variants
- IncompatibleClassChangeError
  AbstractMethodError
  NoSuch(Method|Field)Error
- ClassCastException, IllegalAccessError

Helpful
- -verbose:class
- ClassLoader.getResource()
- javap -private MyClass

More than one class found

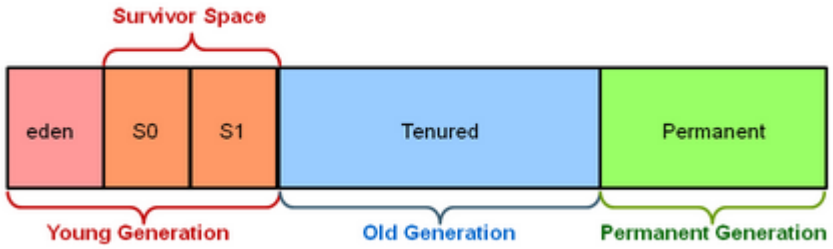
Variants
- LinkageError (class loading constraints violated)
- ClassCastException, IllegalAccessError

Helpful
- -verbose:class
- ClassLoader.getResource()

Web ClassLoaders do not ask Parent!

Garbage Collection in Java

Reference	Garbage Collection
Strong Reference	Not eligible for garbage collection
Soft Reference	Garbage collection possible but will be done as a last option
Weak Reference	Eligible for Garbage Collection
Phantom Reference	Eligible for Garbage Collection



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a ***minor garbage collection***. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.
Stop the World Event - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes.

Composition of the Young Generation
The young generation is divided into 3 spaces:

- One **Eden** space
- Two **Survivor** spaces

The order of execution process of each space is as below:

1. The majority of newly created objects are located in the Eden space.
2. After one GC in Eden space, surviving objects are moved to one of the Survivor spaces.
3. After a GC in the Eden space, the objects are piled up into the Survivor space, where other surviving objects already exist.
4. Once a Survivor space is full, surviving objects are moved to the other Survivor space. Then, Survivor space that is full will be changed to a state where there is no data at all.
5. The objects that survived these steps that have been repeated a number of times are moved to the old generation.

As you can see by checking these steps, one of the Survivor spaces must remain empty. If *data exists in both Survivor spaces, or the usage is 0 for both spaces*, then take that as a sign that **something is wrong with your system**.

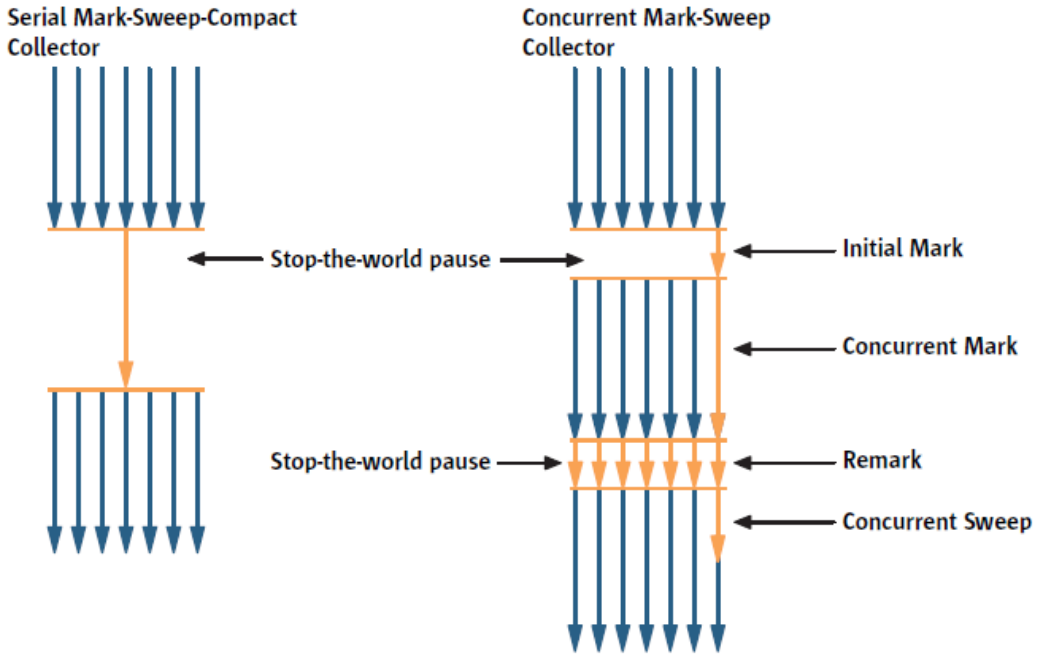
The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object is moved to the old generation. Eventually the old generation needs to be collected. This event is called a ***major garbage collection***. Major garbage collection are also Stop the World

events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here. Classes may get collected (unloaded) if the JVM finds they are no longer needed, and space may be needed for other classes. ***The permanent generation is included in a full garbage collection.***

What if an object in old generation need to reference an object in young generation?
To handle these cases, there is something called the "**card table**" in the old generation, which is a *512 byte chunk*. Whenever an object in the old generation references an object in the young generation, it is recorded in this table. When a GC is executed for the young generation, only this card table is searched to determine whether or not it is subject for GC, instead of checking the reference of all the objects in the old generation. This card table is managed with **write barrier**. This *write barrier* is a device that allows a faster performance for minor GC. Though a bit of overhead occurs because of this, the overall GC time is reduced.

Concurrent Mark-Sweep (CMS) Collector
A collection cycle for the CMS collector starts with a short pause, **called the initial mark** that identifies the initial set of live objects directly reachable from the application code.
Then, during the **concurrent marking phase**, the collector marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields while the marking phase is taking place, not all live objects are guaranteed to be marked at the end of the concurrent marking phase.
To handle this, the application stops again for a second pause, **called remark**, which finalizes marking by revisiting any objects that were modified during the concurrent marking phase. Because the remark pause is more substantial than the initial mark, multiple threads are run in parallel to increase its efficiency.
At the end of the remark phase, all live objects in the heap are guaranteed to have been marked, so the subsequent **concurrent sweep phase** reclaims all the garbage that has been identified. Since some tasks, such as revisiting objects during the remark phase, increase the amount of work the collector has to do, its overhead increases as well. This is a typical trade-off for most collectors that attempt to reduce pause times.
The CMS collector is the only collector that is non-compacting. That is, after it frees the space that was occupied by dead objects, it does not move the live objects to one end of the old generation.



Fragmentation

Whenever we create a new object in Java, the JVM automatically allocates a block of memory large enough to fit the new object on the heap. Repeated allocation and reclamation leads to memory fragmentation, which is similar to disk fragmentation. Memory fragmentation leads to two problems:

- **Reduced allocation speed:** The JVM tracks free memory in lists organized by block size. To create a new object, Java searches through the lists to select and allocate an optimally sized block. Fragmentation slows the allocation process, effectively slowing the application execution.
- **Allocation Errors:** Allocation errors happen when fragmentation becomes so great that the JVM is unable to allocate a sufficiently large block of memory for a new object.