# General Expectations

**General**

General expectation of client from core java developer is he should strong in Data structure , multithreading and fundamental , client expect when complex question asked from candidate he should attempt and try to solve the problem , the idea of asking complex question is they want to see are developer attempting to solve the problem which he has never solved , if trying to solve what approach taking , how  breaking a problem into small  part ,how using design principle and pattern.

 some time client asking open ended question, the purpose of asking open ended question is , developer should answer based on his past experience , if developer answer in totality with confidence the probability of   clearing interview become very high

the idea of asking ambiguous question  is the developer should asked question to clarify the requirement if requirement is not clear and then try to solved the problem

**Problem Solving**

The idea of asking problem solving is to evaluate developer design skills  , some time client not expecting concrete  implementation , they want to understand what approach developer taking to solved the design problem , are they giving up or the attempting to solve the problem

**Fundamental**

In fundamental client expecting , developer should know purpose of all the keyword , static binding , dynamic binding , overloading rule , overriding rule in terms of access modifier, exception handling, impact of dynamic linking on performance , how to improve performance by using final keyword , whats default implementation of has code and equal , cloning, immutability, advantage of immutability , importance of final in security , Exception handling rules

**Data Structure**

In Data structure the expectation is developer should know all basic data structure and how those basic data structure internally working , base on use case developer should answer question which data structure is best fit for given use case , how hashing concept is working in hash set and map , how to improve map /set performance by optimizing hash code , whats are time complexity of different operation on data structure , how re sizing is happening in data structure , how to use comparable comparator , how to implement different sorting algorithm .   How to optimize data structure by changing the capacity and load factor  , could able to implement one data structure by using another , could able to answer how to implement data structure

**Concurrent API**

 developer should know how concurrent hash map internally managing a lock how segmentation is working ,how many thread can work on concurrent hash map , benefit of using concurrent hash map over hash table and synchronize map , what kind of business use case can be implemented by using concurrent hash map , how blocking queue is working what kind of problem can be solved by using blocking queue , when we should used linked blocking queue and when array blocking queue whats implementation of blocking queue , how to use blocking queue in inter thread communication, whats fail safe iterator

how to implement thread pool , whats advantage of thread pool , how many type of thread pool do we have , how we can use executor service , how to use executor service to implement parallel /pipe line processing. , what kind of business problem can be solved by cyclic barrier and count down latch and how its working , how to use semaphore , whats CAS concept (Compare and set) , how atomic API is working internally

**Multithreading**

the expectation here is developer should know basic of multithreaing , should know how wait , notify , sleep , join is working , how locking is working , whats class label lock , whats object lock, how to implement inter thread communication by using wait and notify , how volatile is working , how happen before concept is working in terms of volatile , how to implement thread pool in java 4 , how important is immutability in multithreading , whats code can create deadlock , what code can create starvation,

**Serialization**

developer should know purpose of serialization , purpose of serial version UID , if serial version UID is not define how JVM generating it , how to customize serialization behavior , how to serialize transient variable how   to improve performance by customizing serialization behavior

**Memory management**

Developer should know java memory model , should know heap , how garbage collection is working , how to optimize memory , should aware where class meta data storing in memory , should know reason of Perm gen Exception , reason of Out of memory exception , should aware how to do memory profiling , how to identify which code consuming memory

**Design Pattern**

Developer should know at least 2 to 3 design pattern thoroughly, while explaining use case implementation should use some of the design pattern , must know best way of implementing singleton pattern , factory pattern

**Design Principle**

Developer should know SOLID concept very well , whenever explaining solution design principle should reflect in his solution, how important is code for interface concept

**Object oriented Concept**

Developer should know Encapsulation , Polymorphic , Composition , Inheritance , when should use inheritance when should we use composition

**Database**

Developer should able to write some of query on join and aggregation,Should aware index , type of index and how indexing is working , should aware all key

# RBS

## RBS client 1

1. Started with explain the architecture of your project. In this asked me the authentication strategy that we were using, how did we ensure that the request was secured through the network. How did you handle your exceptions ? Did you create your own custom exceptions if yes then why? Why do we need to catch exception and through it from each layer ? This was followed by a discussion on MVC.
2. Hashmap: Its internal working and how bucket index is evaluated. Asked to write a overridden hashcode and equals method. How to prevent a bad hash key. How does hash map handle if value of hashcode is more than the buckets. How does Hash Map works and question related to equals/hash code. Like what will happen if I return 1 from hash code ?
3. About if worked on Rest API.
4. Immutable classes, Semaphores, Concurrent hash map, Blocking queue Spring transaction: Concept, propagation behavior and isolation level.
5. Hibernate: Concept, Creating entity classes, hibernate configuration (session factory, transaction manager), fetch modes and hibernate caching.
6. Multithreading and Concurrency : Wait, notify, notifyAll, yield, sleep, join, synchronized, Executor, ExecutorService, Types of thread pools Fixed, cached, ReentrantLocks, ReadWriteLocks, Deadlock(its identification, resolution and prevention), Immutable classes(Code an immutable class)
7. Spring IOC : Bean creation and Injection strategies, bean scope
8. Spring AOP : Concept and use cases(Asked to write solution of a problem using AOP)
9. Design patterns : Chain of responsibility, Decorator
10. SQL : Asked to write queries, DB associations etc.
11. Abstract classes and interfaces : difference and use cases
12. Difference between Abstraction and Encapsulation.
13. Why Composition is preferred over inheritance
14. ConcurrentHashMap : Working and How segments are created for thread safe operations.
15. L1 L2 cache in hibernate
16. How to write Enum?

## RBS client 2

1. Architecture of last project, how was security managed.
2. Jenkins.
3. Asked if ever used Linked List in your project, if yes then tell the instance and why you used it.  Gave a scenario when we need to put less than 10 in a DS then which is better Linked List or Array List.
4. Maven plugins, goals, phase
5. Threading wait notify and concurrency package.
6. If I have a singleton Bean and inside it I have a prototype then what will happen.
7. Generics wildcards related
8. Countdown Latch
9. Autowiring in spring
10. Design Singleton Pattern and make it Reflection Safe and also safe if we are working in a distributed system.

## RBS client 3

Two rounds…

### 1st round

1. Project Architecture
2. In this asked question like y u used springs y not this and all these types.
3. Hash Map working
4. Questions around equal and hashcode.
5. Volatile
6. Thread Confinement
7. Connection pool
8. Thread Pool
9. Strategy Pattern
10. Garbage Collection
11. Locks
12. How would you identify memory leaks.
13. SQL indexing
14. Design custom Executor Framework

### 2nd Round

1. Mostly managerial
2. How do u handle conflicts
3. Explain project
4. What u know of RBS
5. How do u read requirement
6. How was ur last project managed
7. A program on Producer consumer
8. Deadlock
9. Any challenges faced tell
10. If I have 12 eggs and I give it to 12 people still I have one left with me how ?

## RBS interview question (First round)

Architecture of previous company project.

Design student management system with respect to restfull web services.

Difference between rest and soap web service.

Can we use soap in student management system?

Single iterator instance working with two thread where one thread will call remove and the other thread continue iteration over it then will be the case: throw concurrent modification exception or will work normally

Note: iterator is fail fast

Concurrent hash map working

Xml parsers, xslt working of xml beans

WAP a program on scenario based for countdownlatch and also implement the same without using countdown latch.

Hashcode and equals contract.

Difference notify() and notifyAll().

Design pattern

How to create singleton class.

Difference between singleton class and static singleton class.

Tool used in project development like Jenkins, Ant, Maven

## RBS interview question (Second round)

Tell me about yourself and project worked on.

Which is the most critical project you had worked and how you handle ?

Design Chess Game .

Probability question: Given 8 ball (4 white and 4 red) and two empty bags.

Place all the balls inside bag such that probability of finding the black ball from any of the bag is maximum.

# Topics

1. Hashmap and ConcurrentHashMap. Working and How segments are created for thread safe operations.
2. Immutable classes, Semaphores, Concurrent hash map, Blocking queue Spring transaction : Concept, propagation behavior and isolation level.
3. Hibernate: Concept, Creating entity classes, hibernate configuration (session factory, transaction manager), fetch modes and hibernate caching. Caching L1 and L2.
4. Multithreading and Concurrency : Wait, notify, notifyAll, yield, sleep, join, synchronized, Executor, ExecutorService, Types of thread pools Fixed, cached, ReentrantLocks, ReadWriteLocks, Deadlock(its identification, resolution and prevention), Immutable classes(Code an immutable class)
5. Spring IOC : Bean creation and Injection strategies, bean scope
6. Spring AOP : Concept and use cases(Asked to write solution of a problem using AOP)
7. Design patterns: Chain of responsibility, Decorator, Singleton with reflection safe, Strategy pattern.
8. SQL: Asked to write queries, DB associations etc.
9. Abstract classes and interfaces : difference and use cases.
10. Difference between Abstraction and Encapsulation.
11. Why Composition is preferred over inheritance.
12. How to write Enum?
13. Maven Goals phases
14. Threading wait notify and concurrency package.
15. If I have a singleton Bean and inside it I have a prototype then what will happen.
16. Generics wildcards related
17. Countdown Latch
18. Project Architecture
19. Volatile
20. Thread Confinement
21. Connection pool
22. Thread Pool
23. Garbage Collection
24. Locks
25. How would you identify memory leaks.
26. SQL indexing
27. Design custom Executor Framework
28. Mostly managerial
29. What u know of RBS
30. How do u read requirement
31. How was ur last project managed
32. A program on Producer consumer
33. Deadlock
34. REST/SOAP web services
35. Design student management system with respect to restfull web services.

# Knowledgebase

### Volatile variables

* Java volatile keyword is used to mark a Java variable as "being stored in main memory"
* A [volatile](http://www.javamex.com/tutorials/synchronization_volatile.shtml) variable is one whose value is always written to and read from "main memory. Thread does not cache this variable.
* Volatile: Used to indicate that the variable is modified by multiple threads. Inherently thread safe. No blocking is done when volatile variables are used.
* An access to a volatile variable **never has the potential to block**: we're only ever doing a simple read or write, so unlike a synchronized block we will never hold on to any lock.
* Can be used to implement counters in multithreaded environment.
* When a thread writes to a volatile variable, then not just the volatile variable itself is written to main memory. Also all other variables changed by the thread before writing to the volatile variable are also flushed to main memory. When a thread reads a volatile variable it will also read all other variables from main memory which were flushed to main memory together with the volatile variable.

### Static inner classes

* Only inner classes can be defined as static. Outer classes only contains static variables and methods.
* static simply means that the nested type does not need an instance of the enclosing type to be instantiated

### Thread confinement

* Two of the most useful tricks for ensuring thread safety are "stack confinement" and "thread confinement". In "**stack confinement**", we ensure that an object never escapes from a method. In "**thread confinement**", we only ever see a particular object from a single thread. Even if the object is not threadsafe, it now does not matter, since it is unshared.
* **Thread confinement** is the practice of ensuring that data is only accessible from one thread. Such data is called **thread-local** as it is local, or specific, to a single thread.  
    
  Thread-local data is thread-safe, as only one thread can get at the data, which eliminates the risk of races. And because races are nonexistent, thread-local data doesn't need locking. Thus thread confinement is a practice that makes your code safer (by eliminating a huge source of programming error) and more scalable (by eliminating locking).  
    
  Most languages don't have mechanisms to enforce thread confinement; it is a higher-level programming pattern and not a language or OS feature. Functionality such as thread local storage (TLS) makes thread confinement easier, but the programmer must still work to ensure references to the data does not escape the owning thread.
* Another common application of thread confinement is the use of pooled JDBC (Java Database Connectivity) Connection objects. The JDBC specification does not require that Connection objects be thread-safe.[[9]](http://techbus.safaribooksonline.com/0321349601/ch03lev1sec3#ch03fn09) In typical server applications, a thread acquires a connection from the pool, uses it for processing a single request, and returns it. Since most requests, such as servlet requests or EJB (Enterprise JavaBeans) calls, are processed synchronously by a single thread, and the pool will not dispense the same connection to another thread until it has been returned, this pattern of connection management implicitly confines the Connection to that thread for the duration of the request.

### Final keyword

2) What is the blank final field?

Uninitialized final field is called blank final field.

3) Can we change the state of an object to which a final reference variable is pointing?

Yes, we can change the state of an object to which a final reference variable is pointing, but we can’t re-assign a new object to this final reference variable.

4) What is the main difference between abstract methods and final methods?

Abstract methods must be overridden in the sub classes and final methods are not at all eligible for overriding.

5) What is the use of final class?

A final class is very useful when you want a high level of security in your application. If you don’t want inheritance of a particular class, due to security reasons, then you can declare that class as a final.

6) Can we change the value of an interface field? If not, why?

No, we can’t change the value of an interface field. Because interface fields, by default, are final and static. They remain constant for whole execution of a program.

7) Where all we can initialize a final non-static global variable if it is not initialized at the time of declaration?

In all constructors or in any one of instance initialization blocks.

8) What are final class, final method and final variable?

final class —> can not be extended.

final method —> can not be overridden in the sub class.

final variable —> can not change it’s value once it is initialized.

(Click [here](http://javaconceptoftheday.com/final-keyword-in-java/) for more info on final class, final method and final variable)

9) Where all we can initialize a final static global variable if it is not initialized at the time of declaration?

In any one of static initialization blocks.

10) Can we use non-final local variables inside a local inner class?

No. Only final local variables can be used inside a local inner class.

11) Can we declare constructors as final?

No, constructors can not be final

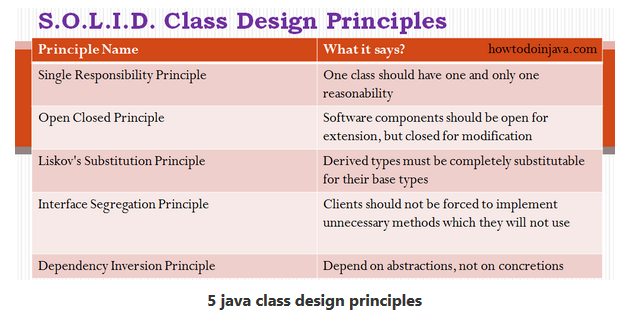
### CAS

This low-level JVMsupport is used by the atomic variable classes (AtomicXxx in java.util.concurrent. atomic) to provide an efficient CAS operation on numeric and reference types; these atomic variable classes are used, directly or indirectly, to implement most of the classes in java.util.concurrent

The approach taken by most processor architectures, including IA32 and Sparc, is to implement a *compare-and-swap* (CAS) instruction. (Other processors, such as PowerPC, implement the same functionality with a pair of instructions: *loadlinked* and *store-conditional*.) CAS has three operands—a memory location *V* on which to operate, the expected old value *A*, and the newvalue *B*. CAS atomically updates *V* to the new value *B*, but only if the value in *V* matches the expected old value *A*; otherwise it does nothing. In either case, it returns the value currently in *V*. (The variant called compare-and-set instead returns whether the operation succeeded.) CAS means “I think *V* should have the value *A*; if it does, put *B* there, otherwise don't change it but tell me I was wrong.” CAS is an optimistic technique—it proceeds with the update in the hope of success, and can detect failure if another thread has updated the variable since it was last examined. SimulatedCAS in [Listing 15.1](http://techbus.safaribooksonline.com/0321349601/ch15lev1sec2#ch15list01) illustrates the semantics (but not the implementation or performance) of CAS.

When multiple threads attempt to update the same variable simultaneously using CAS, one wins and updates the variable's value, and the rest lose. But the losers are not punished by suspension, as they could be if they failed to acquire a lock; instead, they are told that they didn't win the race this time but can try again. Because a thread that loses a CAS is not blocked, it can decide whether it wants to try again, take some other recovery action, or do nothing.[[3]](http://techbus.safaribooksonline.com/0321349601/ch15lev1sec2#ch15fn03) This flexibility eliminates many of the liveness hazards associated with locking

### SOLID design principle



### Composition over inheritance

* Code reuse
* Improved testability – Ability to mock has a relationships
* Many design patterns promote this – Strategy
* Flexibility
* Initial design is simplified by identifying system object behaviors in separate interfaces instead of creating a hierarchical relationship to distribute behaviors among business-domain classes via inheritance. This approach more easily accommodates future requirements changes that would otherwise require a complete restructuring of business-domain classes in the inheritance model.

### Generics

**Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

The most commonly used type parameter names are:

E - Element (used extensively by the Java Collections Framework)

K - Key

N - Number

T - Type

V - Value

S,U,V etc. - 2nd, 3rd, 4th types

Wildcard : In generic code, the question mark (?), called the wildcard, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a *list of unknown type*. There are two scenarios where an unbounded wildcard is a useful approach:

* If you are writing a method that can be implemented using functionality provided in the Object class.
* When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T

Lower Bounded Wildcards : List<? super Integer>

Wildcard Guidelines:

* An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
* An "out" variable is defined with a lower bounded wildcard, using the super keyword.
* In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
* In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

**Type Erasure**

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

* Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
* Insert type casts if necessary to preserve type safety.
* Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

To use Java generics effectively, you must consider the following restrictions:

* [Cannot Instantiate Generic Types with Primitive Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#instantiate)
* [Cannot Create Instances of Type Parameters](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects)
* [Cannot Declare Static Fields Whose Types are Type Parameters](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createStatic)
* [Cannot Use Casts or instanceof With Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCast)
* [Cannot Create Arrays of Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createArrays)
* [Cannot Create, Catch, or Throw Objects of Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCatch)
* [Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotOverload)

### Maven

There are three built-in build lifecycles: default, clean and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project's site documentation.

**A Build Lifecycle is Made Up of Phases**

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the [Lifecycle Reference](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference)):

* validate - validate the project is correct and all necessary information is available
* compile - compile the source code of the project
* test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package - take the compiled code and package it in its distributable format, such as a JAR.
* verify - run any checks on results of integration tests to ensure quality criteria are met
* install - install the package into the local repository, for use as a dependency in other projects locally
* deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

### Spring

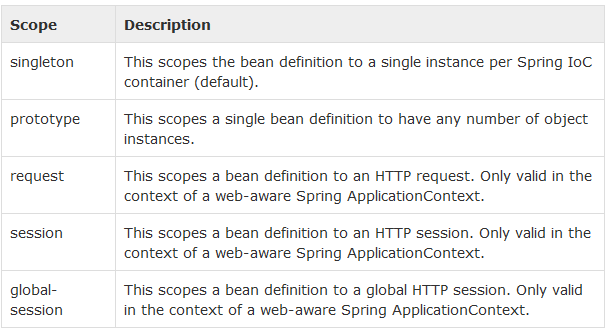
In Spring, 5 Auto-wiring modes are supported.

* no – Default, no auto wiring, set it manually via “ref” attribute
* byName – Auto wiring by property name. If the name of a bean is same as the name of other bean property, auto wire it.
* byType – Auto wiring by property data type. If data type of a bean is compatible with the data type of other bean property, auto wire it.
* constructor – byType mode in constructor argument.
* autodetect – If a default constructor is found, use “autowired by constructor”; Otherwise, use “autowire by type”.

In a multitier application, we will have different layers like presentation, service, business, data access etc. When a class is to be annotated for auto-detection by Spring, then we should use the respective stereotype as below.

* @Component – generic and can be used across application.
* @Service – annotate classes at service layer level.
* @Controller – annotate classes at presentation layers level, mainly used in [Spring MVC](http://javapapers.com/spring/spring-mvc-hello-world/).
* @Repository – annotate classes at persistence layer, which will act as database repository

**Spring Bean scope**



### Spring AOP

Common AspectJ annotations :

* **@Before** – Run before the method execution
* **@After** – Run after the method returned a result
* **@AfterReturning** – Run after the method returned a result, intercept the returned result as well.
* **@AfterThrowing** – Run after the method throws an exception
* **@Around** – Run around the method execution, combine all three advices above.

### Hibernate

**First-level cache** always Associates with the **Session object**. Hibernate uses this cache by default. Here, it processes one transaction after another one, means wont process one transaction many times. Mainly it reduces the number of SQL queries it needs to generate within a given transaction. That is instead of updating after every modification done in the transaction, it updates the transaction only at the end of the transaction.

**Second-level cache** always associates with the **Session Factory object**. While running the transactions, in between it loads the objects at the Session Factory level, so that those objects will be available to the entire application, not bound to single user. Since the objects are already loaded in the cache, whenever an object is returned by the query, at that time no need to go for a database transaction. In this way the second level cache works. Here we can use query level cache also.

**NOTE: Be aware that caches are not aware of changes made to the persistent store by another application. They can, however, be configured to regularly expire cached data.**

**Caching Strategies**

* read only : If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.
* Read/write: If the application needs to update data, a read-write cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required.
* Nonstrict read/write: If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.
* Transactional: The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager\_lookup\_class.

**Query Cache**

Query result sets can also be cached. This is only useful for queries that are run frequently with the same parameters.

This setting creates two new cache regions: one holding cached query result sets (org.hibernate.cache.StandardQueryCache), the other holding timestamps of the most recent updates to queryable tables (org.hibernate.cache.UpdateTimestampsCache). Note that the query cache does not cache the state of the actual entities in the result set; it caches only identifier values and results of value type. The query cache should always be used in conjunction with the second-level cache.

**Hibernate FetchMode vs FetchType**

FetchType (Lazy/Eager) tells whether we want entity to be loaded eagerly or lazy, when there's call in code.

FetchMode (Select/Join) tells whether we want our entitity to be loaded with additional select or in one query with join or subselect.

**Hibernate FetchMode**

* **fetch=”select” or @Fetch(FetchMode.SELECT)**
* **fetch=”join” or @Fetch(FetchMode.JOIN)**
* **batch-size=”10″ or @BatchSize(size = 10)**
* **fetch=”subselect” or @Fetch(FetchMode.SUBSELECT)**

### Java Memory Leaks

* Use visual VM
* Let application servers manage resources. Connection pooling etc.

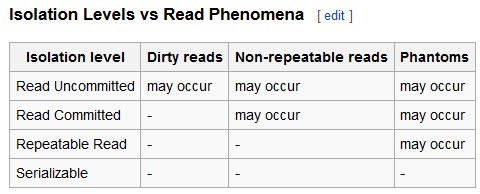
### Exception handling best practices

* Try not to create new custom exceptions if they do not have useful information for client code
* Log Exceptions just once
* Release resources in finally block
* Never use exceptions for flow control
* Do not suppress or ignore exceptions
* Document exceptions in methods javadoc
* Either log exceptions or throw them. Avoid doing both, as this results in logs at multiple places.
* Throw early and catch late principle: It basically says that you should throw an exception as soon as you can, and catch it late as much as possible. You should wait until you have all the information to handle it properly.
* Just remember that IO, parsing, and SQL exceptions are checked

### Spring JPA transaction propagation levels

|  |
| --- |
| [MANDATORY](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#MANDATORY)  Support a current transaction, throw an exception if none exists. |
| [NESTED](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#NESTED)  Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else. |
| [NEVER](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#NEVER)  Execute non-transactionally, throw an exception if a transaction exists. |
| [NOT\_SUPPORTED](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#NOT_SUPPORTED)  Execute non-transactionally, suspend the current transaction if one exists. |
| [REQUIRED](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#REQUIRED)  Support a current transaction, create a new one if none exists. |
| [REQUIRES\_NEW](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#REQUIRES_NEW)  Create a new transaction, and suspend the current transaction if one exists. |
| [SUPPORTS](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#SUPPORTS)  Support a current transaction, execute non-transactionally if none exists. |

### Transaction isolation levels



**Dirty read**

A *dirty read* (aka *uncommitted dependency*) occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.

**Non-repeatable read**

A *non-repeatable read* occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads.

**Phantom read**

A *phantom read* occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first.

### Java 8 new features

* Streams

Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements.

Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

Following are the characteristics of a Stream −

* + **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
  + **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
  + **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
  + **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
  + **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.
* Functional Interfaces

Functional interfaces have a single functionality to exhibit. For example, a Comparable interface with a single method ‘compareTo’ is used for comparison purpose. Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions. The list of functional interfaces is defined in java.util.Function package.

* Lambda expressions
* Java Time java.time

With Java 8, a new Date-Time API is introduced to cover the following drawbacks of old date-time API −

* + **Not thread safe** − java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.
  + **Poor design** − Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.
  + **Difficult time zone handling** − Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

Java 8 introduces a new date-time API under the package **java.time**. Following are some of the important classes introduced in java.time package −

* **Local** − Simplified date-time API with no complexity of timezone handling.
* **Zoned** − Specialized date-time API to deal with various timezones.

**Java 8 LocalDate, LocalDateTime, DateTimeFormatter are immutable classes and hence thread safe**

**NOTE : Hibernate 5 supports java 8 LocalDate and LocalDateTime same as java.util.Date.**

* Nashhorn
* Type annotations : List<@NotNullable String>
* Method References

Method references help to point to methods by their names. A method reference is described using **::** (double colon) symbol. A method reference can be used to point the following types of methods −

* Static methods
* Instance methods
* Constructors using new operator (TreeSet::new)

names.forEach(System.out::println);

* Default Methods

Interface can provide default mehods.

**interface** SampleInterface {

**default** **void** printEmployee() {

System.***out***.println("Sample Employee");

}

}

* Optional class

Optional is a container object which is used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values.

### CAP Theorem

In [theoretical computer science](https://en.wikipedia.org/wiki/Theoretical_computer_science), the **CAP theorem**, also named **Brewer's theorem** after computer scientist [Eric Brewer](https://en.wikipedia.org/wiki/Eric_Brewer_%28scientist%29), states that it is impossible for a [distributed computer system](https://en.wikipedia.org/wiki/Distributed_computing) to simultaneously provide all three of the following guarantees:[[1]](https://en.wikipedia.org/wiki/CAP_theorem#cite_note-Gilbert_Lynch-1)[[2]](https://en.wikipedia.org/wiki/CAP_theorem#cite_note-2)[[3]](https://en.wikipedia.org/wiki/CAP_theorem#cite_note-3)

* [*Consistency*](https://en.wikipedia.org/wiki/Consistency_%28database_systems%29) (every read receives the most recent write or an error)
* [*Availability*](https://en.wikipedia.org/wiki/Availability) (every request receives a response, without guarantee that it contains the most recent version of the information)
* [*Partition tolerance*](https://en.wikipedia.org/wiki/Network_partitioning) (the system continues to operate despite arbitrary partitioning due to network failures)

### BASE model

NoSQL databases, embrace situations where the ACID model is overkill or would, in fact, hinder the operation of the database. Instead, NoSQL relies upon a softer model known, appropriately, as the BASE model. This model accommodates the flexibility offered by NoSQL and similar approaches to the management and curation of unstructured data. BASE consists of three principles:

* **Basic Availability**. The NoSQL database approach focuses on availability of data even in the presence of multiple failures. It achieves this by using a highly distributed approach to database management. Instead of maintaining a single large data store and focusing on the fault tolerance of that store, NoSQL databases spread data across many storage systems with a high degree of replication. In the unlikely event that a failure disrupts access to a segment of data, this does not necessarily result in a complete database outage.
* **Soft State**. BASE databases abandon the consistency requirements of the ACID model pretty much completely. One of the basic concepts behind BASE is that data consistency is the developer's problem and should not be handled by the database.
* **Eventual Consistency**. The only requirement that NoSQL databases have regarding consistency is to require that at some point in the future, data will converge to a consistent state. No guarantees are made, however, about when this will occur. That is a complete departure from the immediate consistency requirement of ACID that prohibits a transaction from executing until the prior transaction has completed and the database has converged to a consistent state.

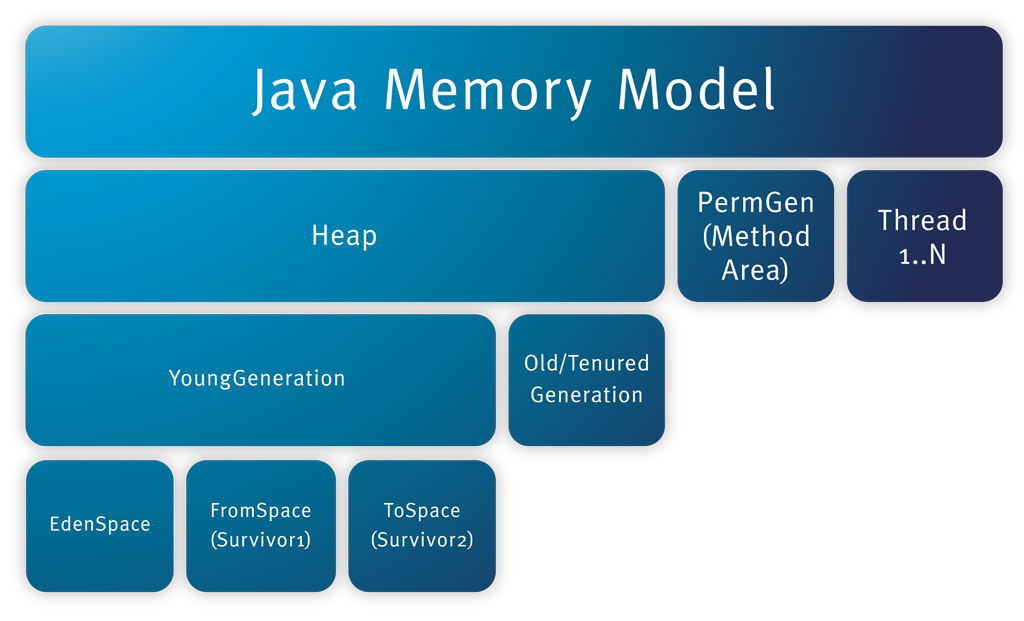
### Oracle Deadlocks

A deadlock occurs when two or more sessions are waiting for data locked by each other, resulting in all the sessions being blocked. **Oracle automatically detects and resolves deadlocks by rolling back the statement associated with the transaction that detects the deadlock**. Typically, deadlocks are caused by poorly implemented locking in application code.

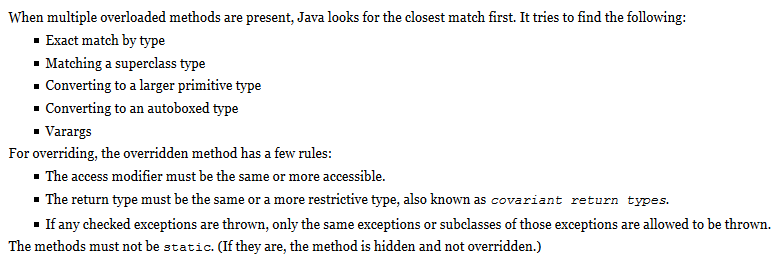
### Oracle

Oracle RAC allows multiple computers to run Oracle [RDBMS](https://en.wikipedia.org/wiki/Relational_database_management_system) software simultaneously while accessing a single [database](https://en.wikipedia.org/wiki/Database), thus providing [clustering](https://en.wikipedia.org/wiki/Computer_cluster).

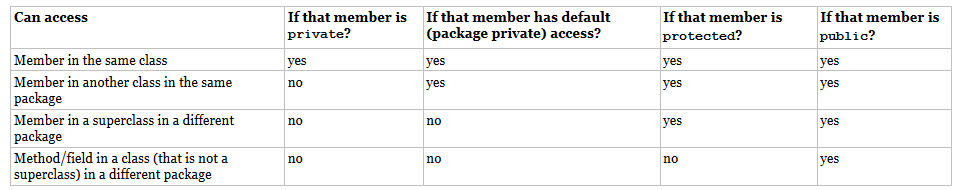
### JVM memory model

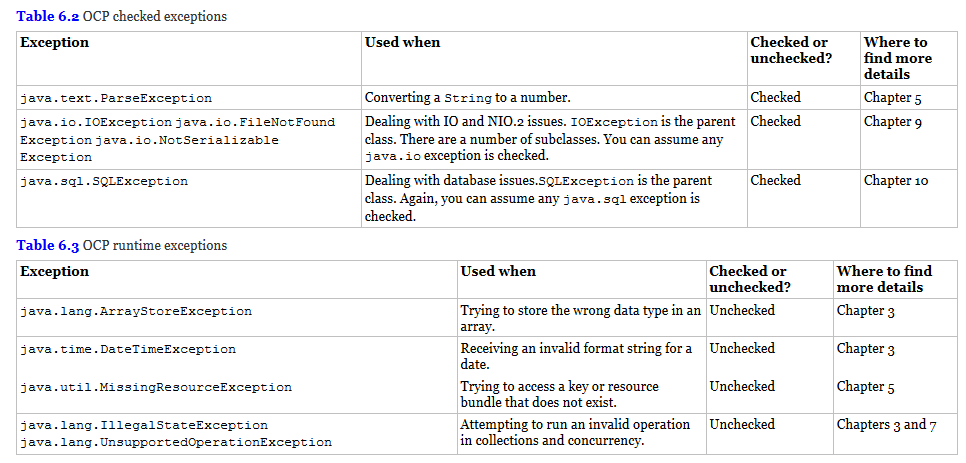


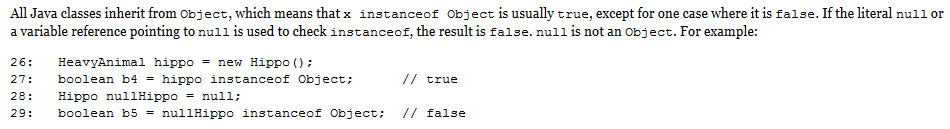
# JAVA

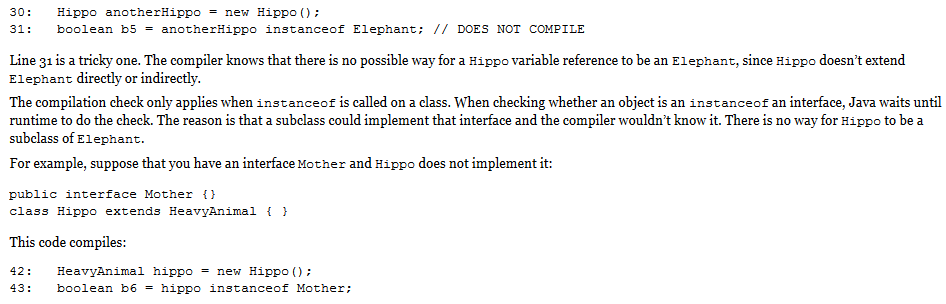


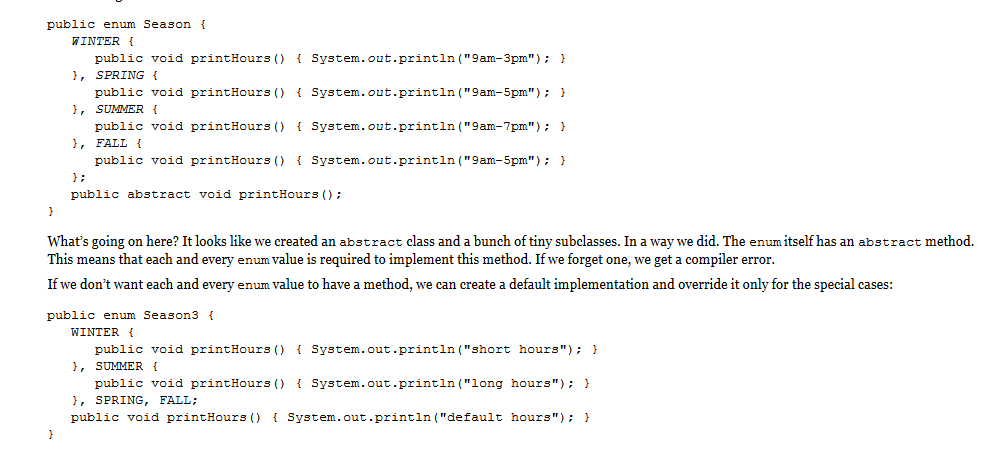
**An abstract** **class is not required to have any methods in it, let alone any abstract** **ones.**

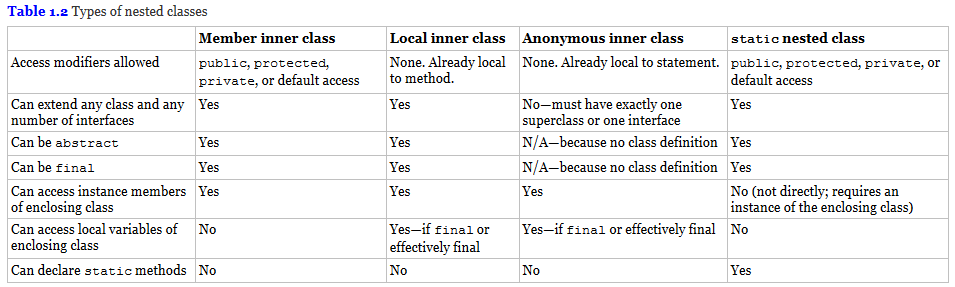












YES

YES