Day - 1 The JVM Architecture

## Features of JDK

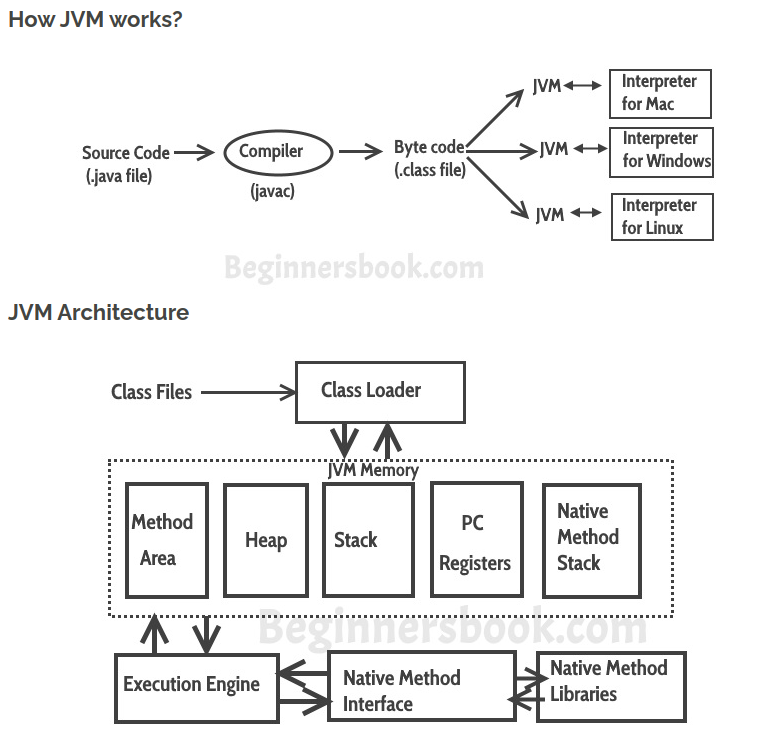
1. The JDK comes with a complete Java Runtime Environment (JRE), that is different from the regular JRE that's the reason it is usually called a private runtime so we can say that it includes all the features that JRE has.
2. It has all the Java development tools such as compiler, JVM, JRE, debugger etc.
3. You need JDK in order to write and run Java programs.
4. JDK supports multiple platforms and can be installed on Windows, Mac and other operating systems.

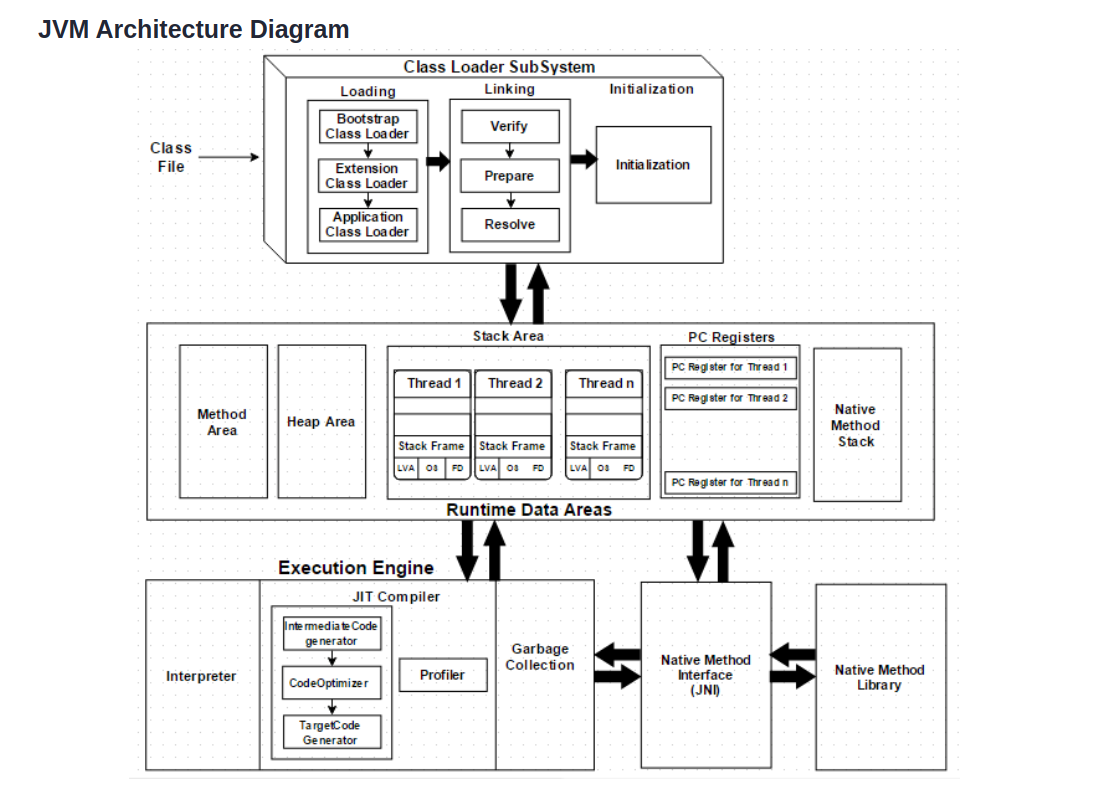
## Features of JRE

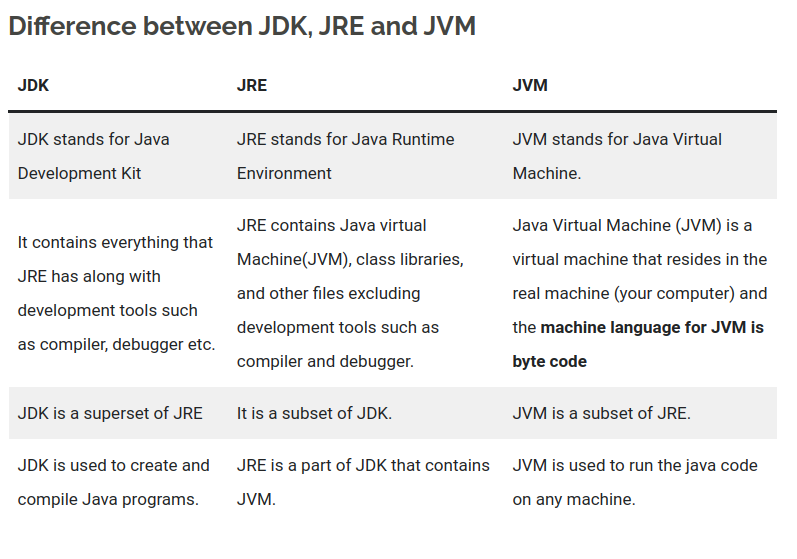
1. JRE contains a set of libraries and other tools that JVM needs at runtime.
2. You can easily run any Java program on JRE but you need JDK to write and compile Java programs.
3. JRE contains libraries that are required for integrations such as Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), etc.

## Features of JVM

1. JVM makes it possible to run Java code on any machine, it is the JVM that makes Java truly platform-independent.
2. It also allows running Java applications on cloud platforms.
3. JDK and JRE both of these contain JVM.
4. JVM is an interpreter as it executes the Java code line by line.
5. JVM converts the bytecode into machine code. JVM is platform independent as JVM doesn’t depend on the hardware and operating system of the machine







Method area and Heap area are not thread safe, it can access with any threads if systems need to create 10 thread, then only one Method and Heap Area will manage it.

### Resources:

1. <https://dzone.com/articles/jvm-architecture-explained>
2. [https://beginnersbook.com/2022/06/JDK-vs-jre-vs-jvm-difference-between-them/](https://beginnersbook.com/2022/06/jdk-vs-jre-vs-jvm-difference-between-them/)

DAY - 2 Java Memory Management & Garbage Collection

## Garbage Collection

#### 1. What is Garbage Collection in Java?

* Garbage Collection is an automatic memory management feature.
* The process of destroying unreferenced objects is called Garbage Collection.
* Once object is unreferenced it is considered as unused object, hence JVM automatically destroys that object.
* In Java developers, the responsibility is only to create objects and unreferencing those objects after usage.

#### 2. How JVM can destroy unreferenced objects?

* JVM internally uses a daemon thread called "garbage collector" to destroy all unreferenced objects.
* A daemon thread is a service thread. The Garbage Collector thread is called a daemon thread because it provides services to JVM to destroy unreferenced objects.
* This thread is low priority thread. Since it is a low-priority thread we can not guarantee this execution.

#### 3. So can you guarantee objects destruction?

* No, we can not guarantee object destruction even though it is unreferenced, because we can not guarantee garbage collector execution.
* So, we can confirm whether object is eligible for garbage collection or not.

#### 4. Can we force garbage collectors?

* No, we can not force garbage collectors to destroy objects, but we can request it.

#### 5. How can we request JVM to start the garbage collection process?

* We have a method called **GC()** in the system class as static method and also in Runtime class as a non-static method to request JVM to start garbage collector execution.
* System.gc();
* Runtime.getRuntime().gc();

#### 6. What is the algorithm JVM internally uses for destroying objects?

* "mark and swap" is the algorithm JVM internally uses.

#### 7. Which part of the memory is involved in Garbage Collection?

* Heap.

#### 8. What is the responsibility of the Garbage Collector?

* Garbage Collector frees the memory occupied by the unreachable objects during the Java program by deleting these unreachable objects.
* It ensures that the available memory will be used efficiently, but does not guarantee that there will be sufficient memory for the program to run.

#### 9. When does an object become eligible for garbage collection?

* An object becomes eligible for garbage collection when no live thread can access it.

#### 10. What are the different ways to make an object eligible for garbage collection when it is no longer needed?

* Set all available object references to "null" once the purpose of creating an object is served.

#### 11. What is the purpose of the overriding finalize() method?

* The finalize() method should be overridden for an object to include the clean-up code or to dispose of the system resources that should be done before the object is garbage collected.

#### 12. How many times does the garbage collector calls the finalize() method for an object?

* Only once.

#### 13. What happens if an uncaught exception is thrown during the execution of finalize() method of an object?

* The exception will be ignored and the garbage collection (finalization) of that object terminates

#### 14. What are the different ways to call a garbage collector?

* System.gc();
* Runtime.getRuntime().gc();

#### 15. How to enable /disable the call of finalize() method of exit of the application?

* Runtime.getRuntime().runFinalizersOnExit(boolean value). passing the boolean value true and false will enable or disable the finalize() call.

### Resources:

1. [http://www.instanceofJava.com/2015/03/garbage-collections-interview-questions.html?fbclid=IwAR0m1dlifG8OFB2geG0nHju\_o7SREETAtz9TwDr2PxUD\_TlCIVgjQ1S56dM#:~:text=Top%2015%20Garbage%20Collection%20Interview%20Questions%201%201.What,is%20responsibility%20of%20Garbage%20Collector%3F%20...%20More%20items](http://www.instanceofjava.com/2015/03/garbage-collections-interview-questions.html?fbclid=IwAR0m1dlifG8OFB2geG0nHju_o7SREETAtz9TwDr2PxUD_TlCIVgjQ1S56dM#:~:text=Top%2015%20Garbage%20Collection%20Interview%20Questions%201%201.What,is%20responsibility%20of%20Garbage%20Collector%3F%20...%20More%20items)
2. [https://dzone.com/articles/Java-memory-management](https://dzone.com/articles/java-memory-management)

DAY - 3 String Constant Pool

#### String is mutable

#### String Buffer and String Builder are immutable but StringBuilder is thread safe but string buffer is not. That’s why string builder is faster.

1. The string is secure because it cannot be changed if it is already assigned

#### Advantages of String Pool in Java

Java String Pool allows caching of strings. Caching here is the process of storing data in a cache. Cache improves performance and reduces memory usage.

Provides reusability: It saves time to create a new string if there is already a string with the same value present in the pool. The old string is reused, and its reference is returned.

Note:

String interning is a method of storing only one copy of each distinct string value, which must be immutable. In Java, String interning is achieved using the concept of String Pool.

#### Disadvantages of Using String Objects

Strings have a constant value, and even if they are altered, instead of reflecting the changes in the original string, a new object is created.

This causes a lot of objects to be created in the heap and wastes a lot of memory if the user keeps on updating the value of the string.

### **Is String immutable or final in Java? If so, then what are the benefits of Strings being Immutable?**

Yes, Strings are immutable in Java. Immutable objects mean they can't be changed or altered once they've been created. However, we can only modify the reference to the string object. The String is immutable in Java because of many reasons like security, caching, synchronization and concurrency, and class loading.

### **State the difference between StringBuffer and StringBuilder in Java.**

StringBuffer and StringBuilder are two Java classes for manipulating strings. These are mutable objects, i.e., they can be modified, and provide various methods such as insert(), substring(), delete(), and append(), for String manipulation.

* StringBuffer: The StringBuffer class was created by the Java Team when they realized the need for an editable string object. Nevertheless, StringBuffer has all methods synchronized, meaning they are thread-safe. Therefore, StringBuffer allows only one thread to access a method at once, so it is not possible to call StringBuffer methods from two threads simultaneously, which means it takes more time to access. The StringBuffer class has synchronized methods, making it thread-safe, slower, and less efficient than StringBuilder. The StringBuffer class was introduced in Java 1.0.
  + Syntax:

StringBuffer var = new StringBuffer(str);

* StringBuilder: It was at that point that the Java Team realized that making all methods of StringBuffer synchronized wasn't the best idea, which led them to introduce StringBuilder. The StringBuilder class has no synchronized methods. Unlike StringBuffer, StringBuilder does not offer synchronized methods, which makes it less thread-safe, faster, and more efficient. StringBuilder was introduced in Java 1.5 in response to StringBuffer's shortcomings.
  + Syntax:

StringBuilder var = new StringBuilder(str);

### **What is the difference between str1 == str2 and str1.equals(str2)?**

Java offers both the equals() method and the "==" operator for comparing objects. However, here are some differences between the two:

* Essentially, equals() is a method, while == is an operator.
* The == operator can be used for comparing references (addresses) and the .equals() method can be used to compare content. To put it simply, == checks if the objects point to the same memory location, whereas .equals() compares the values of the objects.

### 

### **Is it possible to compare Strings using the == operator? If so, what is the risk involved?**

Yes, you can compare strings using the == operator. One can use == operators for reference comparison (address comparison). The majority of the time, developers compare strings with the == operator, instead of using the equals() method, resulting in an error.

### **Is String thread-safe in Java?**

Strings are immutable objects, which means they can't be changed or altered once they've been created. As a result, whenever we manipulate a String object, it creates a new String rather than modifying the original string object. In Java, every immutable object is thread-safe, which means String is also thread-safe. As a result, multiple threads can access a string. For instance, if a thread modifies the value of a string, instead of modifying the existing one, a new String is created, and therefore, the original string object that was shared among the threads remains unchanged.

### **Why char array is preferred over a String in storing passwords?**

There are various reasons why a char array rather than a string should be used to store passwords. The following are a few of them:

* Strings are immutable: The content of Strings cannot be modified/overwritten because any modification will result in the creation of a new String. As a result, we should always save sensitive data like passwords, Social Security numbers, and so on in a char[] array rather than a String.
* Security: Because String is immutable, storing the password as plain text keeps it in memory until it is cleaned up by the garbage collector. As string uses SCP (String Constant Pool) for re-usability of a string, it's possible that it'll remain in memory for a long time, and anyone with access to the SCP or memory dump can simply identify or retrieve the password in plain text. That's another reason why we should use an encrypted password instead of plain text.
* Logfile safety: With an array, the data can be erased or wiped up, overwritten and the password will not be present anywhere in the system. Whereas, when using plain String, the chances of mistakenly printing the password to monitors, logs, or other insecure locations are substantially higher.

### **How many objects will be created for the following codes:**

A.

String str1 = "abc"; //Line1

String str2 = **new** String("abc"); //Line2

B.

String str1 = "abc"; //Line1

String str2 = "abc"; //Line2

C.

String str1 = **new** String("abc"); //Line1

String str2 = **new** String("abc"); //Line2

* For A: In this case, two objects will be created. We know that whenever a Java string is created using a new keyword, then two objects will be created i.e. one in the Heap Area and another one in the String constant pool. When the line1 is executed, the new string object str1 gets created and stored in the string constant pool. However, when line2 is executed, only one object is created using a new operator that gets stored in the heap memory (str2). This is because String constant pool already has a String object with the same string value (abc), and therefore, the reference of the string str1 from the string constant pool is returned.
* For B: In this case, one object will be created. Here, for line1 (str1), one new object will get created in String constant pool, whereas for line 2, string str2 will create a reference to the String str1 because the string constant pool already has a String object str1 with the same string value (abc).
* For C: In this case, three objects will be created. In the case of line1 (str1), two objects are created, one in the string constant pool and one in the heap memory. As for line 2 (str2), one new object is created and stored in heap memory, but not in the string constant pool because a String constant pool object str1 already has the string object str1 with the same string value (abc).

### What is meant by String Pool in Java?

String Pool in Java is a storage space in Java Heap memory where unique string literals are stored.

### Where is the String Pool stored?

String Pool is stored in the Heap Memory.

### Why do we need String Pool in Java?

It is created to decrease the number of string objects created in the memory. Whenever a new string is created, JVM first checks the string pool. If it encounters the same string, then instead of creating a new string, it returns a reference existing string to the variable.

### Does string pool make Java more memory efficient?

Yes, a string pool helps to save memory by preserving immutable strings in a pool so that the instances can be reused.

### Resources:

1. [https://www.scaler.com/topics/Java/string-pool-in-Java/?fbclid=IwAR1Q8LX4VIvqhyddCcD6PXw93DTbTguWss1iZmcqtxwqPHx-2JvZboGTWFo](https://www.scaler.com/topics/java/string-pool-in-java/?fbclid=IwAR1Q8LX4VIvqhyddCcD6PXw93DTbTguWss1iZmcqtxwqPHx-2JvZboGTWFo)
2. [https://www.interviewbit.com/Java-string-interview-questions/](https://www.interviewbit.com/java-string-interview-questions/)

DAY - 4 ACID

-> Atomicity:: A transaction must function as a single indivisible unit of work so that the entire transaction is either applied or rolled back. When transactions are atomic, there is no such thing as a

partially completed transaction: it’s all or nothing.

-> Consistency:: The database should always move from one consistent state to the next.

-> Isolation:: transactions occur independently without interference.

-> Durability:: Once committed, a transaction’s changes are permanent. This means the changes must be recorded such that data won’t be lost in a system crash.

Isolation Levels:

-------------

@Transactional (isolation=Isolation.READ\_COMMITTED)

The default is Isolation.DEFAULT

DEFAULT: Use the default isolation level of the underlying database.

READ\_COMMITTED: A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.

READ\_UNCOMMITTED: This isolation level states that a transaction may read data that is still uncommitted by other transactions.

REPEATABLE\_READ: A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.

SERIALIZABLE: A constant indicating that dirty reads, non-repeatable reads, and phantom reads are prevented.

What do these Jargon's dirty reads, phantom reads, or repeatable reads mean?

Dirty Reads: Transaction "A" writes a record. Meanwhile, Transaction "B" reads that same record before Transaction A commits. Later, Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ\_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.

Non-Repeatable Reads: Transaction "A" reads some record. Then Transaction "B" writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.

Phantom Reads: Transaction "A" reads a range of records. Meanwhile, Transaction "B" inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

--> postgress default isolation level is Read Committed.

--> mysql default isolation level is Repeatable Read.

@Transactional(timeout=60)

Defaults to the default timeout of the underlying transaction system.

--> mysql default transaction timeout is 50 sec

--> postgres default transaction timeout need to set in postgrsql.conf file

Propagation:

------------

@Transactional(propagation=Propagation.REQUIRED)

If not specified, the default propagational behavior is REQUIRED.

Other options are REQUIRES\_NEW , MANDATORY , SUPPORTS , NOT\_SUPPORTED , NEVER , and NESTED .

REQUIRED

Indicates that the target method cannot run without an active txn. If active txn has already been started before the invocation of this method, then it will continue in the same txn or a new txn would begin soon as this method is called.

REQUIRES\_NEW

Indicates that a new txn has to start every time the target method is called. If already active txn is going on, it will be suspended before starting a new one.

MANDATORY

Indicates that the target method requires an active txn to be running. If active txn is not going on, it will fail by throwing an exception.

SUPPORTS

Indicates that the target method can execute irrespective of active txn. If active txn is running, it will participate in the same txn. If executed without a txn it will still execute if no errors.

Methods which fetch data are the best candidates for this option.

NOT\_SUPPORTED

Indicates that the target method doesn’t require the transaction context to be propagated.

Mostly those methods which run in a transaction but perform in-memory operations are the best candidates for this option.

NEVER

Indicates that the target method will raise an exception if executed in a transactional process.

This option is mostly not used in projects.

PROPAGATION\_NESTED

uses a single physical transaction with multiple savepoints that it can roll back to. Such partial rollbacks allow an inner transaction scope to trigger a rollback for its scope, with the outer transaction being able to continue the physical transaction despite some operations having been rolled back. This setting is typically mapped onto JDBC savepoints, so will only work with JDBC resource transactions.

@Transactional (rollbackFor=Exception.class)

-----------------------------------------

Default is rollbackFor=RunTimeException.class

In Spring, all API classes throw RuntimeException, which means if any method fails, the container will always rollback the ongoing transaction.

The problem is only with checked exceptions. So this option can be used to declaratively rollback a transaction if Checked Exception occurs.

@Transactional (noRollbackFor=IllegalStateException.class)

-----------------------------------------

Indicates that a rollback should not be issued if the target method raises this exception.

### Resources:

1. <https://www.youtube.com/watch?v=TpLOTXd7djw>

DAY -5 SOLID, Bean Life cycle

## 1. Single responsibility principle:

According to the single responsibility principle, each Java class must perform only one function. Multiple capabilities in a single class mash up the code, and any changes made to the code may influence the entire class. It specifies the code and makes it simple to maintain.

**In other words, Every class should have one and only reason to change it.**

Let’s understand this by using an example:

Suppose we have a \*\*class FinalExam\*\* that has 3 methods that perform 3 operations

```c

AddQuestion(), ExpectedAnswer(), Marksdistribution()

```

Now all these 3 methods perform different actions. By using the single responsibility principle, we can separate these functionalities into three separate classes to fulfil the goal of the principle.

**Why is this Principle Required?**

Testing is made easier when the Single Responsibility Principle is implemented. The class will have fewer test cases with a single responsibility. Because there is less functionality, there are fewer dependencies on other classes. Because smaller, well-purposed classes are easier to search, it leads to better code organization.

## 

## 2. Open-Closed Principle

**Open Close Principle (OCP) states or ensures that A class, component or entity should be open for extension but close for modification. In detail, We can extend any class via Interface, Inheritance or Composition whenever it's required instead of opening a class and modifying it's code.**

According to this idea, a class should be created in such a way that it does its task flawlessly without the expectation that it will be changed in the future. As a result, the class should stay closed to alteration, but it should be possible to extend it. Extending the class can be done in a variety of ways, including

1. Inheriting from class.

2. Overwriting the required behaviour from the class.

3. Extending certain behaviour of the class.

**Why is this principle required?**

Because classes may come from third-party libraries, OCP is essential. We should be able to extend those classes without having to worry about whether or not the base classes will be able to support our expansions. However, inheritance may result in subclasses that are dependent on the implementation of the base class. The use of interfaces is recommended to avoid this. Loose coupling occurs as a result of the added abstraction.

## 3. Liskov substitution principle

Barbara Liskov proposed the Liskov Substitution Principle (LSP). **It pertains to inheritance in the sense that derived classes must be 100% interchangeable with their base classes**

**Liskov Substitution Principle (LSP) states that Objects in a program can be replaced by the instances of their subtypes without modifying the correctness of a program. In other words, if A is subtype of B then instances of B may be replaced by the instances of A without altering the program correctness**

It goes beyond the open-close principle to look at how a superclass and its subclasses behave. Unless there is a compelling reason to do otherwise, we should build the classes to preserve the property. Let’s understand it with an example:

*```Java*

*public class Rectangle {*

*private double height;*

*private double width;*

*public void setHeight(double h) { height = h; }*

*public void setWidht(double w) { width = w; }*

*...*

*}*

*public class Square extends Rectangle {*

*public void setHeight(double h) {*

*super.setHeight(h);*

*super.setWidth(h);*

*}*

*public void setWidth(double w) {*

*super.setHeight(w);*

*super.setWidth(w);*

*}*

*}*

```

**Why is this principle is required?**

This prevents inheritance from being abused. It assists us in adhering to the “is-a” relationship. Subclasses must also adhere to a contract established by the base class. In this way, it’s similar to Bertrand Meyer’s Design by Contract. It’s easy to think of a circle as a form of an ellipse, yet circles lack two foci or major/minor axes.

## 

## 4. Interface segregation principle

**Interface Segregation Principle (ISP) states that use many client specific interfaces instead of one general purpose interface. In other words No client should be forced to implement other methods which it does not require. It means it's better to create a separate interface and allow your classes to implement multiple interfaces.**

Suppose there’s an interface for a vehicle and a Bike class:

*```Java*

*public interface Vehicle {*

*public void drive();*

*public void stop();*

*public void refuel();*

*public void openDoors();*

*}*

*public class Bike implements Vehicle {*

*// Can be implemented*

*public void drive() {...}*

*public void stop() {...}*

*public void refuel() {...}*

*// Can not be implemented*

*public void openDoors() {...}*

*}*

*```*

As you can see, a Bike class should not implement the openDoors() method because a bike does not have any doors! To address this, ISP offers to break down the interfaces into several, small coherent interfaces so that no class is required to implement any interfaces (and thus methods) that it does not require.

*```Java*

*public interface Driving {*

*public void drive();*

*}*

*public interface stops{*

*public void stop();*

*}*

*public interface refueling{*

*public void refuel();*

*}*

*public interface opening{*

*public void openDoors();*

*}*

*public class Bike implements Driving,stops,refueling {*

*public void drive() {...}*

*public void stop() {...}*

*public void refuel() {...}*

*}*

*```*

***Why is this principle required?***

The Interface Segregation Principle makes our code more readable and maintainable. We’ve pared down our class implementation to simplify the operations that are required, with no extra or extraneous code. As we only use that method that is required and can leave other methods.

## 5. Dependency Inversion Principle

**High-level modules should not depend on low-level modules. Instead, they should both depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.**

The Dependency Inversion Principle (DIP) asserts that rather than real implementations, we should rely on abstractions (interfaces and abstract classes) (classes). Details should not be dependent on abstractions; rather, abstractions should be dependent on details.

Let’s understand it by using an example:

We have a class WindowMachine that contains keyboard and monitor classes inside it and we create instances of monitor and keyboard classes inside WindowMachine constructor to use them.

## Hibernate N+1 Problem

Hibernate N+1 problem occurs when you use FetchType.LAZY for your entity associations. If you perform a query to select n-entities and if you try to call any access method of your entity's lazy association, Hibernate will perform n-additional queries to load lazily fetched objects

*public class Author {*

*@Id*

*@GeneratedValue(strategy = GenerationType.IDENTITY)*

*private Integer id;*

*private String fullName;*

*@OneToMany(fetch = FetchType.LAZY)*

*private Set<Book> books;*

*}*

Let’s try to load all authors and print each author’s name with his books collection size:

*entityManager.createQuery("select a from Author a", Author.class)*

*.getResultList()*

*.forEach(a -> System.out.printf("%s had written %d books\n",*

*a.getFullName(), a.getBooks().size()));*

The first query Hibernate will generate is to select all authors:

*SELECT author0\_.id AS id1\_0\_,*

*author0\_.fullName AS fullname2\_0\_*

*FROM authors author0\_;*

After that, when we call size() method on the books collection, this association needs to be initialized, so Hibernate will perform an additional query:

*SELECT books0\_.author\_id AS author\_i4\_1\_0\_,*

*books0\_.id AS id1\_1\_0\_,*

*books0\_.id AS id1\_1\_1\_,*

*books0\_.author\_id AS author\_i4\_1\_1\_,*

*books0\_.title AS title2\_1\_1\_,*

*books0\_.year AS year3\_1\_1\_*

*FROM books books0\_*

*WHERE books0\_.author\_id=?;*

Hibernate provides a couple of ways to eliminate this issue:

### 1. The first solution is to use join fetch:

entityManager.createQuery("select a from Author a left join fetch a.books", Author.class);

*SELECT author0\_.id AS id1\_0\_0\_,*

*books1\_.id AS id1\_1\_1\_,*

*author0\_.fullName AS fullname2\_0\_0\_,*

*books1\_.author\_id AS author\_i4\_1\_1\_,*

*books1\_.title AS title2\_1\_1\_,*

*books1\_.year AS year3\_1\_1\_,*

*books1\_.author\_id AS author\_i4\_1\_0\_\_,*

*books1\_.id AS id1\_1\_0\_\_*

*FROM authors author0\_*

*LEFT OUTER JOIN books books1\_ ON author0\_.id=books1\_.author\_id;*

### 2. Another way is to use @BatchSize on the lazy association:

*public class Author {*

*@OneToMany(fetch = FetchType.LAZY, mappedBy = "author")*

*@BatchSize(size = 10)*

*private Set<Book> books;*

*}*

This query will be called N/M times, where N is the number of authors and M is the specified batch size. Totally we will call N/M+1 queries.

### 3. The third way is to use a sub query returning a list of author identifiers

Hibernate provides this opportunity by setting @Fetch(FetchMode.SUBSELECT) on the lazy association:

*public class Author {*

*@OneToMany(fetch = FetchType.LAZY, mappedBy = "author")*

*@Fetch(FetchMode.SUBSELECT)*

*private Set<Book> books;*

*}*

## QNA

1. **What are the advantages of following SOLID principles in software development?**

Maintainability:

Flexibility:

Testability:

### Resources:

SOLID

1. <https://www.youtube.com/watch?v=BM_lSZPMClo>
2. [https://medium.com/@Javatechie/solid-design-principle-Java-ae96a48db97](https://medium.com/@javatechie/solid-design-principle-java-ae96a48db97)

DAY - 6 OOP

## 1. Object

An object can be represented as an entity that has state and behavior. For example A car is an object that has states such as color, model, price and behaviour such as speed, start, gear change, stop etc.

**Examples of states and behaviors**

Example 1:

Class: House

State: address, color, area

Behaviour: Open door, close door

Example 2:

Class: Car

State: color, brand, weight, model

Behavior: Break, Accelerate, Slow Down, Gear change.

## 2. Class

**A class can be considered as a blueprint that you can use to create as many objects as you like.**

### Resources:

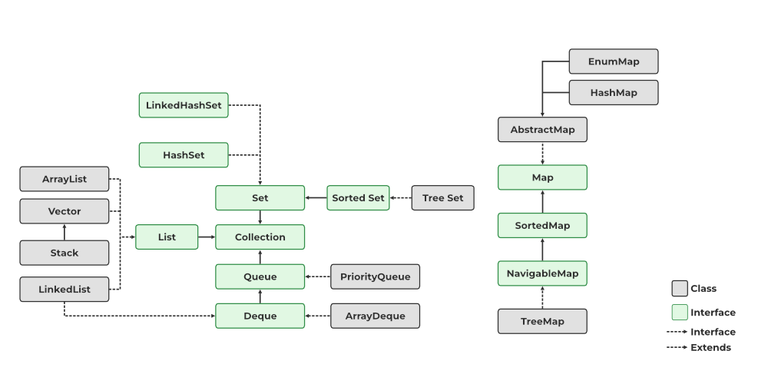
DAY - 10 Collection Framework

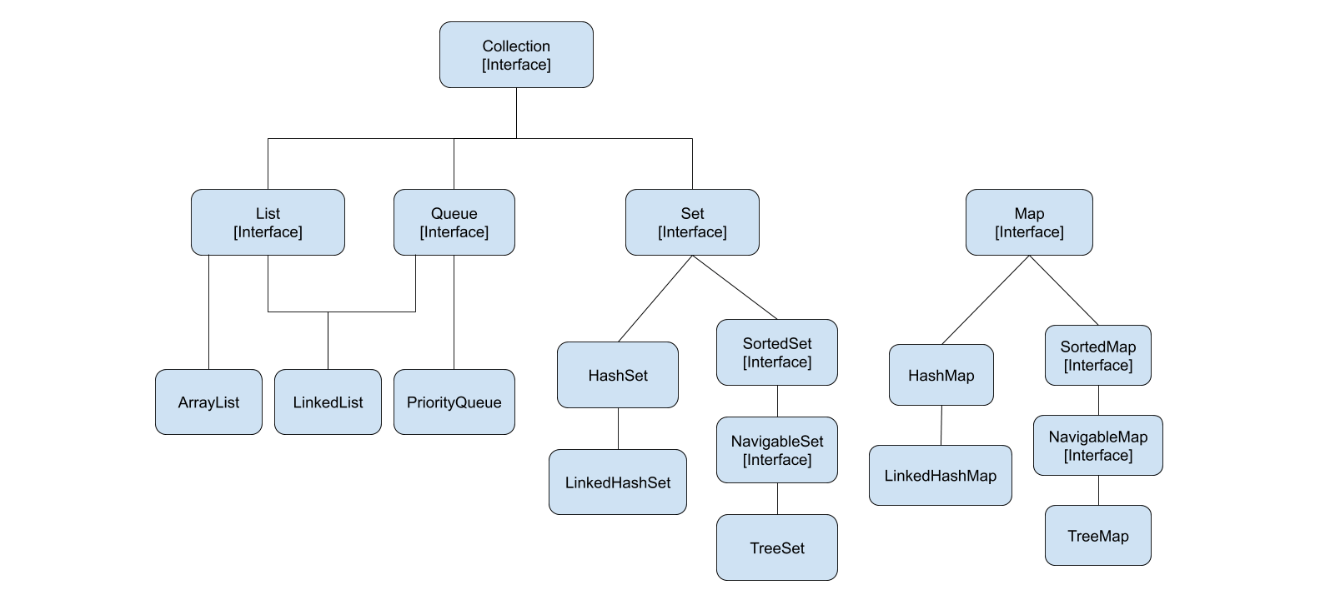
## Collections Framework Vs. Collection Interface

The Collection interface is the root interface of the collections framework. The framework includes other interfaces as well: **Map and Iterator**. These interfaces may also have subinterfaces.

## Advantages of the Collection Framework

1. Consistent API: The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
2. Reduces programming effort: A programmer doesn’t have to worry about the design of the Collection but rather he can focus on its best use in his program
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms





**Class:** A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

**Interface**: Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, nobody). Interfaces specify what a class must do and not how. It is the blueprint of the class.

## The classes which implement the List interface are as follows:

1. ArrayList
2. LinkedList
3. Vector
4. Stack

## 

## FAQs in Collections Framework?

1. Why Map interface doesn’t extend the Collection interface?

Although Map interface and its implementations are part of the Collections Framework, Map is not collections and collections are not Map. Hence it doesn't make sense for Map to extend Collection or vice versa. If Map extends Collection interface, then where are the elements? The map contains key-value pairs and it provides methods to retrieve the list of Keys or values as Collection but it doesn't fit into the "group of elements" paradigm.

1. What is difference between Enumeration and Iterator interface?

Enumeration is twice as fast as Iterator and uses very little memory. Enumeration is very basic and fits basic needs. But the Iterator is much safer as compared to Enumeration because it always denies other threads to modify the collection object which is being iterated by it. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection which is not possible with Enumeration. Iterator method names have been improved to make their functionality clear.

1. What is difference between fail-fast and fail-safe?

Iterator fail-safe property work with the clone of underlying collection, hence it's not affected by any modification in the collection. By design, all the collection classes in `java.util` package are fail-fast whereas collection classes in `java.util.concurrent` are fail-safe. Fail-fast iterators throw ConcurrentModificationException whereas fail-safe iterator never throws ConcurrentModificationException.

Fail-fast behavior is commonly found in collections like **ArrayList, HashSet, HashMap,** etc., which do not provide guarantees about the behavior of iterators if the underlying collection is modified during iteration

Fail-safe iterators are designed to continue the iteration without any interruption, even if the underlying data structure is modified during iteration. This approach aims to provide consistency and stability during concurrent access.

Fail-safe behavior is commonly found in collections like **CopyOnWriteArrayList, ConcurrentHashMap,** etc., which are specifically designed to handle concurrent modifications gracefully.

1. Why can’t we write code as **List<Number> numbers = new ArrayList<Integer>();?**

Generics doesn't support sub-typing because it will cause issues in achieving type safety. That's why List<T> is not considered as a subtype of List<S> where S is the super-type of T. To understanding why it's not allowed, let's see what could have happened if it has been supported.

```

List<Long> listLong = new ArrayList<Long>();

listLong.add(Long.valueOf(10));

List<Number> listNumbers = listLong; // compiler error

listNumbers.add(Double.valueOf(1.23));

```

## Wrapper Classes in Java

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

### Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a **method (because primitive types are passed by value).**
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

### Advantages of Wrapper Classes

1. Collections allowed only object data.
2. On object data we can call multiple methods compareTo(), equals(), toString()
3. Cloning process only objects
4. Object data allowed null values.
5. Serialization can allow only object data.

### Autoboxing and Unboxing

1. Autoboxing

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

2. Unboxing

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

### FAQs in Wrapper Class

1. Which are the wrapper classes in Java?

A Wrapper class in Java is a class whose object wraps or contains primitive data types.

2. Why use the wrapper class in Java?

The wrapper class in Java is used to convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method.

3. What are the 8 wrapper classes in Java?

There are 8 Wrapper classes in Java these are **char, byte, short, int, long, float, double and boolean.**

### Resources:

1. <https://www.geeksforgeeks.org/collections-in-java-2/>
2. <https://www.digitalocean.com/community/tutorials/java-collections-interview-questions-and-answers>
3. <https://medium.com/javarevisited/50-java-collections-interview-questions-for-beginners-and-experienced-programmers-4d2c224cc5ab>
4. [**https://javaconceptoftheday.com/java-collections-cheat-sheet/**](https://javaconceptoftheday.com/java-collections-cheat-sheet/)