Java Memory Structure:

Java's memory management is primarily handled by JVM, through a combination of automatic memory allocation, Garbage collection and Memory optimization strategies.

Automatic memory allocation:

- In java, memory allocation for object is managed automatically by the JVM.
- When we create an object using new keyword, memory is allocated on the heap and JVM tracks allocation and deallocation of memory for that object

Garbage Collection:

- Java uses automatic garbage collection to reclaim the memory occupied by the object that are no longer reachable or in use.
- JVM periodically runs garbage collector to identify and remove the objects that are no longer referenced by any active part of the program.

Memory Optimization Strategies:

Java optimizes memory usage through techniques like:

- Object Pooling: Reusing objects to reduce memory allocation overhead.
- String Pool: Reusing string literals to conserve memory.
- Generational Garbage Collection: Dividing heap memory into generations (young and old) for more efficient garbage collection.

Memory Leaks:

- While Java manages memory automatically, improper handling of resources (like not closing I/O streams) can lead to memory leaks.
- Memory leaks occur when objects are unintentionally kept in memory due to lingering references, preventing the garbage collector from reclaiming them.

Benefits of Java Memory Management:

- Simplifies Development: Developers don't need to manually allocate and deallocate memory for objects, reducing the risk of memory-related bugs like dangling pointers and memory leaks.
- Improves Security: Java's memory management prevents common vulnerabilities like buffer overflow and dangling pointer dereferencing.
- Enhances Portability: Java's platform-independent memory management ensures consistent behavior across different operating systems and hardware platforms.

JVM Memory structure:

JVM memory structure is divided into five parts, as shown in below diagram:

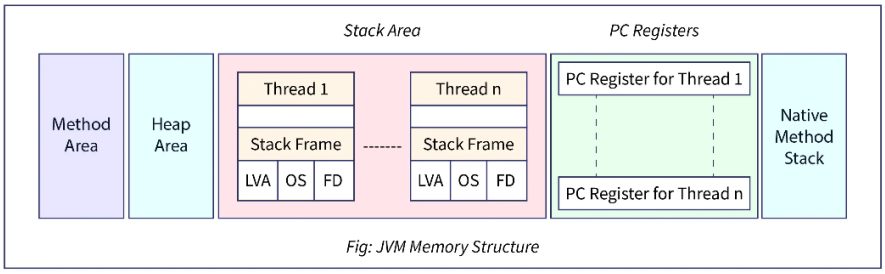


Fig: JVM Memory Structure

* Method Area: It is a logical part of the heap area and is created on virtual machine startup. This memory is allocated for class structures, method data and constructor field data, and also for interfaces or special method used in class. Heap can be of fixed or dynamic size depending upon the system's configuration. Can be of a fixed size or expanded as required by the computation. Needs not to be contiguous.

** Note: Though method area is logically a part of heap, it may or may not be garbage collected even if garbage collection is compulsory in heap area.

* Native method Stacks: Also called as C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when its created. And it can be of fixed or dynamic nature. It usually stores the methods from native languages such as c and c++. Also the methods compiled by JIT compiler are stored in this stack area.

* Program counter (PC) registers: Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non-native method has a PC register which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

* Stack Memory in Java:

Stack Memory in Java is used for static memory allocation and the execution of a thread. It contains primitive values that are specific to a method and references to objects referred from the method that are in a heap. Access to this memory is in Last-In-First-Out (LIFO) order. Whenever we call a new method, a new block is created on top of the stack which contains values specific to that method, like primitive variables and references to objects. When the method finishes execution, its corresponding stack frame is flushed, the flow goes back to the calling method, and space becomes available for the next method.

A stack is created at the same time when a thread is created and is used to store data and partial results which will be needed while returning value for method and performing dynamic linking. Stacks can either be of fixed or dynamic size. The size of a stack can be chosen independently when it is created. The memory for stack needs not to be contiguous.

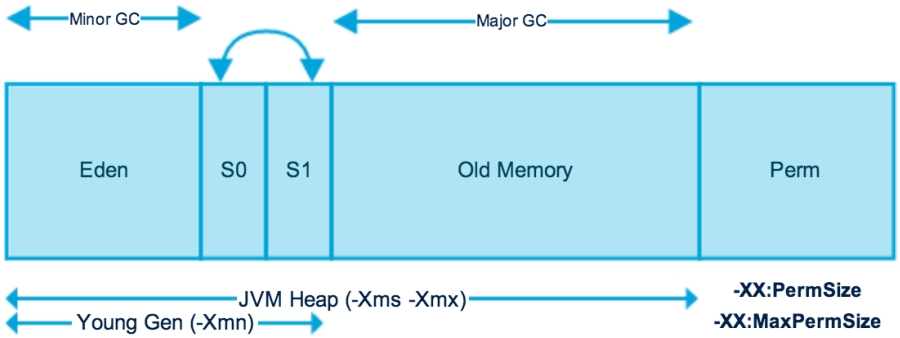
* Key Features of Stack Memory:

Some other features of stack memory include:

1. It grows and shrinks as new methods are called and returned, respectively.
2. Variables inside the stack exist only as long as the method that created them is running.
3. It's automatically allocated and deallocated when the method finishes execution.
4. If this memory is full, Java throws java.lang.StackOverFlowError.
5. Access to this memory is fast when compared to heap memory.
6. This memory is thread-safe, as each thread operates in its own stack.

* Heap Space in Java:

Heap space is used for the dynamic memory allocation of Java objects and JRE classes at runtime. New objects are always created in heap space, and the references to these objects are stored in stack memory. These objects have global access and we can access them from anywhere in the application. We can break this memory model down into smaller parts, called generations, which are:



1. Young Generation – This is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up. This garbage collection is called as "Minor Garbage Collection". Young generation is again divided into 3 part – Eden memory and 2 survivor spaces.
   - Most of the newly created objects are located in the eden memory space, when eden space is filled with objects minor gc is performed and all the survivor objects are moved to one of the survivor space.
   - Minor GC also checks the survivor space and then move them to the other survivor space, so at a time one of the survivor space is always empty.
   - Objects that are survived after many cycles of Minor GC are moved to the old generation memory space, usually it is done by setting a threshold for the age of young generation objects before promoting to old generation.

2. Old or Tenured Generation – This is where long surviving objects are stored. When objects are stored in the Young Generation, a threshold for the object's age is set, and when that threshold is reached, the object is moved to the old generation. The garbage collection is usually performed when this memory gets full. This GC is called as Major GC and usually takes longer time.

3. Permanent Generation – This consists of JVM metadata for the runtime classes and application methods. PermGen is populated by JVM at runtime based on the classes used by the application. PermGen also contains Java SE library classes and methods.

In older versions of Java (up to Java 8), this region was used to store class metadata, constants, and interned strings. In Java 8 and later, class metadata is stored in the native memory area known as Metaspace.

4. Heap Size:

- The heap size can be configured using JVM command-line options (-Xms and -Xmx) to specify the initial and maximum heap size.

- The JVM manages the heap size dynamically based on the available system resources and the application's memory requirements.

We can always manipulate the size of heap memory as per our requirement.

* Key Features of Java Heap Memory:

Some other features of heap space include:

1.It's accessed via complex memory management techniques that include the Young Generation, Old or Tenured Generation, and Permanent Generation.

2.If heap space is full, Java throws java.lang.OutOfMemoryError.

3.Access to this memory is comparatively slower than stack memory

4.This memory, in contrast to stack, isn't automatically deallocated. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage.

5.Unlike stack, a heap isn't thread-safe and needs to be guarded by properly synchronizing the code.

* Working of a Garbage Collector:

1. JVM triggers this process and as per the JVM garbage collection process is done or else withheld. It reduces the burden of programmer by automatically performing the allocation or de-allocation of memory.

2. Garbage collection process causes the rest of the processes or threads to be paused and thus is costly in nature. This problem is unacceptable for the client but can be eliminated by applying several garbage collector-based algorithms. This process of applying algorithm is often termed as Garbage Collector tuning and is important for improving the performance of a program.

3. Another solution is the generational garbage collectors that adds an age field to the objects that are assigned a memory. As more and more objects are created, the list of garbage grows thereby increasing the garbage collection time. On the basis of how many clocks cycles the objects have survived, objects are grouped and are allocated an 'age' accordingly. This way the garbage collection work gets distributed.

4. In the current scenario, all garbage collectors are generational, and hence, optimal.

** Note: System.gc() and Runtime.gc() are the methods which requests for Garbage collection to JVM explicitly but it does't ensures garbage collection as the final decision of garbage collection is of JVM only. Knowing how the program and it's data is stored or organized is essential as it helps when the programmer intends to write an optimized code in terms of resources and it's consumption. Also it helps in finding the memory leaks or inconsistency, and helps in debugging memory related errors. However, the memory management concept is extremely vast and therefore one must put his best to study it as much as possible to improve the knowledge of the same.

* Advantages of Heap Memory:

- Flexibility: Heap memory allows for dynamic allocation and de-allocation of memory, enabling the creation and destruction of objects as needed.

- Automatic Management: The JVM's garbage collector automates memory management by reclaiming unused memory and preventing memory leaks.

- Supports Dynamic Data Structures: Heap memory supports dynamic data structures like linked lists, trees, and dynamic arrays where memory allocation is not known at compile time.

* Considerations and Best Practices:

- Optimize Memory Usage: Avoid unnecessary object creation and ensure proper object lifecycle management to optimize heap memory usage.

- Tune Garbage Collection: Understand garbage collection algorithms and tune JVM parameters to optimize garbage collection performance based on the application's memory requirements.

- Avoid Memory Leaks: Be mindful of retaining references to objects longer than necessary to prevent memory leaks and excessive heap memory consumption.

In summary, heap memory plays a crucial role in Java's memory management model, providing a flexible and efficient mechanism for allocating and managing memory dynamically during program execution. Understanding heap memory and its management is essential for writing efficient and scalable Java applications.

Why is memory management important in Java?

Memory management is important in Java for the following key reasons:

1. Automatic Memory Management: Java provides automatic memory management through its garbage collector, which frees developers from the burden of manual memory allocation and deallocation. However, understanding how the memory management works is still important to write efficient and high-performing Java applications.
2. Preventing Memory Leaks: Even though Java has automatic memory management, it is still possible to create memory leaks if objects are not properly referenced. Knowing how memory management works helps developers identify and prevent such memory leaks.
3. Optimizing Performance: Understanding Java's memory structure (heap, stack, method area, etc.) and the garbage collection process allows developers to optimize their code and application design to reduce the load on the garbage collector, leading to better performance.
4. Debugging Memory-Related Issues: When Java applications encounter issues related to memory, such as OutOfMemoryError, understanding memory management is crucial for effectively debugging and resolving these problems.
5. Tuning the JVM: Advanced Java developers may need to tune the JVM's memory management parameters, such as heap size, garbage collection algorithm, and more, to achieve the desired performance characteristics of their applications. This requires a deep understanding of Java's memory management.

6. Writing Low-Latency Applications: In the context of low-latency systems, where performance is critical, a deep understanding of Java's memory management is essential to minimize the impact of garbage collection and other memory-related operations.
7. Efficient Resource Utilization: Proper memory management ensures that the available memory resources are utilized efficiently, preventing unnecessary memory consumption and wastage.
8. Scalability and Concurrency: Understanding memory management is crucial when building scalable and concurrent Java applications, as it helps developers manage shared memory access and avoid race conditions or other concurrency-related issues.
9. Compliance with Memory Constraints: In certain environments, such as embedded systems or mobile devices, Java applications may need to operate within strict memory constraints. Knowing how to manage memory effectively is crucial in such scenarios.
10. Interoperability with Native Code: When Java applications need to interact with native code (e.g., through the Java Native Interface, or JNI), understanding memory management becomes essential to ensure proper data exchange and avoid memory-related bugs.
11. Predictable Behavior: Comprehending Java's memory management model helps developers write more predictable and deterministic code, as they can anticipate and control the behavior of their applications concerning memory usage and garbage collection.
12. Reduced Cognitive Overhead: By understanding memory management, Java developers can focus more on the core functionality of their applications, rather than spending time debugging and troubleshooting memory-related issues.
13. Compliance with Best Practices: Adhering to memory management best practices, such as proper object lifecycle management and efficient memory allocation, is essential for writing high-quality, maintainable, and secure Java code.
14. Improved Testability and Observability: Knowledge of memory management aids in designing testable and observable Java applications, as developers can better understand and control the memory-related aspects of their systems.

Overall, a deep understanding of Java's memory management is a crucial skill for Java developers, as it enables them to write efficient, scalable, and robust applications that make the most of the language's automatic memory management capabilities.

* String class in Java:

String is the fundamental and widely used class in java that represents a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created.

* Creating a String: There are two ways to create string in Java:

1. String literal: String s = “GeeksforGeeks”;

2. Using new keyword: String s = new String (“GeeksforGeeks”);

Key Features of the String Class:

1. Immutable Nature: Strings in Java are immutable, which means their values cannot be changed after they are created. Operations that modify strings (such as concatenation) actually create new String objects rather than modifying the existing ones.

2. **String Literal vs. String Object**: Strings can be created using string literals or by explicitly creating String objects using the new keyword. String literals (e.g., "Hello") are stored in a special memory area called the "String constant pool" to optimize memory usage and facilitate string interning. Whereas the objects created using new keyword are stored in heap memory space.
3. **String Pool**: The string constant pool is a pool of unique string literals stored in memory. When a string literal is encountered, Java checks if it already exists in the string pool. If the string literal already exists, Java returns a reference to the existing string from the pool rather than creating a new object.
4. **Commonly Used Methods**: The String class provides a wide range of methods for string manipulation and comparison, such as:

    length(): Returns the length of the string.
    charAt(int index): Returns the character at the specified index.
    substring(int beginIndex): Returns a substring starting from the specified index.
    equals(Object obj): Compares the string with the specified object for equality.
    toUpperCase(), toLowerCase(): Converts the string to uppercase or lowercase.

5. **String Concatenation**: String concatenation in Java can be performed using the + operator or the concat() method. String concatenation involving string literals is optimized by the compiler to use StringBuilder under the hood for efficiency.
6. **Unicode Support**: Java String class supports Unicode characters, allowing representation of text in different languages and scripts.

* String Constructors in Java:

1. String(byte[] byte_arr): Construct a new String by decoding the byte array. It uses the platform's default character set for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

String s_byte =new String(b_arr); //Geeks
```

2. String(byte[] byte_arr, Charset char_set): Construct a new String by decoding the byte array. It uses the char_set for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

Charset cs = Charset.defaultCharset();

String s_byte_char = new String(b_arr, cs); //Geeks
```

3. String(byte[] byte_arr, String char_set_name): Construct a new String by decoding the byte array. It uses the char_set_name for decoding. It looks similar to the above constructs and they appear before similar functions but it takes the TIllMay2024.String(which contains char_set_name) as parameter while the above constructor takes CharSet.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

String s = new String(b_arr, "US-ASCII"); //Geeks
```

4. String(byte[] byte_arr, int start_index, int length): Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

String s = new String(b_arr, 1, 3); // eek
```

5. String(byte[] byte_arr, int start_index, int length, Charset char_set):

Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).Uses char_set for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

Charset cs = Charset.defaultCharset();

String s = new String(b_arr, 1, 3, cs); // eek
```

6. String(byte[] byte_arr, int start_index, int length, String char_set_name): Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).Uses char_set_name for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};

String s = new String(b_arr, 1, 4, "US-ASCII"); // eeks
```

7. String(char[] char_arr): Allocates a new String from the given Character array

Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};

String s = new String(char_arr); //Geeks
```

8. String(char[] char_array, int start_index, int count): Allocates a String from a given character array but choose count characters from the start_index.

Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};
```

String s = new String(char_arr , 1, 3); //eek

9. String(int[] uni_code_points, int offset, int count): Allocates a String from a uni_code_array but choose count characters from the start_index.

   Example:

   int[] uni_code = {71, 101, 101, 107, 115};

   String s = new String(uni_code, 1, 3); //eek

10. String(StringBuffer s_buffer): Allocates a new string from the string in s_buffer

    Example:

    StringBuffer s_buffer = new StringBuffer("Geeks");

    String s = new String(s_buffer); //Geeks

11. String(StringBuilder s_builder): Allocates a new string from the string in s_builder

    Example:

    StringBuilder s_builder = new StringBuilder("Geeks");

    String s = new String(s_builder); //Geeks

*String Methods in Java:

1. int length(): Returns the number of characters in the String.

   "GeeksforGeeks".length(); // returns 13

2. Char charAt(int i): Returns the character at ith index.

   "GeeksforGeeks".charAt(3); // returns ‘k’

3. String substring (int i): Return the substring from the ith index character to end.

   "GeeksforGeeks".substring(3); // returns “ksforGeeks”

4. String substring (int i, int j): Returns the substring from i to j-1 index.

   "GeeksforGeeks".substring(2, 5); // returns “eks”

5. String concat(String str): Concatenates specified string to the end of this string.

   String s1 = ”Geeks”;

   String s2 = ”forGeeks”;

   String output = s1.concat(s2); // returns “GeeksforGeeks”

6. int indexOf (String s): Returns the index within the string of the first occurrence of the specified string.

   If String s is not present in input string then -1 is returned as the default value.

1. String s = ”Learn Share Learn”;

   int output = s.indexOf(“Share”); // returns 6

2. String s = "Learn Share Learn"

   int output = s.indexOf(“Play”); // return -1

7. int indexOf (String s, int i): Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

   String s = ”Learn Share Learn”;

   int output = s.indexOf("ea",3);// returns 13

8. Int lastIndexOf(String s): Returns the index within the string of the last occurrence of the specified string. If String s is not present in input string then -1 is returned as the default value.

   1. String s = ”Learn Share Learn”;

      int output = s.lastIndexOf("a"); // returns 14

   2. String s = "Learn Share Learn"

      int output = s.indexOf(“Play”); // return -1

9. boolean equals(Object otherObj): Compares this string to the specified object.

   Boolean out = “Geeks”.equals(“Geeks”); // returns true

   Boolean out = “Geeks”.equals(“geeks”); // returns false

10. boolean equalsIgnoreCase (String anotherString): Compares string to another string, ignoring case considerations.

    Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true

    Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true

11. int compareTo(String anotherString): Compares two string lexicographically.

    int out = s1.compareTo(s2);

    // where s1 and s2 are

    // strings to be compared

    This returns difference s1-s2. If :

    out < 0 // s1 comes before s2

    out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2.

12. int compareToIgnoreCase(String anotherString): Compares two string lexicographically, ignoring case considerations.

int out = s1.compareToIgnoreCase(s2);

// where s1 and s2 are

// strings to be compared

This returns difference s1-s2. If :

out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2.

Note: In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).

13. String toLowerCase(): Converts all the characters in the String to lower case.

String word1 = “HeLLo”;

String word3 = word1.toLowerCase(); // returns “hello"

14. String toUpperCase(): Converts all the characters in the String to upper case.

String word1 = “HeLLo”;

String word2 = word1.toUpperCase(); // returns “HELLO”

15. String trim(): Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

String word1 = “ Learn Share Learn “;

String word2 = word1.trim(); // returns “Learn Share Learn”

16. String replace (char oldChar, char newChar)

Returns new string by replacing all occurrences of oldChar with newChar.

String s1 = “feeksforfeeks“;

String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // return “geeksforgeeks”

Note: s1 is still feeksforfeeks and s2 is geeksgorgeeks

17. boolean contains(string): Returns true if string contains contains the given string

String s1="geeksforgeeks";

String s2="geeks";

s1.contains(s2) // return true

18. Char[] toCharArray(): Converts this String to a new character array.

String s1="geeksforgeeks";

char []ch=s1.toCharArray(); // returns ['g', 'e' , 'e' , 'k' , 's' , 'f', 'o', 'r' , 'g' , 'e' , 'e' , 'k' ,'s']

19. boolean starsWith(string): Return true if string starts with this prefix.

String s1="geeksforgeeks";

String s2="geeks";

s1.startsWith(s2) // return true

What are Strings in Java?

=> Strings are the type of objects that can store the character of values and in Java, every character is stored in 16 bits i.e using UTF 16-bit encoding. A string acts the same as an array of characters in Java.

There are two ways to create a string in Java:

1. String literal: To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).
2. Using new keyword: String s = new String("Welcome");

In such a case, JVM will create a new string object in normal (non-pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variables will refer to the object in the heap (non-pool)

3. String comparison: Objects.equals(str, "Text") this syntax can be used to check whether two strings are equal or not.

Why make string as immutable?

Ans: As most of the string objects are stored in scp, there a single string object is referenced by many references. If any one of the reference is trying to change the object that will affect all other reference objects, so to prevent this java has made string class as immutable.

The key differences between final and immutable are:

- Final: Relates to variables, methods, or classes that cannot be changed or overridden after initialization or declaration.
- Immutable: Relates to objects whose state cannot be modified after construction, typically achieved by making all fields final and ensuring no mutation methods are provided.

While final emphasizes immutability at the variable or method level, "immutable" describes the characteristic of an entire object's state being unchangeable once created. Immutable objects provide benefits such as thread safety, simplification of code, and improved reliability in concurrent environments.

StringBuffer and StringBuilder in Java:

StringBuffer Class: StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth. In order to create a string buffer, an object needs to be created, (i.e.), if we wish to create a new string buffer with name str, then:

```
StringBuffer str = new StringBuffer();
```

StringBuilder Class: Similar to StringBuffer, the StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters. The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters. Similar to StringBuffer, in order to create a new string with the name str, we need to create an object of StringBuilder, (i.e.):

```
StringBuilder str = new StringBuilder();
```

Conversion from StringBuffer to StringBuilder:

The StringBuffer cannot be directly converted to StringBuilder. We first need to convert the StringBuffer to a String object by using the inbuilt method toString(). After converting it to a string object, we can simply create a StringBuilder using the constructor of the class.

Conversion from StringBuilder to StringBuffer:

Similar to the above conversion, the StringBuilder cannot be converted to the StringBuffer directly. We first need to convert the StringBuilder to the String object by using the inbuilt method toString(). Now, we can create a StringBuilder using the constructor.

StringBuilder vs StringBuffer in Java:

StringBuffer Class	StringBuilder Class
StringBuffer is present in Java.	StringBuilder was introduced in Java 5.
StringBuffer is synchronized. This means that multiple threads cannot call the methods of StringBuffer simultaneously.	StringBuilder is asynchronized. This means that multiple threads can call the methods of StringBuilder simultaneously.
Due to synchronization, StringBuffer is called a thread safe class.	Due to its asynchronous nature, StringBuilder is not a thread safe class.
Due to synchronization, StringBuffer is lot slower than StringBuilder.	Since there is no preliminary check for multiple threads, StringBuilder is a lot faster than StringBuffer.

String vs StringBuilder vs StringBuffer:

Feature	String	StringBuilder	StringBuffer
Introduction	Introduced in JDK 1.0	Introduced in JDK 1.5	Introduced in JDK 1.0
Mutability	Immutable	Mutable	Mutable
Thread Safety	Thread Safe	Not Thread Safe	Thread Safe
Memory Efficiency	High	Efficient	Less Efficient
Performance	High(No-Synchronization)	High(No-Synchronization)	Low(Due to Synchronization)
Usage	This is used when we want immutability.	This is used when Thread safety is not required.	This is used when Thread safety is required.

Methods in StringBuffer class:

Both the methods in stringbuffer and StringBuilder class are almost similar to each other, they are as follows

Methods	Action Performed
append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method.
capacity()	the total allocated capacity can be found by the capacity( ) method.
charAt()	This method returns the char value in this sequence at the specified index.
delete()	Deletes a sequence of characters from the invoking object.
deleteCharAt()	Deletes the character at the index specified by the *loc.*
ensureCapacity()	Ensures capacity is at least equal to the given minimum.
insert()	Inserts text at the specified index position.
length()	Returns the length of the string.
reverse()	Reverse the characters within a StringBuffer object.
replace()	Replace one set of characters with another set inside a StringBuffer object.

Similarly, there are many methods in these (can refer to: https://www.geeksforgeeks.org/stringbuffer-class-in-java/)

HashCode:

In Java, the hashCode() method is used to generate a hash code for an object. The hash code is an integer value that is used to support hash tables such as those provided by HashMap, HashSet, and Hashtable. Understanding how hashCode() works and how to override it properly is crucial for ensuring the correct behavior of hash-based collections.

Contract with equals():

- The hashCode() method must be consistent with the equals() method. This means that if two objects are equal according to the equals() method, they must have the same hash code.
- The converse is not necessarily true: two objects having the same hash code do not have to be equal according to the equals() method (though this will lead to hash collisions).

* 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.

1. What 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 Boolean, Byte, Short, Character, Integer, Long, Float, Double.

Need of Wrapper Class in Java:

Wrapper classes in Java are used to wrap primitive data types (such as int, char, and boolean) in an object. This is necessary because in Java, everything is an object, and objects are required in certain situations, such as when a method requires an object as a parameter or when you need to store multiple values in a single structure like an ArrayList or a HashMap.

Custom Wrapper Class in Java:

In Java we use a wrapper class in java to wrap the primitive data type but instead of using the wrapper class in java, we can create our own custom class which will wrap the primitive data type just like the wrapper class in java.

Example of Custom Wrapper class in Java
Now, we will look at the example of creating a custom wrapper class.

```
class Javatpoint{

private int i;
Javatpoint(){}
Javatpoint(int i){
this.i=i;
}
public int getValue(){
return i;
}
public void setValue(int i){
this.i=i;
}
@Override
public String toString() {
return Integer.toString(i);
}
}
class TestJavatpoint{
public static void main(String[] args){
Javatpoint j=new Javatpoint(40);
System.out.println(j);
}}
```

Uses of Wrapper Class in Java:

The main uses of the wrapper class in Java are

- **Representing primitive data types as objects:** Wrapper classes provide a way to use primitive data types as objects, which is necessary in certain situations where only objects are accepted, such as in collections like ArrayList and HashMap.
- **Storing null values:** Primitive data types in Java cannot store null values, but wrapper classes can be set to null, which can be useful in representing missing or unknown values.
- **Method parameters:** Wrapper classes can be used as method parameters, allowing methods to accept objects instead of primitive data types.
- **Autoboxing and unboxing:** Wrapper classes provide automatic conversions between primitive data types and their corresponding wrapper objects through a feature called autoboxing and unboxing.
- **Converting between primitive data types and strings:** Wrapper classes provide methods for converting primitive values to and from strings, which can be useful for reading data from a file or user input.
- **Comparing values:** Wrapper classes provide methods for comparing values, such as equals() and compareTo(), which can be useful for sorting and searching.

- **Performing mathematical operations:** Wrapper classes provide methods for performing mathematical operations, such as addition, subtraction, and multiplication, on primitive values.

Advantages of Wrapper Class in Java

The following are the advantages of the wrapper class in java

- **Provide an Object representation of primitive data types:** Wrapper classes provide an object representation of primitive data types, which allows developers to use primitives as objects in their code.
- **Facilitate type conversion:** Wrapper classes can be used to convert primitive data types to and from String representation, making it easy to store primitives in collections or pass them as method arguments.
- **Help in implementing Autoboxing and Unboxing:** Java provides a feature called Autoboxing and Unboxing, which automatically converts between primitive and wrapper class objects, simplifying the code and reducing the number of explicit type conversions required.

Disadvantages of Wrapper Class in Java

Below are the disadvantages of the wrapper class in java.

- **Performance Overhead:** The process of converting primitive data types to and from Wrapper classes can result in performance overhead, especially in large-scale applications where this conversion takes place frequently.
- **Increased Memory Usage:** Wrapper classes consume more memory than primitive data types, as they are objects and contain additional information like type information, methods, etc.
- **Immutable:** Wrapper classes are immutable, meaning their values cannot be changed once they are created. This can be limiting in certain situations where it is necessary to modify the value of a primitive data type.

* Arrays in Java => In Java, all arrays are dynamically allocated. Arrays may be stored in contiguous memory [consecutive memory locations]. Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using sizeof. A Java array variable can also be declared like other variables with [] after the data type. The variables in the array are ordered, and each has an index beginning with 0. Java array can also be used as a static field, a local variable, or a method parameter. An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class depending on the definition of the array. In the case of primitive data types, the actual values might be stored in contiguous memory locations (JVM does not guarantee this behavior). In the case of class objects, the actual objects are stored in a heap segment.

Array Literal in Java => In a situation where the size of the array and variables of the array are already known, array literals can be used.

```
// Declaring array literal
int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
```

The length of this array determines the length of the created array. There is no need to write the new int[] part in the latest versions of Java.

Accessing Java Array Elements using for Loop => Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

```
 // accessing the elements of the specified array
for (int i = 0; i < arr.length; i++){
        System.out.println("Element at index " + i + " : "+ arr[i]);
}
```

*Exceptions in Java:

Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

What are Java Exceptions?

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Difference between Error and Exception:

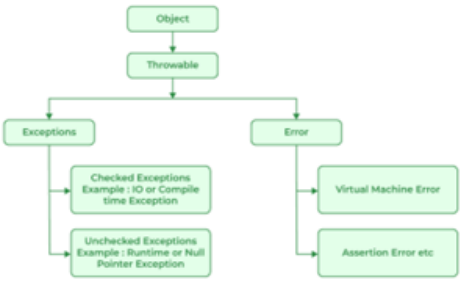
Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

Error: An Error indicates a serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

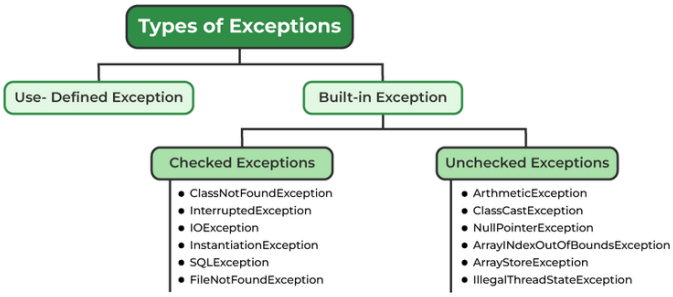
Exception Hierarchy:

All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Types of Exceptions:

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. Built-in Exceptions:

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

Checked Exceptions: Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

Unchecked Exceptions: The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Note: *For checked vs unchecked exception, see Checked vs Unchecked Exceptions*

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of Errors
- Meaningful Error Reporting
- Identifying Error Types

Methods to print the Exception information:

1. **printStackTrace():** This method prints exception information in the format of the Name of the exception: description of the exception, stack trace.
2. **toString()**: The toString() method prints exception information in the format of the Name of the exception: description of the exception.
3. **getMessage()**: The getMessage() method prints only the description of the exception.

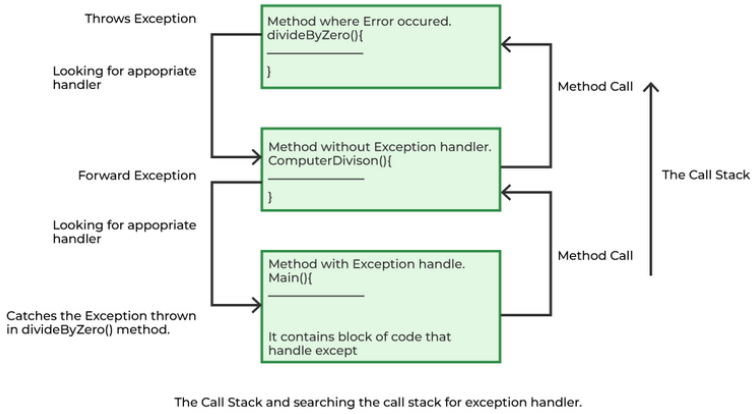
How Does JVM Handle an Exception?

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
- The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.
- If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.
- If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.

```
Exception in thread "xxx" Name of Exception: Description
... ..... .. // Call Stack
```

Look at the below diagram to understand the flow of the call stack.



The Call Stack and searching the call stack for exception handler.

Java Exception Keywords:

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java try block:

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method. If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception. Java try block must be followed by either catch or finally block.

Syntax of Java try-catch:

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block:

try{

//code that may throw an exception

}**finally**{}

Java catch block:

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception. The catch block must be used after the try block only. You can use multiple catch block with a single try block.

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Certain key points need to be remembered that are as follows:

- In a method, there can be more than one statement that might throw an exception, So put all these statements within their own **try** block and provide a separate exception handler within their own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate the exception handler, we must put a **catch** block after it. There can be more than one exception handler. Each **catch** block is an exception handler that handles the exception to the type indicated by its argument. The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the **Throwable** class.
- For each try block, there can be zero or more catch blocks, but **only one** final block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after **try and catch blocks.** And if an exception does not occur, then it will be executed after the **try** block. The finally block in Java is used to put important codes such as clean-up code e.g., closing the file or closing the connection.
- If we write System.exit in the try block, then finally block will not be executed.

What is an Unchecked Exception in Java ?

It occurs at the time of execution and is known as a run time exception. It includes **bugs**, **improper usage of API**, and **syntax** or **logical error** in programming. In Java, exceptions that are under **Error** and**, Runtime exception** classes are unchecked exceptions.

Types of Unchecked Exceptions:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- NumberFormatException
- InputMismatchException
- IllegalStateException

Arithmetic exception:

It is a type of unchecked error in code that is thrown whenever there is a **wrong mathematical** or **arithmetic calculation** in the code, especially during run time. For instance, when the denominator in a fraction is zero, the arithmetic exception is thrown.

NullPointerException:

It is a **run time exception**. It is thrown when a null value is assigned to a reference object and the program tries to use that null object.

ArrayIndexOutOfBoundsException:

It occurs when we access an array with an invalid index. This means that either the index value is less than zero or greater than that of the array's length.

NumberFormatException:

It is a type of unchecked exception that occurs when we are trying to convert a string to an int or other numeric value. This exception is thrown in cases when it is not possible to convert a string to other numeric types.

InputMismatchException:

It occurs when an input provided by the user is incorrect. The type of incorrect input can be out of range or incorrect data type.

IllegalStateException:

It is a run time exception that occurs when a method of a code is triggered or invoked at the wrong time. This exception is used to give a signal that the method is invoked at the wrong time.

What are Checked Exceptions in Java?

In Java, Checked Exceptions are exceptions that are checked by the compiler during the compilation of the program. An example of a Checked Exception is the FileNotFoundException, which occurs during file handling in a Java program when the path of the file is invalid. This is a common cause of FileNotFoundException during compilation of a Java program.

Types of checked Exception

1. IO Exception: IO Exception generally occurs during the read and write operations on a particular file. We have already seen an IO Exception at the starting of an article i.e FileNotFoundException is an IO Exception that occurs due to the invalid path of the file.
2. SQL Exception: SQL Exception is also a type of checked exception which occurs both in the driver and the database. It provides information of database access errors and other database connection errors.
3. ParseException: The ParseException is also a checked exception. This exception occurs when we fail to parse a string for a specific format of type. This type of exception is generally seen during the formatting of the specific data.

Basis of Comparison	Checked Exceptions	Unchecked Exceptions
Compilation	These Exceptions occur during the compilation time of the java programs.	These Exceptions occur at runtime of the java programs.
Compiler Checking	Checked Exceptions are checked by the java compiler.	Unchecked Exceptions are not checked by the java compiler.
Exceptions Handling	These Exceptions can be handled during the compilation time.	These Exceptions cannot be handled at compilation time.
Examples:	IOException FileNotFoundException InterruptedException	ArithmeticException InputMismatchException NullPointerException

Finally Block in Java:

A “finally” is a keyword used to create a block of code that follows a try or catch block. A finally block contains all the crucial codes such as closing connections, stream, etc that is always executed whether an exception occurs within a try block or not. When finally block is attached with a *try-catch block*, it is always executed whether the catch block has handled the exception thrown by try block or not.

Some important rules of using finally block or clause are:

1. A finally block is optional but at least one of the catch or finally block must exist with a try.
2. It must be defined at the end of last catch block. If finally block is defined before a catch block, the program will not compile successfully.
3. Unlike catch, multiple finally blocks cannot be declared with a single try block. That is there can be only one finally clause with a single try block.

Use of finally block in Java:

1. Generally, finally block or clause is used for freeing up resources, cleaning up code, db closing connection, io stream, etc.

2. A java finally block is used to prevent resource leak. While closing a file or recovering resources, the code is put inside the finally block to ensure that the resource is always recovered.
3. Finally clause is used for terminating threads.

Conditions where finally block does not execute:

There are the following conditions where finally block does not execute in Java. They are as follows:

1. When System.exit() method is invoked before executing finally block.
2. When an exception happens in the finally block.
3. When the return statement is declared in the finally block, the control is transferred to the calling routine, and statements after return statement inside finally block will not be executed.

Return Statement in Try Catch Finally Block in Java:

Case 1: Return statement in try block but do not have a return statement at the end of method:

In the preceding program, we did not return a value at the end of the method. Therefore, we will get compile-time error: “This method must return a result of type int”. So, this is an invalid case.

Case 2: Return statement in try block and end of method:

This is a valid case because we have returned a value 20 at the end of method.

Case 3: Return statement in try block and end of method but statement after return:

When you will try to execute the preceding program, you will get an unreachable code error. This is because any statement after return statement will result in compile-time error stating “Unreachable code”.

Case 4: Return statement in try block and at end of method but exception occurred in try block:

In the preceding code, an exception occurred in a try block, and the control of execution is transferred to catch block to handle exception thrown by the try block. Due to an exception occurred in try block, return statement in try block did not execute. Return statement defined at the end of method has returned a value 20 to the calling method.

Case 5: Return statement in try-catch block:

Here we'll get output along with the returned value

Case 6: Return statement in try-catch block and a statement at end of method:

We'll get compile time error: Un-reachable code

Case 7: Return statement in catch block but no exception in try block:

Here we'll get output with the value

Case 8: Return statement in catch block but exception occurred in try block.

Case 9: Return statement in try block and finally block:

In the preceding code, finally block overrides the value returned by try block. Therefore, this would return value 50 because the value returned by try has been overridden by finally block.

Case 10: Return statement in catch and finally blocks:

In the above example program, finally block overrides the value returned by catch block. Therefore, the returned value is 50.

Case 11: Return statement in catch and finally blocks but a statement after finally block:

Here we'll get compile time error: Un-reachable code

Throw Keyword in Java | Java Throw Exception:

You will have seen earlier in all the programs of *exception handling* that Java runtime system (JVM) was responsible for identifying exception class, creating its object, and throwing that object.

JVM automatically throws system-generated exceptions. All those exceptions are called implicit exceptions. If we want to throw an exception manually or explicitly, for this, Java provides a keyword throw.

Throw in Java is a keyword that is used to throw a built-in exception or a custom exception explicitly or manually. Using throw keyword, we can throw either checked or unchecked exceptions in Java programming.

When an exception occurs in the try block, throw keyword transfers the control of execution to the caller by throwing an object of exception.

Only one object of exception type can be thrown by using throw keyword at a time. Throw keyword can be used inside a method or static block provided that exception handling is present.

Key points of Throw keyword:

1. In Java exception handling, we use throw keyword to throw a single exception explicitly. It is followed by an instance variable.
2. Using throw keyword, we can throw either checked or unchecked exception in Java.
3. The keyword throw raises an exception by creating a subclass object of Exception explicitly.
4. We mainly use throw keyword to throw custom exception on the basis of some specified condition.
5. We use keyword throw inside the body of method or constructor to invoke an exception.
6. With the help of throw keyword, we cannot throw more than one exception at a time.

Syntax of Throw Keyword:

```
throw exception_name;
```

Control flow of try-catch block with Java throw Statement:

When a throw statement is encountered in a program, the flow of execution of subsequent statements stops immediately in the try block and the corresponding catch block is searched. The nearest try block is inspected to see if it has a catch block that matches the type of exception. If corresponding catch block is found, it is executed otherwise the control is transferred to the next statement. In case, no matching catch block is found, JVM transfers the control of execution to the default exception handler that stops the normal flow of program and displays error message on the output screen.

Rethrowing an Exception in Java:

Java version 7 introduces a mechanism **rethrowing an exception**. When an exception occurs in a try block, it is caught by a catch block inside the same method. The same exception object out from the catch block can be rethrown explicitly using throw keyword. This is called rethrowing of exception in Java. When the same exception object is rethrown, it preserves details of original exception.

Throws Keyword in Java | Throws Clause:

In Java, sometimes a method may throw an exception in a program but cannot handle it due to not have an appropriate exception handling mechanism. In such a case, the programmer has to throw that exception to the caller of the method using **throws clause**. Throws clause consists of throws keyword followed by a comma-separated by the list of all exceptions thrown by that method.

Throws Keyword in Java:

Throws keyword in Java is used in the method declaration. It provides information to the caller method about exceptions being thrown and the caller method has to take the responsibility of handling the exception. Throws keyword is used in case of checked exception only because if we are not handling runtime exceptions (unchecked exceptions), Java compiler does not give any error related to runtime exceptions. If an error occurs, we are unable to do anything. When the code generates a checked exception inside a method but the method does not handle it, Java compiler detects it and informs us about it to handle that exception. In this case, compulsorily, we must handle that checked exception otherwise we will get an error flagged by Java compiler. To prevent this error flagged by the compiler, we need to handle exceptions using throw clause.

There are two ways to handle the exception:

1. By using try-catch block
2. By using throws keyword

Key Points of Throws Keyword

1. In Java exception handling, we use throws keyword to define a list of exceptions which may be thrown by that method.
2. We can use throws keyword in method or constructor declaration (signature) to denote exceptions that can be possibly thrown by that method.
3. Throws is followed by the exception class name.
4. With throws clause, we can declare more than one exception.

Syntax:

access_specifier return_type method_name(parameter list) throws exception

Difference between Throw and Throws in Java:

There are some key difference between throw and throws keywords in Java. They are as:

1. The keyword throw is used to throw an exception explicitly, while the throws clause is used to declare an exception.
2. Throw is followed by an instance variable, while throws is followed by the name of exception class.
3. We use throw keyword inside method body to call an exception, while the throws clause is used in method signature.
4. With throw keyword, we cannot throw more than one exception at a time, while we can declare multiple exceptions with throws.

User defined Exception in Java:

Sometimes, predefined exceptions in Java are not suitable for describing a certain situation. In other words, the predefined exception classes of Java do not fit as per our needs in certain scenarios. In such cases, the programmer wants to create his own customized exception as per requirements of the application, which is called user-defined exception or custom exception in Java.

User-defined exceptions in Java are those exceptions that are created by a programmer (or user) to meet the specific requirements of the application. That's why it is also known as user-defined exception. It is useful when we want to properly handle the cases that are highly specific and unique to different applications.

For example:

1. A banking application, a customer whose age is lower than 18 years, the program throws a custom exception indicating "needs to open a joint account".
2. Voting age in India: If a person's age entered is less than 18 years, the program throws "invalid age" as a custom exception.

Actually, there are mainly two drawbacks of predefined exception handling mechanism. They are:

- Predefined exceptions of Java always generate the exception report in a predefined format.
- After generating the exception report, it immediately terminates the execution of the program.

The user-defined exception handling mechanism does not contain the above explained two drawbacks. It can generate the report of error message in any special format that we prefer. It will automatically resume the execution of the program after generating the error report.

How to Create Your Own User-defined Exception in Java?

There are the following steps that are followed in creating a user-defined exception or custom exception in Java. They are as follows:

Step 1: User-defined exceptions can be created simply by extending the Exception class. This is done as:

class OwnException extends Exception

Step 2: If you do not want to store any exception details, define a default constructor in your own exception class. We can do this as follows:

```
OwnException()
{
}
```

Step 3: If you want to store exception details, define a parameterized constructor with string as a parameter, call the superclass (Exception) constructor from this, and store variable “str”. This can be done as follows:

```
OwnException(String str)
{  super(str); // Call superclass exception constructor and store variable "str" in it.
}
```

Step 4: In the last step, we need to create an object of the user-defined exception class and throw it using throw clause.

```
OwnException obj = new OwnException("Exception details");
 throw obj;
```

or,

```
 throw new OwnException("Exception details");
```

Chained Exceptions in Java:

Chained exception in Java is a technique to handle exceptions that occur one after another in a program. This technique helps us to know when one exception causes another in a program.

For example, let us assume that a, b, and c are objects of three different exception types A, B, and C, respectively. The object a of type A causes an exception of type B to occur and an object of B type also causes an exception of C type.

This process is called chaining of exceptions in Java, and the exceptions involved in this process are called chained exceptions. This feature helps the programmer to know when and where is the cause for the exception.

Constructors of Throwable Class to Support Chained Exceptions:

1. **Throwable(Throwable causeExc):**

This form of constructor creates a new Throwable object with the specified cause. It takes only one parameter: Throwable causeExc, which represents an exception that causes the current exception. If the causeExc is null, it will return the null of message. Otherwise, it will return string representation of the message. The message contains the name of class and the detailed information of the cause.

2. **Throwable(String msg, causeExc):**

This form of constructor creates a new Throwable object with the specified detail message and cause. It takes two parameters: String msg and Throwable causeExc. Here, msg is an exception message and causeExc is an exception that causes the current exception.

Methods of Throwable Class to Support Chained Exceptions:

There are the following methods added to the Throwable class that supports chained exceptions in Java.

1. **toString():** This method returns an exception followed by a description of the exception. The general syntax is as follows:

```
public String toString()
```

2. **getMessage():** It returns the description of exception. The general syntax is as:

```
public String getMessage()
```

3. **printStackTrace():** This method displays stack trace. It returns nothing. The general syntax is given below.

```
public void printStackTrace()
```

4. **getCause():** The getCause() method returns the exception that caused the occurrence of current exception. If there is no caused exception then null is returned. The syntax is as follows:

```
public Throwable getCause()
```

5. **initCause():** The initCause() method joins "causeExc" with the invoking exception and returns a reference to the exception.

```
public Throwable initCause(Throwable causeExc)
```

6. **fillInStackTrace():** This method returns a Throwable object that contains a completed stack trace. The object can be rethrown. The basic syntax for fillInStackTrace() method is as below:

```
public Throwable fillInStackTrace()
```

7. **getStackTrace():** The getStackTrace() method returns an array that contains each element on the stack trace. The element at index 0 represents the top of call stack and the last element represents the bottom of call stack. The general syntax for this method is as follows:

```
public StackTraceElement[] getStackTrace()
```

Errors in Java:

Errors in Java occur when a programmer violates the rules of Java programming language. It might be due to programmer's typing mistakes while developing a program. It may produce incorrect output or may terminate the execution of the program abnormally. For example, if you use the right parenthesis in a Java program where a right brace is needed, you have made a syntax error. You have violated the rules of Java language. Therefore, it is important to detect and fix properly all errors occurring in a program so that the program will not terminate during execution.

Types of Errors in Java Programming:

In Java, or other programming languages, when we write a program for the first time, it usually contains errors. We mainly divided these errors into three types:

- Compile-time errors (Syntax errors)
- Runtime errors
- Logical errors

Compile Time Errors in Java:

Compile-time error occurs when syntactical problems occur in a Java program due to incorrect use of Java syntax. These syntactical problems may be missing semicolons, missing brackets, misspelled keywords, use of undeclared variables, class not found, missing double-quote in Strings, and so on. These problems occurring in a program are called **syntax error in Java**. Since all the syntax errors are detected by *Java compiler*, therefore, these errors are also known as compile-time errors in Java. When Java compiler finds any syntax error in a program, it prevents the code from compiling successfully and will not create a .class file until syntax errors are not corrected. An error message will be displayed on the console screen. These errors must be removed by debugging before successfully compile and run the program.

Runtime Errors in Java:

Runtime error occurs when a program is successfully compiled, creating the .class file but does not run properly. It is detected at run time (i.e. during the execution of the program). Java compiler has no technique to detect runtime errors during compilation because a compiler does not have all the runtime information available to it. *JVM* is responsible for detecting runtime errors while the program is running.

Such a program that contains runtime errors may produce wrong results due to wrong logic or terminate the program. These runtime errors are usually known as exceptions. For example, if a user inputs a value of string type in a program, but the computer is expecting an integer value, a runtime error will be generated.

The most common runtime errors in Java programming language are as follows:

1. Dividing an integer by zero.
2. Accessing an element that is out of range of the array.
3. Trying to store a value into an array that is not compatible type.
4. Passing an argument that is not in a valid range or valid value for a method.
5. Striving to use a negative size for an array.
6. Attempting to convert an invalid string into a number.
7. and many more.

When such errors are encountered in a program, Java generates an error message and terminates the program abnormally. To handle these kinds of errors during the runtime, we use *exception handling technique in Java* program.

Logical Errors in Java Program:

Logical errors in Java are the most critical errors in a program, and they are difficult to detect. These errors occur when the programmer uses incorrect logic or wrong formula in the coding. The program will be compiled and executed successfully, but does not return the expected output. Logical errors are not detected either by Java compiler or JVM (Java runtime system). The programmer is entirely responsible for them. Application testers can detect them when they compare the actual result with its expected result.

For example, a programmer wants to print even numbers from an array, but he uses a division (/) operator instead of a modulus (%) operator to get the remainder of each number. Due to which he got the wrong results.

Try-with-resources Feature in Java:

In Java, the **Try-with-resources** statement is a try statement that declares one or more resources in it. It was introduced in java 1.7. A resource is an object that must be closed once your program is done using it. For example, a File resource or a Socket connection resource. The try-with-resources statement ensures that each resource is closed at the end of the statement execution. If we don't close the resources, it may constitute a resource leak and also the program could exhaust the resources available to it.

You can pass any object as a resource that implements *java.lang.AutoCloseable*, which includes all objects which implement java.io.Closeable. By this, now we don't need to add an extra finally block for just passing the closing statements of the resources. The resources will be closed as soon as the try-catch block is executed.

Syntax: Try-with-resources:

```
try(declare resources here) {
   // use resources
}
catch(FileNotFoundException e) {
   // exception handling
}
```

Exceptions:

When it comes to exceptions, there is a difference in try-catch-finally block and try-with-resources block. If an exception is thrown in both try block and finally block, the method returns the exception thrown in finally block. For try-with-resources, if an exception is thrown in a try block and in a try-with-resources statement, then the method returns the exception thrown in the try block. The exceptions thrown by try-with-resources are suppressed, i.e. we can say that try-with-resources block throws suppressed exceptions.

We have to cases for this try-with-resource feature

1. Single resource: Used when we have only one resource in a program
2. Multiple resource: Used when we have multiple resources in a program (We separate them by " ; ")

Multiple Catch Block in Java:

Starting from Java 7.0, it is possible for a single catch block to catch multiple exceptions by separating each with | (pipe symbol) in the catch block. Catching multiple exceptions in a single catch block reduces code duplication and increases efficiency. The bytecode generated while compiling this program will be smaller than the program having multiple catch blocks as there is no code redundancy.

***Note**: If a catch block handles multiple exceptions, the catch parameter is implicitly final. This means we cannot assign any values to catch parameters.*

Syntax:

```
try {
   // code
}
catch (ExceptionType1 | Exceptiontype2 ex) {
   // catch block
}
```

Important Points:

1. If all the exceptions belong to the same class hierarchy, we should be catching the base exception type. However, to catch each exception, it needs to be done separately in their catch blocks.

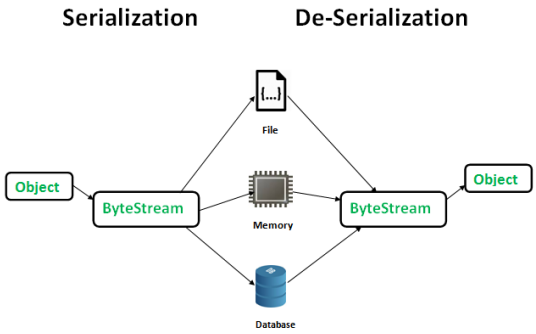
2. Single catch block can handle more than one type of exception. However, the base (or ancestor) class and subclass (or descendant) exceptions cannot be caught in one statement. For Example,

```
// Not Valid as Exception is an ancestor of NumberFormatException
catch(NumberFormatException | Exception ex)
```

3. All the exceptions must be separated by vertical bar pipe **|**.

Serialization and Deserialization in Java:

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

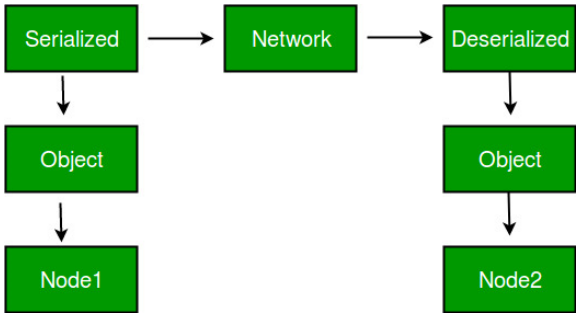
public final void writeObject(Object obj) throws IOException

The ObjectInputStream class contains **readObject()** method for deserializing an object.

```
public final Object readObject()
        throws IOException,
      ClassNotFoundException
```

Advantages of Serialization

- To save/persist state of an object.
- To travel an object across a network.



Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface. Serializable is a **marker interface** (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

Points to remember

1. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true.
2. Only non-static data members are saved via Serialization process.
3. Static data members and transient data members are not saved via Serialization process. So, if you don't want to save value of a non-static data member then make it transient.
4. Constructor of object is never called when an object is deserialized.
5. Associated objects must be implementing Serializable interface.
6. Serializable is a marker interface — it does not consist of any methods or data members. If a java class implements a Serializable interface it gets certain capabilities. It is also important to note that objects of a class can only be serialized if the class implements the Serializable interface.

Example :

```
class A implements Serializable{

// B also implements Serializable
// interface.
B ob=new B();
}
```

Serialization in JAVA — Code

To serialize a Java object, the class (to which the object belongs) must implement the **java.io.Serializable** interface.

The ObjectOutputStream contains the method writeObject() which is used for serializing an object and the ObjectInputStream contains the method readObject() used for deserializing the byte stream.

SerialVersionUID:

The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender's class, the Deserialization will result in an **InvalidClassException**.

A Serializable class can declare its own UID explicitly by declaring a field name. It must be static, final and of type long. i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L; If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data. It is also recommended to use private modifier for UID since it is not useful as inherited member. **serialver** The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.

In case of **transient variables:-** A variable defined with transient keyword is not serialized during serialization process.This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0).

In case of **static Variables:-** A variable defined with static keyword is not serialized during serialization process.This variable will be loaded with current value defined in the class during deserialization.

Transient Vs Final:
final variables will be participated into serialization directly by their values.
Hence declaring a final variable as transient there is no use.
//the compiler assign the value to final variable

example:

```
final int x= 10;
int y = 20;
System.out.println(x);// compiler will replace this as System.out.println(10)->10
because x is final.
System.out.println(y);//20
```

- ## What is a Process in Java?
  - A Process in Java is an executing program with its own memory space managed by the operating system.
  - In the simplest terms, a process is a program in execution. Each process has its own memory space, including the heap and stack memory. Consider it as a separate entity entirely. The Operating System manages these processes, deciding when they run and managing their resources. Here's an example of starting a new process in Java:

```
Process process = Runtime.getRuntime().exec("notepad.exe");
```

The above code starts a new process that opens Notepad.

- **What is a Thread in Java?**
  - A Thread in Java is the smallest unit of execution within a process. Multiple threads can exist within a single process, sharing the process's memory space.
  - A thread, on the other hand, is the smallest unit of execution within a process. If a process is a program in action, a thread is a pathway through which the program runs. A single process can have multiple threads, all sharing the same memory space. This shared memory model enables threads to communicate with each other more easily compared to processes. Below is an example of creating a new thread in Java:

```
Thread thread = new Thread(){
   public void run(){
      System.out.println("Thread Running");
   }
};

thread.start();
```

The run method contains the code that the thread will execute.

- **What is the main difference between a Process and a Thread in Java?**
  - The main difference lies in their memory space: Processes have separate memory spaces, while threads within the same process share memory.
  - The major difference between processes and threads lies in their memory space and communication method. Processes have separate memory spaces, and inter-process communication is slower and more complex due to the need to protect memory spaces. Threads, however, share memory space, leading to faster and more straightforward communication.

S.NO	Process	Thread
1	Process means any program is in execution.	Thread means a segment of a process.
2	The process takes more time to terminate.	The thread takes less time to terminate.
3	It takes more time for creation.	It takes less time for creation.
4	It also takes more time for context switching.	It takes less time for context switching.
5	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
6.	Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
7	The process is isolated.	Threads share memory.
8	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
9	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
10	If one process is blocked then it will not affect the execution of other processes	If a user-level thread is blocked, then all other user-level threads are blocked.
11	The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.
12	Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.
13	A system call is involved in it.	No system call is involved, it is created using APIs.
14	The process does not share data with each other.	Threads share data with each other.

1. What is Concurrency?

Concurrency is essentially applicable when we talk about a minimum of two tasks or more. When an application is capable of executing two tasks virtually at the same time, we call it a concurrent application. This is why **concurrency is also referenced as virtual parallelism**.

Though, in this case, tasks look like they are running simultaneously, but essentially they MAY not. They take advantage of the CPU time-slicing feature of the operating system where each task runs part of its task and then goes to the waiting state. When the first task is waiting, the CPU is assigned to the second task to complete its part of the task.

The operating system based on the priority of tasks, thus, assigns CPU and other computing resources e.g. memory; turn by turn to all tasks and gives them a chance to complete. To the end-user, it seems that all tasks are running in parallel. This helps by concurrency solve multiple tasks faster.

2. What is Parallelism?

Parallelism does not require two tasks to exist. It, literally, physically runs parts of tasks OR multiple tasks, at the same time using the multi-core infrastructure of the CPU, by assigning one core to each task or sub-task.

Generally, in the case of parallelism, a task is split into subtasks across multiple CPU cores. These subtasks are computed in parallel and each of them represents a partial solution for the given task. By joining these partial solutions, we obtain the final solution. Ideally, solving a task in parallel should result in less wall-clock time than in the case of solving the same task sequentially.

In a nutshell, **in parallelism, at least two threads are running at the same time** which means that parallelism can solve a single task faster. Parallelism requires the hardware with multiple processing units, essentially. In a single-core CPU, we may get concurrency but NOT parallelism.

3. Differences between concurrency vs. parallelism

Now let's list down the remarkable differences between concurrency and parallelism.

- Concurrency is when two tasks can start, run, and complete in overlapping time periods. Parallelism is when tasks literally run at the same time, eg. on a multi-core processor.
- Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.
- Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.
- An application can be concurrent but not parallel, which means that it processes more than one task at the same time, but no two tasks are executed at the same time instant.
- An application can be parallel but not concurrent, which means that it processes multiple sub-tasks of a task in a multi-core CPU at the same time.
- An application can be neither parallel nor concurrent, which means that it processes all tasks one at a time, sequentially.
- An application can be both parallel and concurrent, which means that it processes multiple tasks concurrently in a multi-core CPU at the same time.
- Typically, we measure parallelism efficiency in *latency* (the amount of time needed to complete the task), while the efficiency of concurrency is measured in *throughput* (the number of tasks that we can solve).

Multithreading in Java:

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

- Extending the Thread class
- Implementing the Runnable Interface

Thread creation by extending the Thread class:

We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

Thread creation by implementing the Runnable Interface:

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

Thread Class vs Runnable Interface:

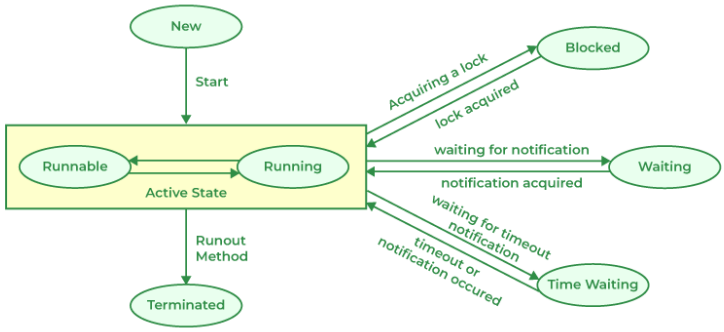
- If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
- We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
- Using runnable will give you an object that can be shared amongst multiple threads.

Lifecycle and States of a Thread in Java:

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

- New State
- Runnable State
- Blocked State
- Waiting State
- Timed Waiting State
- Terminated State

The diagram shown below represents various states of a thread at any instant in time.



Life Cycle of a Thread

There are multiple states of the thread in a lifecycle as mentioned below:

- ***New Thread:*** *When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.*
- ***Runnable State:*** *A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.*
  A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
- ***Blocked:*** *The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.*
- ***Waiting state:*** *The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.*
- ***Timed Waiting:*** *A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.*
- ***Terminated State:*** *A thread terminates because of either of the following reasons:*
  * *Because it exits normally. This happens when the code of the thread has been entirely executed by the program.*
  * *Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.*

Implementing the Thread States in Java

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread, as a summary of which is given below:

1. New: Thread state for a thread that has not yet started.

public static final Thread.State NEW

2. Runnable: Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as a processor.

public static final Thread.State RUNNABLE

3. Blocked: Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling Object.wait().

public static final Thread.State BLOCKED

4. Waiting: Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

* Object.wait with no timeout
* Thread.join with no timeout
* LockSupport.park

public static final Thread.State WAITING

5. Timed Waiting: Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

* Thread.sleep
* Object.wait with timeout
* Thread.join with timeout
* LockSupport.parkNanos
* LockSupport.parkUntil

public static final Thread.State TIMED_WAITING

6. Terminated: Thread state for a terminated thread. The thread has completed execution.

Declaration: public static final Thread.State TERMINATED

Thread Scheduler:

If more than one thread is waiting for a chance to run, the Thread Scheduler will determine which thread should be executed. The exact algorithm followed by the Thread scheduler will not be determined, as the algorithm followed by each Thread scheduler varies from JVM to JVM. Hence, in multithreading, we can't guarantee the exact output; every time we run the code, we will get different outputs.

Algorithms used by the Thread Scheduler:

Round-Robin Scheduling:

In this algorithm, each thread is given a fixed time slice (quantum) to execute. When the time slice expires, the scheduler interrupts the thread and moves it to the back of the queue. The next thread in line gets a chance to run.

Priority-Based Scheduling:

Threads are assigned priority levels, and the scheduler selects the highest-priority thread to execute. This approach can lead to priority inversion, where lower-priority threads block higher-priority ones, impacting overall system performance. To mitigate this, techniques like priority inheritance and priority ceiling protocols are employed.

Shortest Job Next (SJN) Scheduling:

Also known as Shortest Job First (SJF) or shortest remaining time scheduling, this algorithm selects the thread with the smallest execution time remaining. SJN aims to minimize average waiting time and turnaround time, but it requires knowledge of thread execution times, which can be challenging to estimate accurately.

First-Come, First-Served (FCFS) Scheduling:

Threads are executed in the order they arrive in the ready queue. While simple to implement, FCFS may lead to the "convoy effect," where a long-running thread prevents shorter tasks from executing, causing inefficiencies.

Multilevel Queue Scheduling:

Threads are divided into multiple queues based on priority, and each queue may have its own scheduling algorithm. This approach provides differentiation between threads of varying importance, such as interactive and background tasks.

Completely Fair Scheduler (CFS):

Found in the Linux kernel, CFS allocates CPU time based on the concept of fairness among threads. It attempts to distribute CPU time proportionally among threads, ensuring that each thread gets its fair share over time.

Deadline-Based Scheduling:

Primarily used in real-time systems, this algorithm assigns deadlines to threads and schedules them to meet their respective deadlines. Threads with tighter deadlines are given priority.

Multicore Scheduling:

With the advent of multi-core processors, thread scheduling extends to distributing threads across multiple cores. Algorithms focus on load balancing, minimizing contention, and optimizing cache utilization.

User-Level Threading Schedulers:

User-level threading (ULT) schedulers operate at the application level rather than within the operating system kernel. In this approach, the application itself manages its own threads and scheduling, bypassing the kernel's thread management. User-level threads provide a level of control and flexibility for application developers, but they also come with trade-offs in terms of efficiency and system interaction.

Time-Sliced Scheduling:

Time-sliced scheduling, also known as time-sharing or round-robin scheduling, involves dividing CPU time into fixed intervals called time slices or quanta. Each thread is allocated a time slice during which it can execute on

the CPU. Once a thread's time slice expires, it is preemptively moved out of the CPU, and another thread is given a chance to execute. The preempted thread is placed at the end of the scheduling queue and will receive another time slice when its turn comes up again. Time-sliced scheduling ensures that all threads get a fair share of CPU time, preventing any single thread from monopolizing the CPU for extended periods. This approach is particularly effective for scenarios where responsiveness and fairness are essential, such as interactive multitasking environments.

Preemptive Scheduling:

Preemptive scheduling takes the concept of time slicing further by allowing the scheduler to forcibly interrupt a running thread and switch to another thread. This interruption is known as preemption. In preemptive scheduling, threads can be preempted at any time, even before their time slice expires, based on certain events or priorities. Preemptive scheduling introduces finer control over thread execution and enables the operating system to respond quickly to high-priority tasks or events. It is especially useful in scenarios where real-time responsiveness, priority-based execution, and resource allocation are critical.

Working of Thread Scheduler:

Thread States:

Threads typically go through different states during their lifecycle, including "running," "ready," "blocked," and "terminated." The scheduler manages transitions between these states based on thread behaviour and external events.

Thread Queue Management:

Threads that are ready to execute but are waiting for CPU time are placed in a queue known as the "ready queue." The scheduler maintains this queue, which holds threads with varying priorities or characteristics.

Scheduling Criteria:

The scheduler uses various criteria to make decisions about which thread to run next. These criteria can include thread priorities, execution history, expected CPU burst times, and any special requirements (e.g., real-time constraints).

Scheduling Algorithms:

Different scheduling algorithms dictate how threads are selected from the ready queue for execution. Common algorithms include:

1. Round-Robin Scheduling: Threads are given a fixed time slice (quantum) to execute, and they rotate in and out of the CPU in a circular fashion.

2. Priority-Based Scheduling: Threads are assigned priorities, and the scheduler runs the highest-priority thread that is ready to execute.

3. Shortest Job Next (SJN) Scheduling: The thread with the shortest estimated execution time is selected next.

4. Multilevel Queue Scheduling: Threads are divided into priority-based queues, and the scheduler chooses threads from different queues based on their priorities.

5. Preemption: Many modern schedulers use preemption, which allows the scheduler to forcibly interrupt a running thread to give another thread a chance to execute. Preemption is essential for enforcing priority-based scheduling and ensuring that high-priority threads get CPU time when needed.

Context Switching:

When the scheduler decides to switch from one thread to another, it performs a context switch. During a context switch, the current thread's state is saved, and the state of the next thread to be executed is loaded. This involves updating registers, memory mappings, and other relevant information.

Interrupt Handling:

The scheduler interacts with hardware interrupts and timers to handle events such as I/O completion, timeouts, or hardware interrupts. These events can trigger context switches and affect thread execution.

Resource Management:

The scheduler manages various system resources that threads might contend for, such as memory, I/O devices, and synchronization primitives like locks and semaphores.

Dynamic Adjustment:

Some schedulers dynamically adjust priorities or time slices based on factors like thread behavior, past execution history, or system load. This adaptive approach helps optimize performance under varying conditions.

Real-Time Scheduling:

In real-time systems, where meeting deadlines is critical, the scheduler ensures that threads with strict timing constraints are given priority to execute within their specified time frames.

Java Thread Priority in Multithreading:

As we already know java being completely object-oriented works within a multithreading environment in which thread scheduler assigns the processor to a thread based on the priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly. **Priorities in threads** is a concept where each thread is having a priority which in layman's language one can say every object is having priority here which is represented by numbers ranging from 1 to 10.

- The default priority is set to 5 as excepted.
- Minimum priority is set to 1.
- Maximum priority is set to 10.

Here 3 constants are defined in it namely as follows:

- public static int NORM_PRIORITY
- public static int MIN_PRIORITY
- public static int MAX_PRIORITY

The accepted value of priority for a thread is in the range of 1 to 10.

- **public final int getPriority():** java.lang.Thread.getPriority() method returns priority of given thread.
- **public final void setPriority(int newPriority):** java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

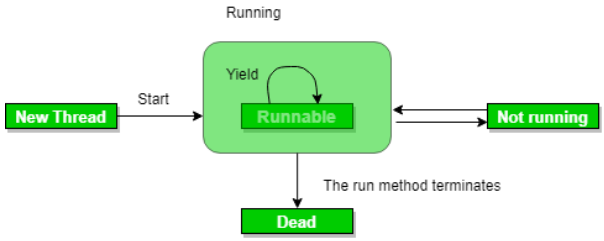
Methods to prevent execution of a thread(Temporary):

We can prevent the execution of a thread by using one of the following methods of the Thread class. All three methods will be used to prevent the thread from execution.

1. yield() Method:

Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. The completion time for thread t1 is 5 hours and the completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent the execution of a thread in between if something important is pending. yield() helps us in doing so.

The **yield()** basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.



Use of yield method:

- Whenever a thread calls **java.lang.Thread.yield** method gives hint to the thread scheduler that it is ready to pause its execution. The thread scheduler is free to ignore this hint.
- If any thread executes the yield method, the thread scheduler checks if there is any thread with the same or high priority as this thread. If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – the current thread will keep executing.
- Once a thread has executed the yield method and there are many threads with the same priority is waiting for the processor, then we can't specify which thread will get the execution chance first.
- The thread which executes the yield method will enter in the Runnable state from Running state.
- Once a thread pauses its execution, we can't specify when it will get a chance again it depends on the thread scheduler.
- The underlying platform must provide support for preemptive scheduling if we are using the yield method.

2. sleep() Method:

This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Syntax:

```
//  sleep for the specified number of milliseconds
public static void sleep(long millis) throws InterruptedException


//sleep for the specified number of milliseconds plus nano seconds
public static void sleep(long millis, int nanos) throws InterruptedException
```

Note:

- *Based on the requirement we can make a thread to be in a sleeping state for a specified period of time*
- *Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power-saving mode).*

3. join() Method

The join() method of a Thread instance is used to join the start of a thread's execution to the end of another thread's execution such that a thread does not start running until another thread ends. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing. The join() method waits at most this many milliseconds for this thread to die. A timeout of 0 means to wait forever

Syntax:

```
// waits for this thread to die.
public final void join() throws InterruptedException


// waits at most this much milliseconds for this thread to die
public final void join(long millis) throws InterruptedException


// waits at most milliseconds plus nanoseconds for this thread to die.
The java.lang.Thread.join(long millis, int nanos)
```

Note:

- *If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.*
- *Giving a timeout within join(), will make the join() effect to be nullified after the specific timeout.*

Feature	`wait()`	`sleep()`	`yield()`
Class	Can be called on any object	`Thread` class method	`Thread` class method
Lock Release	Releases the lock acquired on the object	Does not release any locks	Does not release any locks
Execution Pause	Pauses the execution of the current thread	Pauses the execution of the current thread	Offers a hint to the scheduler to pause execution
Time Unit	Does not accept a specific time unit	Accepts time duration in milliseconds	Does not accept a specific time unit
Synchronization	Must be called within a synchronized context	Does not require synchronization	Does not require synchronization
Interruption	Can be interrupted by another thread	Can be interrupted by another thread	Cannot be interrupted by another thread
Wake-up Condition	Requires a call to `notify()` or `notifyAll()`	Does not require any wake-up condition	Does not require any wake-up condition
Utilization	Used for inter-thread communication and signaling	Used for introducing delay or pause in a thread	Used for cooperative thread scheduling

Comparison of yield(), join(), sleep() Methods

Property	yield()	join()	sleep()
Purpose	If a thread wants to pass its execution to give chance to remaining threads of the same priority then we should go for yield()	If a thread wants to wait until completing of some other thread then we should go for join()	If a thread does not want to perform any operation for a particular amount of time, then it goes for sleep()
Is it overloaded?	NO	YES	YES

Property	yield()	join()	sleep()
Is it final?	NO	YES	NO
Is it throws?	NO	YES	YES
Is it native?	YES	NO	sleep(long ms)->native & sleep (long ms, int ns)-> non native
Is it static?	YES	NO	YES

Synchronization in Java:

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

Why use Java Synchronization?

Java Synchronization is used to make sure by some synchronization method that only one thread can access the resource at a given point in time.

Java Synchronized Blocks:

Java provides a way of creating threads and synchronizing their tasks using synchronized blocks.

A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object and can only have one thread executed inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Syntax:

```
// the monitor object
synchronized(sync_object)
{
  // Access shared variables and other
  // shared resources
}
```

This synchronization is implemented in Java with a concept called monitors or locks. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Types of Synchronization:

There are two synchronizations in Java mentioned below:

1. Process Synchronization
2. Thread Synchronization

1. Process Synchronization in Java:

Process Synchronization is a technique used to coordinate the execution of multiple processes. It ensures that the shared resources are safe and in order.

2. Thread Synchronization in Java:

Thread Synchronization is used to coordinate and ordering of the execution of the threads in a multi-threaded program. There are two types of thread synchronization are mentioned below:

- Mutual Exclusive
- Cooperation (Inter-thread communication in Java)

Mutual Exclusive:

Mutual Exclusive helps keep threads from interfering with one another while sharing data. There are three types of Mutual Exclusive mentioned below:

- Synchronized method.
- Synchronized block.
- Static synchronization.

Inter-thread Communication in Java:

Inter-thread communication in Java is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

Note: Inter-thread communication is also known as Cooperation in Java.

What is Polling, and what are the problems with it?

The process of testing a condition repeatedly till it becomes true is known as polling. Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, a certain action is taken. This wastes many CPU cycles and makes the implementation inefficient.

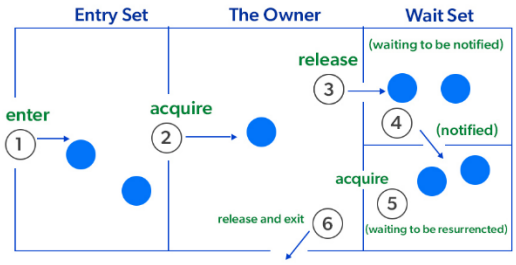
For example, in a classic queuing problem where one thread is producing data, and the other is consuming it.

How Java multi-threading tackles this problem?

To avoid polling, Java uses three methods, namely, wait(), notify(), and notifyAll(). All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

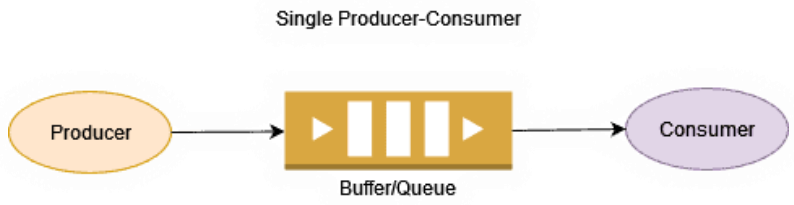
- wait(): It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().

- notify(): It wakes up one single thread called wait() on the same object. It should be noted that calling notify() does not give up a lock on a resource.
- notifyAll(): It wakes up all the threads called wait() on the same object.



Producer-Consumer Problem:

It is also known as bounded-buffer problem. Producer and Consumer are two separate processes. Both processes share a common buffer or queue. The producer continuously produces certain data and pushes it onto the buffer, whereas the consumer consumes those data from the buffer.



Let's review a diagram showing this simple scenario:

Inherently, this problem has certain complexities to deal with:

- Both producer and consumer may try to update the queue at the same time. This could lead to data loss or inconsistencies.
- Producers might be slower than consumers. In such cases, the consumer would process elements fast and wait.
- In some cases, the consumer can be slower than the producer. This situation leads to a queue overflow issue.
- In real scenarios, we may have multiple producers, multiple consumers, or both. This may cause the same message to be processed by different consumers.

We need to handle resource sharing and synchronization to solve a few complexities:

- Synchronization on queue while adding and removing data
- When the queue is empty, the consumer has to wait until the producer adds new data to the queue
- When the queue is full, the producer has to wait until the consumer consumes data and the queue has some empty buffer

The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

In this problem, we need two threads, Thread t1 (produces the data) and Thread t2 (consumes the data). However, both the threads shouldn't run simultaneously.

Daemon Thread in Java:

In Java, daemon threads are low-priority threads that run in the background to perform tasks such as garbage collection or provide services to user threads. The life of a daemon thread depends on the mercy of user threads, meaning that when all user threads finish their execution, the Java Virtual Machine (JVM) automatically terminates the daemon thread.

To put it simply, daemon threads serve user threads by handling background tasks and have no role other than supporting the main execution.

Example of Daemon Thread in Java:

Some examples of daemon threads in Java include garbage collection (gc) and finalizer. These threads work silently in the background, performing tasks that support the main execution without interfering with the user's operations.

Properties of Java Daemon Thread:

1. **No Preventing JVM Exit**: Daemon threads cannot prevent the JVM from exiting when all user threads finish their execution. If all user threads complete their tasks, the JVM terminates itself, regardless of whether any daemon threads are running.
2. **Automatic Termination**: If the JVM detects a running daemon thread, it terminates the thread and subsequently shuts it down. The JVM does not check if the daemon thread is actively running; it terminates it regardless.
3. **Low Priority**: Daemon threads have the lowest priority among all threads in Java.

Default Nature of Daemon Thread:

By default, the main thread is always a non-daemon thread. However, for all other threads, their daemon nature is inherited from their parent thread. If the parent thread is a daemon, the child thread is also a daemon, and if the parent thread is a non-daemon, the child thread is also a non-daemon.

Note: *Whenever the last non-daemon thread terminates, all the daemon threads will be terminated automatically.*

Methods of Daemon Thread

1. void setDaemon(boolean status):

This method marks the current thread as a daemon thread or user thread. Setting a user thread as a daemon can be done using the**'tU.setDaemon(true)'**, while setting a daemon thread as a user thread can be done using the **'tD.setDaemon(false)'**.

Syntax: public final void setDaemon(boolean on);

Parameters:

- **on:** If true, marks this thread as a daemon thread.

Exceptions:

- **IllegalThreadStateException**: if only this thread is active.
- **SecurityException:** if the current thread cannot modify this thread.

2. boolean isDaemon():

This method is used to check that the current thread is a daemon. It returns true if the thread is Daemon. Else, it returns false.

Syntax: public final boolean isDaemon();

Returns: This method returns true if this thread is a daemon thread; false otherwise

Daemon vs. User Threads

- **Priority**: When only daemon threads remain in a process, the JVM exits. This makes sense because when only daemon threads are running, there is no need for a daemon thread to provide a service to another thread.
- **Usage**: Daemon threads are primarily used to provide background support to user threads. They handle tasks that support the main execution without interfering with the user's operations.

Understanding daemon threads is essential for Java developers to effectively manage thread behavior and optimize application performance.

Java.lang.ThreadGroup class in Java:

ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads. The thread group form a tree in which every thread group except the initial thread group has a parent. A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.

Constructors:

1. **public ThreadGroup (String name):** Constructs a new thread group. The parent of this new group is the thread group of the currently running thread.

   Throws: SecurityException - if the current thread cannot create a thread in the specified thread group.

2. **public ThreadGroup (ThreadGroup parent, String name):** Creates a new thread group. The parent of this new group is the specified thread group.

   Throws: NullPointerException - if the thread group argument is null.

   SecurityException - if the current thread cannot create a thread in the specified ThreadGroup.

Methods:

1. **int activeCount():** This method returns the number of threads in the group plus any group for which this thread is parent.
2. **int activeGroupCount():** This method returns an estimate of the number of active groups in this thread group.
3. **void checkAccess():** Causes the security manager to verify that the invoking thread may access and/ or change the group on which **checkAccess()** is called.
4. **void destroy():** Destroys the thread group and any child groups on which it is called.
5. **int enumerate(Thread group[]):** The thread that comprise the invoking thread group are put into the group array.
6. **int enumerate(Thread[] group, boolean recurse):** The threads that comprise the invoking thread group are put into the group array. If all is **true**, then threads in all subgroups of the thread are also put into group.
7. **int enumerate(ThreadGroup[] group):** The subgroups of the evoking thread group are put into the group array.
8. **int enumerate(ThreadGroup[] group, boolean all):** The subgroups of the invoking thread group are put into the group array. If all is **true**, then all subgroups of the subgroups(and so on) are also put into group.
9. **int getMaxPriority():** Returns the maximum priority setting for the group.
10. **String getName():** This method returns the name of the group
11. **ThreadGroup getParent():** Returns null if the invoking ThreadGroup object has no parent. Otherwise, it returns the parent of the invoking object.
12. **void interrupt():** Invokes the **interrupt()** methods of all threads in the group.
13. **boolean isDaemon():** Tests if this thread group is a daemon thread group. A daemon thread group is automatically destroyed when its last thread is stopped or its last thread group is destroyed.
14. **boolean isDestroyed():** This method tests if this thread group has been destroyed.
15. **void list():** Displays information about the group.
16. **boolean parentOf(ThreadGroup group):** This method tests if this thread group is either the thread group argument or one of its ancestor thread groups. **void setDaemon(boolean isDaemon):** This method changes the daemon status of this thread group. A daemon thread group is automatically destroyed when its last thread is stopped or its last thread group is destroyed.
17. **void setDaemon(boolean isDaemon):** This method changes the daemon status of this thread group. A daemon thread group is automatically destroyed when its last thread is stopped or its last thread group is destroyed.
18. **void setMaxPriority(int priority):** Sets the maximum priority of the invoking group to priority.
19. **String toString():** This method returns a string representation of this Thread group.

Lock Interface:

A java.util.concurrent.locks.Lock is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block. Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. ReentrantLock is one such implementation of Lock interface.

Here is the simple use of Lock interface.

```
Lock lock = new ReentrantLock();
lock.lock();
//critical section
lock.unlock();
```

First a Lock is created. Then it's lock() method is called. Now the Lock instance is locked. Any other thread calling lock() will be blocked until the thread that locked the lock calls unlock(). Finally unlock() is called, and the Lock is now unlocked so other threads can lock it.

Difference between Lock and *synchronized* Keyword:

The main differences between a Lock and a synchronized block are:

- Having a timeout trying to get access to a synchronized block is not possible. Using Lock.tryLock(long timeout, TimeUnit timeUnit), it is possible.
- The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() and unlock() in separate methods.

Reentrant Lock in Java:

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

As the name says, ReentrantLock allows threads to enter into the lock on a resource more than once. When the thread first enters into the lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlocks request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

ReentrantLock() Methods:

- **lock():** Call to the lock() method increments the hold count by 1 and gives the lock to the thread if the shared resource is initially free.
- **unlock():** Call to the unlock() method decrements the hold count by 1. When this count reaches zero, the resource is released.

- **tryLock():** If the resource is not held by any other thread, then call to tryLock() returns true and the hold count is incremented by one. If the resource is not free, then the method returns false, and the thread is not blocked, but exits.
- **tryLock(long timeout, TimeUnit unit):** As per the method, the thread waits for a certain time period as defined by arguments of the method to acquire the lock on the resource before exiting.
- **lockInterruptibly():** This method acquires the lock if the resource is free while allowing for the thread to be interrupted by some other thread while acquiring the resource. It means that if the current thread is waiting for the lock but some other thread requests the lock, then the current thread will be interrupted and return immediately without acquiring the lock.
- **getHoldCount():** This method returns the count of the number of locks held on the resource.
- **isHeldByCurrentThread():** This method returns true if the lock on the resource is held by the current thread.
- **hasQueuedThread()**: This Method Queries whether the given thread is waiting to acquire this lock.
- **isLocked():** Queries if this lock is held by any thread.
- **newCondition()**: Returns a Condition instance for use with this Lock instance.

Important Points:

1. One can forget to call the unlock() method in the finally block leading to bugs in the program. Ensure that the lock is released before the thread exits.
2. The fairness parameter used to construct the lock object decreases the throughput of the program.

The ReentrantLock is a better replacement for synchronization, which offers many features not provided by synchronized. However, the existence of these obvious benefits are not a good enough reason to always prefer ReentrantLock to synchronize. Instead, make the decision on the basis of whether you need the flexibility offered by a ReentrantLock.

Thread Pools in Java:

A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

- Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
- They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
- To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

Tuning Thread Pool:

The optimum size of the thread pool depends on the number of processors available and the nature of the tasks. On a N processor system for a queue of only computation type processes, a maximum thread pool size of N or N+1 will achieve the maximum efficiency.But tasks may wait for I/O and in such a case we take into account the ratio of waiting time(W) and service time(S) for a request; resulting in a maximum pool size of N*(1+ W/S) for maximum efficiency.

The thread pool is a useful tool for organizing server applications. It is quite straightforward in concept, but there are several issues to watch for when implementing and using one, such as deadlock, resource thrashing. Use of executor service makes it easier to implement.

Syntax: Executor executor = Executors.newSingleThreadExecutor();

Submit() Method:

1. **Purpose**: The submit() method is used to submit a task for execution in the thread pool. It can accept a Runnable, Callable, or a Future task.
2. **Return Type**:
    - When a Runnable is submitted, it returns a Future<?> object that can be used to check if the task is complete and to retrieve the result (which will be null for Runnable tasks).
    - When a Callable is submitted, it returns a Future<T> object that represents the result of the computation.
3. **Usage**:
    - The submit() method allows you to submit tasks that can be executed asynchronously. The tasks are queued and executed by the available threads in the pool.
    - It provides a way to handle exceptions thrown during task execution, as you can check the status of the Future object.

Shutdown() Method:

1. **Purpose**: The shutdown() method is used to initiate an orderly shutdown of the thread pool. It prevents new tasks from being submitted and allows previously submitted tasks to complete.
2. **Behavior**:
    - After calling shutdown(), the thread pool will not accept any new tasks. However, it will continue to execute all previously submitted tasks.
    - Once all tasks have completed, the thread pool will terminate.
3. **Usage**:
    - It is important to call shutdown() when you are done using the thread pool to free up resources and avoid memory leaks.
    - If you want to immediately stop all actively executing tasks, you can use shutdownNow(), which attempts to stop all actively executing tasks and returns a list of the tasks that were waiting to be executed.

Callable and Future in Java:

There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable. However, one feature lacking in Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes. For supporting this feature, the Callable interface is present in Java.

Future:

When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned. How will the program store and obtain this result later? For this, a Future object can be used. Think of a Future as an object that holds the result – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a Future is basically one way the main thread can keep track of the progress and result from other threads.

Runnable interface	Callable interface
It is a part of java.lang package since Java 1.0	It is a part of the java.util.concurrent package since Java 1.5.
It cannot return the result of computation.	It can return the result of the parallel processing of a task.
It cannot throw a checked Exception.	It can throw a checked Exception.
In a runnable interface, one needs to override the run() method in Java.	In order to use Callable, you need to override the call()

Collections in Java

Any group of individual objects that are represented as a single unit is known as a Java Collection of Objects. In Java, a separate framework named the *"Collection Framework"* has been defined in JDK 1.2 which holds all the Java Collection Classes and Interface in it.

In Java, the Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main "root" interfaces of Java collection classes.

What is a Framework in Java?

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

Need for a Separate Collection Framework in Java

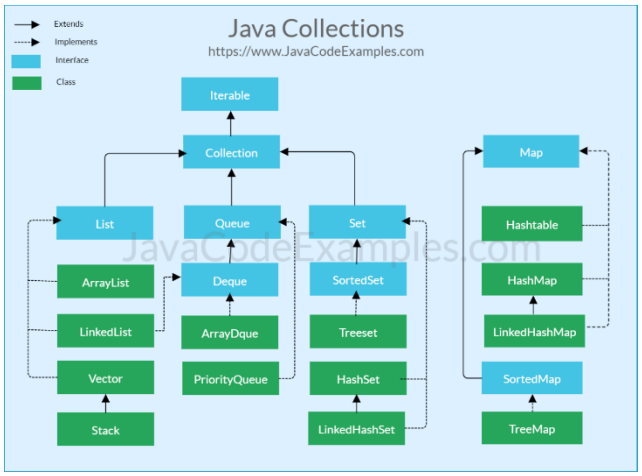
Before the Collection Framework(or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were **Arrays** or **Vectors**, or **Hashtables**. All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them. And also, it is very difficult for the users to remember all the different **methods**, syntax, and **constructors** present in every collection class.

Advantages of the Java Collection Framework

Since the lack of a collection framework gave rise to the above set of disadvantages, the following are the 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. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.
3. **Increases program speed and quality:** Increases performance by providing high-performance implementations of useful data structures and algorithms because in this case, the programmer need not think of the best implementation of a specific data structure. He can simply use the best implementation to drastically boost the performance of his algorithm/program.

Hierarchy of the Collection Framework in Java:



Before understanding the different components in the above framework, let's first understand a class and an interface.

- **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.

Java Collections API Interfaces:

Java collection interfaces are the foundation of the Java Collections Framework. Note that all the core collection interfaces are generic; for example public interface Collection<E>. The <E> syntax is for Generics and when we declare Collection, we should use it to specify the type of Object it can contain. It helps in reducing run-time errors by type-checking the Objects at compile-time.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. If an unsupported operation is invoked, a collection implementation throws an UnsupportedOperationException.

1. Collection interface: This is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn't provide any direct implementations of this interface.

The interface has methods to tell you how many elements are in the collection (size, isEmpty), to check whether a given object is in the collection (contains), to add and remove an element from the collection (add, remove), and to provide an iterator over the collection (iterator). Collection interface also provides bulk operations methods that work on the entire collection – containsAll, addAll, removeAll, retainAll, clear. The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input.

2. Iterator Interface: Iterator interface provides methods to iterate over the elements of the Collection. We can get the instance of iterator using iterator() method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Iterators in collection classes implement **Iterator Design Pattern**.

3. Set Interface: Set is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards. The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet. Set interface doesn't allow random-access to an element in the Collection. You can use iterator or foreach loop to traverse the elements of a Set.

4. List Interface: List is an ordered collection and can contain duplicate elements. You can access any element from its index. List is more like array with dynamic length. List is one of the most used Collection type. ArrayList and LinkedList are implementation classes of List interface.

List interface provides useful methods to add an element at a specific index, remove/replace element based on the index and to get a sub-list using the index.

Collections class provide some useful algorithm for List – sort, shuffle, reverse, binarySearch etc.

5. Queue Interface: Queue is a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue.

6. Dequeue Interface: A linear collection that supports element insertion and removal at both ends. The name deque is short for "double-ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element.

7. Map Interface: Java Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap.

The basic operations of Map are put, get, containsKey, containsValue, size, and isEmpty.

8. ListIterator Interface: An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

Java ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

9. SortedSet Interface: SortedSet is a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

10. SortedMap Interface: A map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

Java Collections Classes:

Java Collections framework comes with many implementation classes for the interfaces. Most common implementations are ArrayList, HashMap and HashSet. Java 1.5 included Concurrent implementations; for

example ConcurrentHashMap and CopyOnWriteArrayList. Usually Collection classes are not thread-safe and their iterator is fail-fast. In this section, we will learn about commonly used collection classes.

1. HashSet Class:

Java HashSet is the basic implementation the Set interface that is backed by a HashMap. It makes no guarantees for iteration order of the set and permits the null element.

This class offers constant time performance for basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. We can set the initial capacity and load factor for this collection. The load factor is a measure of how full the hash map is allowed to get before its capacity is automatically increased.

2. TreeSet Class:

A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

Refer: Java Comparable Comparator

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove, and contains).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal.

3. ArrayList Class:

Java ArrayList is the resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and list iterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Further reading: Java ArrayList and Iterator

4. LinkedList Class:

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as expected for a doubly-linked list. Operations that index into the list will traverse the list from the start or the end, whichever is closer to the specified index.

5. HashMap Class:

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations and permits null values and the null key. HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits null. This class makes no guarantees for the order of the map.

This implementation provides constant-time performance for the basic operations (get and put). It provides constructors to set initial capacity and load factor for the collection.

Further Read: HashMap vs ConcurrentHashMap

Few important points about **HashMap :**

1. **HashMap** uses its static inner class **Node<K,V>** for storing the entries into the map.
2. **HashMap** allows at most one **null** key and multiple **null** values.
3. The **HashMap** class does not preserve the order of insertion of entries into the map.
4. **HashMap** has multiple buckets or bins which contain a head reference to a singly linked list. That means there would be as many linked lists as there are buckets. Initially, it has a bucket size of 16 which grows to 32 when the number of entries in the map crosses the 75%. (That means after inserting in 12 buckets bucket size becomes 32)
5. **HashMap** is almost similar to Hashtable except that it's **unsynchronized** and allows at max one null key and multiple null values.
6. **HashMap** uses **hashCode()** and **equals()** methods on keys for the **get** and **put** operations. So HashMap key objects should provide a good implementation of these methods.
7. That's why the **Wrapper classes** like **Integer** and String classes are a good choice for keys for HashMap as they are immutable and their object state won't change over the course of the execution of the program.

Internal Working Of Hashmap:

HashMap stores the data in the form of key-value pairs. Each key-value pair is stored in an object of Entry<K, V> class. Entry<K, V> class is the static inner class of **HashMap** which is defined like below.

```
static class Entrsy<K,V> implements Map.Entry<K,V>
{
    final K key;
    V value;
    Entry<K,V> next;
    int hash;
    //Some methods are defined here
}
```

As you see, this inner class has four fields. *key*, *value*, next, and hash.

key: It stores the key of an element and its final.

value: It holds the value of an element.

next: It holds the pointer to the next key-value pair. ***This attribute makes the key-value pairs stored as a linked list.***

hash: It holds the hashcode of the key.

These Entry objects are stored in an array called table[]. This array is initially of size 16. It is defined like below.

```
/* The table, resized as necessary. Length MUST Always be a power of two.
   */
  transient Entry<K,V>[] table;
```

- To summarize the whole *HashMap* structure, each key-value pair is stored in an object of *Entry<K, V>* class. This class has an attribute called *next* which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these *Entry<K, V>* objects are stored in an array called *table[]*. The below image best describes the *HashMap* structure.

What Is Hashing?

The whole HashMap data structure is based on the principle of **Hashing**. Hashing is nothing but the function or algorithm or method which when applied on any object/variable returns a unique integer value representing that object/variable. This unique integer value is called hash code. Hash function or simply hash is said to be the best if it returns the same hash code each time it is called on the same object. Two objects can have the same hash code.

Whenever you insert new key-value pair using the **put()** method, HashMap blindly doesn't allocate a slot in the table[] array. Instead, it calls a hash function on the key. HashMap has its own hash function to calculate the hash code of the key. This function is implemented so that it overcomes poorly implemented hashCode() methods. Below is the implementation code of hash().

```
/**
   * Retrieve object hash code and applies a supplemental hash function to the
   * result hash, which defends against poor quality hash functions.  This is
   * critical because HashMap uses power-of-two length hash tables, that
   * otherwise encounter collisions for hashCodes that do not differ
   * in lower bits. Note: Null keys always map to hash 0, thus index 0.
   */
  final int hash(Object k) {
    int h = 0;
    if (useAltHashing) {
```

```
        if (k instanceof String) {
            return sun.misc.Hashing.stringHash32((String) k);
        }
        h = hashSeed;
    }
    h ^= k.hashCode();
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

After calculating the hash code of the key, it calls **indexFor()** method by passing the hash code of the key and length of the **table[]** array. This method returns the index in the **table[]** array for that particular key-value pair.

```
/*
 * Returns index for hash code h.
 */
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

Note: To have a high-performance hashMap we need good implementation of hashCode() and equals() method along with hash function.

hashCode():

The hash function is a function that maps a key to an index in the hash table. It obtains an index from a key and uses that index to retrieve the value for a key.

A hash function first converts a search key (object) to an integer value (known as hash code) and then compresses the hash code into an index to the hash table.

The Object class (root class) of Java provides a hashCode method that other classes need to override. hashCode() method is used to retrieve the hash code of an object. It returns an integer hash code by default that is a memory address (memory reference) of an object.

The general syntax for the hashCode() method is as follows:

public native hashCode()

The general syntax to call hashCode() method is as follows:

int h = key.hashCode();

The value obtained from the hashCode() method is used as the bucket index number. The bucket index number is the address of the entry (element) inside the map. If the key is null then the hash value returned by the hashCode() will be 0.

equals():

The equals() method is a method of the Object class that is used to check the equality of two objects. HashMap uses the equals() method to compare Keys to whether they are equal or not.

The equals() method of the Object class can be overridden. If we override the equals() method, it is mandatory to override the hashCode() method.

Put Operation: How put() method of Hashmap works internally in Java?

The put() method of HashMap is used to store the key-value pairs. The syntax of the put() method to add key/value pair is as follows:

hashmap.put(key, value);

Let's take an example where we will insert three (Key, Value) pairs in the HashMap.

```
HashMap<String, Integer> hmap = new HashMap<>();

 hmap.put("John", 20);

 hmap.put("Harry", 5);

 hmap.put("Deep", 10);
```

Let's understand at which index the key-value pairs will be stored into HashMap.

When we call the put() method to add a key-value pair to hashmap, HashMap calculates a hash value or hash code of key by calling its hashCode() method. HashMap uses that code to calculate the bucket index in which key/value pair will be placed.

The formula for calculating the index of the bucket (where n is the size of an array of the bucket) is given below:

Index = hashCode(key) & (n-1);

Suppose the hash code value for "John" is 2657860. Then the index value for "John" is:

Index = 2657860 & (16-1) = 4

The value 4 is the computed index value where the key and value will be stored in HashMap.

Note: Since HashMap allows only one null Key, the hash value returned by the hashCode(key) method will be 0 because the hashcode for null is always 0. The 0th bucket location will be used to place key/value pair.

How is Hash Collision occurred and resolved?

A hash collision occurs when the hashCode() method generates the same index value for two or more keys in the hash table. To overcome this issue, HashMap uses the technique of linked-list.

When hashCode() method produces the same index value for a new Key and the Key that already exists in the hash table, HashMap uses the same bucket index that already contains nodes in the form of a linked list.

A new node is created at the last of the linked list and connects this node object to the existing node object through the LinkedList Hence both Keys will be stored at the same index value.

When a new value object is inserted with an existing Key, HashMap replaces the old value with the current value related to the Key. To do it, HashMap uses the equals() method.

This method checks whether both Keys are equal or not. If Keys are the same, this method returns true and the value of that node is replaced with the current value.

How get() method in HashMap works internally in Java?

The get() method in HashMap is used to retrieve the value by its key. If we don't know the Key, it will not fetch the value. The syntax for calling get() method is as follows:

value = hashmap.get(key);

When the get(K Key) method takes a Key, it calculates the index of the bucket using the method mentioned above. Then that bucket's List is searched for the given key using the equals() method and the final result is returned.

Time Complexity of put() and get() methods:

HashMap stores a key-value pair in constant time which is O(1) for insertion and retrieval. But in the worst case, it can be O(n) when all node returns the same hash value and are inserted into the same bucket.

The traversal cost of n nodes will be O(n) but after the changes made by Java 1.8 version, it can be a maximum of O(log n).

Concept of Rehashing:

Rehashing is a process that occurs automatically by HashMap when the number of keys in the map reaches the threshold value. The threshold value is calculated as **threshold** = capacity * (load factor of 0.75).

In this case, a new size of bucket array is created with more capacity and all the existing contents are copied over to it.

For example:

Load Factor: 0.75

Initial Capacity: 16 (Available Capacity initially)

Threshold = Load Factor * Available Capacity = 0.75 * 16 = 12

When the 13th key-value pair is inserted into the HashMap, HashMap grows its bucket array size to 16*2 = 32.

Now Available capacity: 32

Threshold = Load Factor * Available Capacity = 0.75 * 32 = 24

Next time when 25th key-value pair is inserted into HashMap, HashMap grows its bucket array size to 32*2 = 64 and so on.

Key notes on internal working of HashMap

1. Data structure to store entry objects is an array named table of type Entry.
2. A particular index location in array is referred as bucket, because it can hold the first element of a linkedlist of entry objects.
3. Key object's hashCode() is required to calculate the index location of Entry object.
4. Key object's equals() method is used to maintain uniqueness of keys in map.
5. Value object's hashCode() and equals() method are not used in HashMap's get() and put() methods.
6. Hash code for null keys is always zero, and such entry object is always stored in zero index in Entry[].

6. TreeMap Class:

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put, and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Note that the ordering maintained by a TreeMap, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with equals if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

7. PriorityQueue Class:

Queue processes its elements in FIFO order but sometimes we want elements to be processed based on their priority. We can use PriorityQueue in this case and we need to provide a Comparator implementation while instantiation the PriorityQueue. PriorityQueue doesn't allow null values and it's unbounded. For more details about this, please head over to **Java Priority Queue** where you can check its usage with a sample program.

Fail-Fast and Fail-Safe:

In Java, the terms **fail-fast** and **fail-safe** refer to two different strategies for handling concurrent modifications to collections during iteration. Here's a detailed explanation of both concepts based on the search results.

Fail-Fast Iterators:

1. **Definition**: Fail-fast iterators immediately throw a ConcurrentModificationException if they detect that the underlying collection has been structurally modified after the iterator was created. Structural modifications include adding, removing, or updating elements in the collection.
2. **How It Works**:
    - Fail-fast iterators use an internal counter (often called modCount) to track the number of structural modifications made to the collection.
    - When an iterator is created, it captures the current value of modCount.
    - On each call to the next() method, the iterator checks if modCount has changed. If it has, the iterator throws a ConcurrentModificationException.
3. **Examples**: Common examples of fail-fast iterators include those found in ArrayList, HashMap, and other standard collection classes in Java. Here's an example:

java

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
public class FailFastExample {
   public static void main(String[] args) {
      Map<String, String> cityCode = new HashMap<>();
      cityCode.put("Delhi", "India");
      cityCode.put("Moscow", "Russia");
      cityCode.put("New York", "USA");
      Iterator<String> iterator = cityCode.keySet().iterator();
      while (iterator.hasNext()) {
         System.out.println(cityCode.get(iterator.next()));
         // Adding an element to the map
         cityCode.put("Istanbul", "Turkey"); // This will cause ConcurrentModificationException
      }
   }
}
```

Fail-Safe Iterators:

1. **Definition**: Fail-safe iterators, on the other hand, do not throw exceptions if the underlying collection is modified during iteration. Instead, they operate on a copy (or clone) of the collection, allowing for safe iteration even if the original collection is modified.
2. **How It Works**:
    - Fail-safe iterators create a snapshot of the collection at the time the iterator is created. Any modifications to the original collection do not affect the snapshot.
    - As a result, the iterator can continue to operate without throwing exceptions, even if the original collection is altered.

Key Differences Between Fail-Fast and Fail-Safe

- **Behavior on Modification**:
    - **Fail-Fast**: Throws ConcurrentModificationException if the collection is modified during iteration.
    - **Fail-Safe**: Allows modifications without throwing exceptions, as it iterates over a snapshot of the collection.
- **Use Cases**:
    - **Fail-Fast**: Suitable for scenarios where data integrity is critical, and modifications during iteration should be avoided.
    - **Fail-Safe**: Useful in concurrent environments where you want to allow modifications without interrupting the iteration process.

Conclusion

Understanding the difference between fail-fast and fail-safe iterators is crucial for managing concurrent modifications in Java collections. Fail-fast iterators prioritize immediate feedback on structural changes, while fail-safe iterators provide flexibility in concurrent scenarios by allowing modifications without interruption. This knowledge helps developers choose the right collection type based on their specific use case and concurrency requirements.

Comparable:

Comparable interface is mainly used to sort the arrays (or lists) of **custom objects**. Lists (and arrays) of objects that implement Comparable interface can be sorted automatically by Collections.sort (and Arrays.sort).

The **Comparable** interface is used to define how a class is to be sorted. It is not to be confused with the Comparator interface, which is implemented in a separate class. The Comparable interface is implemented in the class to be sorted.

Syntax:

```
class MyClass implements Comparable<MyClass> {
  // Class body.
```

```
...
@Override public int compareTo(MyClass value)
{
   // Comparison Logic
   ...
   return result;
}
}
```

Applying the Comparable interface to MyClass requires the implements keyword (e.g., Comparable<MyClass>). This interface has a single .compareTo() method that returns an int value based on whether the value of a current class instance (referenced by this) can be logically sorted with the value of another instance of the same class. The compareTo method should follow the contract defined by the Comparable interface. The comparison logic should be consistent with the equals method (i.e., if compareTo returns 0, equals should return true).

Return Value	Meaning
>= 1	this instance > passed instance
0	this instance = passed instance
<= -1	this instance < passed instance

This method determines how items are sorted by methods such as Arrays.sort() and Collections.sort().

Differences:

Difference between Array and Collection

	Array	Collection
1	Arrays are fixed in size and hence once we created an array we are not allowed to increase or decrease the size based on our requirement.	Collections are grow-able in nature and hence based on our requirement we can increase or decrease the size.
2	Arrays can hold both primitives as well as objects.	Collections can hold only objects but not primitive.
3	Performance point of view arrays faster than collection	Performance point of view collections are slower than array
4	Arrays can hold only homogeneous elements.	Collections can hold both homogeneous and heterogeneous elements.
5	Memory point of view arrays are not recommended to use.	Memory point of view collections are recommended to use.
6	For any requirement, there is no ready method available.	For every requirement ready made method support is available.

Collection	Collections
Collection is a interface	Collections is a Utility class
Collection represents a group of objects, known as its elements.	Collections class provides static utility methods for various collections
Basic Collection interface implementations do not inherently support thread-safety.	Collections class provides methods to make specific collections thread-safe
Collection interface has static and default methods (from Java 8)	Collections class has only static methods

Aspect	List	Set
Ordering	Maintains the order of insertion.	Does not guarantee order of elements.
Duplicates	Allows duplicate elements.	Does not allow duplicate elements.
Null Elements	May allow null elements (depends on the implementation).	May allow null elements (depends on the implementation).
Main Implementations	ArrayList, LinkedList, etc.	HashSet, LinkedHashSet, TreeSet, etc.
Access by Index	Provides methods to access elements by an index.	No index-based access.
Performance	Generally faster at accessing elements by index.	Generally faster at searching, adding, and deleting elements.
Use Case	When order and duplicate values are needed.	When a collection of unique elements is needed.

Category	ArrayList	Vector
What is it?	ArrayList is implemented as a resizable array.	Vector is similar with ArrayList, but it is synchronized.
Resizing	ArrayList increments 50% of the current array size if the number of elements exceeds its capacity	Vector increments 100% – essentially doubling the current array size.
Which is faster?	ArrayList is faster, since it is non-synchronized	Vector operations give slower performance since they are synchronized (thread-safe).
Traversal	ArrayList can only use Iterator for traversing.	Vector can use both Enumeration and Iterator for traversing over elements of vector
When to Use?	ArrayList is a better choice if your program is thread-safe.	Choose Vector if you need a thread-safe collection.

ArrayList	LinkedList
1) ArrayList internally uses **dynamic array** to store the elements.	LinkedList internally uses **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3) ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

List	Set	Map
The list interface allows duplicate elements	Set does not allow duplicate elements.	The map does not allow duplicate elements
The list maintains insertion order.	Set do not maintain any insertion order.	The map also does not maintain any insertion order.
We can add any number of null values.	But in set almost only one null value.	The map allows a single null key at most and any number of null values.
List implementation classes are Array List, LinkedList.	Set implementation classes are HashSet, LinkedHashSet, and TreeSet.	Map implementation classes are HashMap, HashTable, TreeMap, ConcurrentHashMap, and LinkedHashMap.
The list provides get() method to get the element at a specified index.	Set does not provide get method to get the elements at a specified index	The map does not provide get method to get the elements at a specified index
If you need to access the elements frequently by using the index then we can use the list	If you want to create a collection of unique elements then we can use set	If you want to store the data in the form of key/value pair then we can use the map.
To traverse the list elements by using ListIterator.	Iterator can be used traverse the set elements	Through keyset, value, and entry set.

Property	Enumeration	Iterator	ListIterator
1) Is it a legacy?	Yes	no	no
2) It is applicable for?	Only legacy classes.	Applicable for any collection object.	Applicable for only list objects.

3) Moment?	Single direction cursor(forward)	Single direction cursor(forward)	Bi-directional.
4) How to get it?	By using elements() method.	By using iterator()method.	By using listIterator() method.
5) Accessibility?	Only read.	Both read and remove.	Read/remove/replace/add.
6) Methods	hasMoreElement() nextElement()	hasNext() next() remove()	9 methods

Difference between HashSet and LinkedHashSet

HashSet	LinkedHashSet
HashSet is based on the HashTable data structure.	LinkedHashSet is based on the combination of HashTable and LinkedList data structure.
In case of HashSet insertion order is **not preserved**.	In case of LinkedHashSet **insertion order is preserved**.

HashMap	Hashtable
HashMap is **non synchronized**. It is **not-thread safe** and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is **thread-safe** and can be shared with many threads
HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**
HashMap is much faster and uses less memory than Hashtable **Unsynchronized objects are often much better in performance in compare to synchronized object**	Hashtable is **slow**.
HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

Comparable	Comparator
1) Comparable provides a **single sorting sequence** with natural ordering. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences** for different attributes. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo(Object a) method** to sort elements. Comparable interface compares "this" reference with the object specified.	Comparator provides **compare(Object o1, Object o2) method** to sort elements. Comparator in Java compares two different class objects provided.
4) Comparable interface belongs to **java.lang** package.	Comparator interface belongs to **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

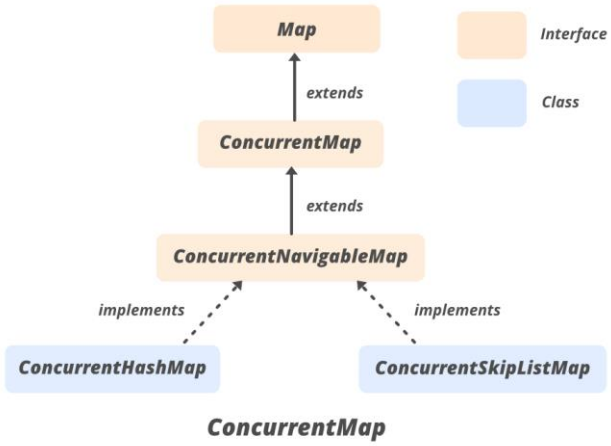
Feature	HashMap	LinkedHashMap
Ordering of Elements	No order guarantees for keys or values	Maintains order based on the insertion of entries
Internal Data Structure	Hash table	Hash table + Linked list (for order)
Performance	Slightly faster for basic operations	Slightly slower due to maintenance of the linked list
Memory Usage	Lower	Higher (due to the extra pointers of the linked list)
Use-cases	When order of elements doesn't matter	When order of insertion or access matters
Iteration Order	Unpredictable	Order of insertion or optionally access order

ConcurrentMap Interface in java:

ConcurrentMap is an interface and it is a member of the Java Collections Framework, which is introduced in JDK 1.5 represents a Map that is capable of handling concurrent access to it without affecting the consistency of entries in a map. ConcurrentMap interface present in **java.util.concurrent** package. It provides some extra methods apart from what it inherits from the SuperInterface i.e. java.util.Map. It has inherited the Nested Interface Map.Entry<K, V>.

HashMap operations are not synchronized, while Hashtable provides synchronization. Though Hashtable is a thread-safe, it is not very efficient. To solve this issue, the Java Collections Framework introduced **ConcurrentMap** in Java 1.5.

The Hierarchy of ConcurrentMap



ConcurrentMap

Declaration:

public interface ConcurrentMap<K,V> extends Map<K,V>

Here, **K** is the type of key Object and **V** is the type of value Object.

- It extends the Map interface in Java.
- ConcurrentNavigableMap<K,V> is the SubInterface.
- ConcurrentMap is implemented by ConcurrentHashMap, **ConcurrentSkipListMap** classes.
- ConcurrentMap is known as a synchronized Map.

Implementing Classes

Since it belongs to **java.util.concurrent** package, we must import is using

import java.util.concurrent.ConcurrentMap

or

import java.util.concurrent.*

The ConcurrentMap has two implementing classes which are **ConcurrentSkipListMap** and **ConcurrentHashMap**. The ConcurrentSkipListMap is a scalable implementation of the ConcurrentNavigableMap interface which extends ConcurrentMap interface. The keys in ConcurrentSkipListMap are sorted by natural order or by using a Comparator at the time of construction of the object. The ConcurrentSkipListMap has the expected time cost of **log(n)** for insertion, deletion, and searching operations. It is a thread-safe class, therefore, all basic operations can be accomplished concurrently.

Syntax:

// ConcurrentMap implementation by ConcurrentHashMap

```
CocurrentMap<K, V> numbers = new ConcurrentHashMap<K, V>();

// ConcurrentMap implementation by ConcurrentSkipListMap
ConcurrentMap< ? , ? > objectName = new ConcurrentSkipListMap< ? , ? >();
```

Basic Methods

1. Add Elements

The put() method of ConcurrentSkipListMap is an in-built function in Java which associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

2. Remove Elements

The remove() method of ConcurrentSkipListMap is an in-built function in Java which removes the mapping for the specified key from this map. The method returns null if there is no mapping for that particular key. After this method is performed the size of the map is reduced.

3. Accessing the Elements

We can access the elements of a ConcurrentSkipListMap using the get() method.

4. Traversing

We can use the Iterator interface to traverse over any structure of the Collection Framework. Since Iterators work with one type of data we use Entry< ? , ? > to resolve the two separate types into a compatible format. Then using the next() method we print the elements of the ConcurrentSkipListMap.

ConcurrentHashMap in Java:

The **ConcurrentHashMap** class is introduced in JDK 1.5 belongs to **java.util.concurrent** package, which implements ConcurrentMap as well as to Serializable interface also. ConcurrentHashMap is an enhancement of HashMap as we know that while dealing with Threads in our application HashMap is not a good choice because performance-wise HashMap is not up to the mark.

ConcurrentHashMap is a thread-safe implementation of the Map interface in Java, which means multiple threads can access it simultaneously without any synchronization issues. It's part of the java.util.concurrent package and was introduced in Java 5 as a scalable alternative to the traditional HashMap class.

One of the key features of the ConcurrentHashMap is that it provides fine-grained locking, meaning that it locks only the portion of the map being modified, rather than the entire map. This makes it highly scalable and efficient for concurrent operations. Additionally, the ConcurrentHashMap provides various methods for atomic operations such as putIfAbsent(), replace(), and remove().

Key points of ConcurrentHashMap:

- The underlined data structure for ConcurrentHashMap is Hashtable.
- ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.
- At a time any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.
- In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.
- The default concurrency-level of ConcurrentHashMap is 16.

- In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as **Segment locking or bucket locking**. Hence at a time, 16 update operations can be performed by threads.
- Inserting null objects is not possible in ConcurrentHashMap as a key or value.

Declaration:

public class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements ConcurrentMap<K,V>, Serializable

Constructors of ConcurrentHashMap

- **Concurrency-Level:** It is the number of threads concurrently updating the map. The implementation performs internal sizing to try to accommodate this many threads.
- **Load-Factor:** It's a threshold, used to control resizing.
- **Initial Capacity:** Accommodation of a certain number of elements initially provided by the implementation. if the capacity of this map is 10. It means that it can store 10 entries.

1. ConcurrentHashMap(): Creates a new, empty map with a default initial capacity (16), load factor (0.75) and concurrencyLevel (16).

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>();

2. ConcurrentHashMap(int initialCapacity): Creates a new, empty map with the specified initial capacity, and with default load factor (0.75) and concurrencyLevel (16).

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(int initialCapacity);

3. ConcurrentHashMap(int initialCapacity, float loadFactor): Creates a new, empty map with the specified initial capacity and load factor and with the default concurrencyLevel (16).

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(int initialCapacity, float loadFactor);

4. ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel): Creates a new, empty map with the specified initial capacity, load factor, and concurrency level.

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(int initialCapacity, float loadFactor, int concurrencyLevel);

5. ConcurrentHashMap(Map m): Creates a new map with the same mappings as the given map.

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(Map m);

ConcurrentHashMap vs Hashtable

HashTable

- **Hashtable** is an implementation of Map data structure
- This is a legacy class in which all methods are synchronized on Hashtable instances using the synchronized keyword.
- Thread-safe as it's method are synchronized

ConcurrentHashMap

- **ConcurrentHashMap** implements Map data structure and also provide thread safety like Hashtable.

- It works by dividing complete hashtable array into segments or portions and allowing parallel access to those segments.
- The locking is at a much finer granularity at a hashmap bucket level.
- Use **ConcurrentHashMap** when you need very high concurrency in your application.
- It is a thread-safe without synchronizing the whole map.
- Reads can happen very fast while the write is done with a lock on segment level or bucket level.
- There is no locking at the object level.
- ConcurrentHashMap doesn't throw a **ConcurrentModificationException** if one thread tries to modify it while another is iterating over it.
- ConcurrentHashMap does not allow NULL values, so the key can not be null in **ConcurrentHashMap**
- ConcurrentHashMap doesn't throw a **ConcurrentModificationException** if one thread tries to modify it, while another is iterating over it.

Properties	Hashtable	ConcurrentHashMap
Creation	Map ht = new Hashtable();	Map chm = new ConcurrentHashMap();
Is Null Key Allowed ?	No	No
Is Null Value Allowed ?	No	No (does not allow either null keys or values)
Is Thread Safe ?	Yes	Yes, Thread safety is ensured by having separate locks for separate buckets, resulting in better performance. Performance is further improved by providing read access concurrently without any blocking.
Performance	Slow due to synchronization overhead.	Faster than Hashtable. ConcurrentHashMap is a better choice when there are **more reads than writes**.
Iterator	Hashtable uses enumerator to iterate the values of Hashtable object. Enumerations returned by the Hashtable keys and elements methods are not fail-fast.	Fail-safe iterator: Iterator provided by the ConcurrentHashMap is fail-safe, which means it will not throw **ConcurrentModificationException**.

Conclusion:

If a thread-safe highly-concurrent implementation is desired, then it is recommended to use **ConcurrentHashMap** in place of **Hashtable**.

Advantages of ConcurrentHashMap:

1. Thread-safe: ConcurrentHashMap is designed to be used by multiple threads simultaneously, making it an ideal choice for applications that need to handle concurrent access to data.
2. Fine-grained locking: Unlike other synchronization mechanisms that lock the entire data structure, ConcurrentHashMap uses fine-grained locking to lock only the portion of the map being modified. This makes it highly scalable and efficient for concurrent operations.

3. Atomic operations: ConcurrentHashMap provides several methods for performing atomic operations, such as putIfAbsent(), replace(), and remove(), which can be useful for implementing complex concurrent algorithms.
4. High performance: Due to its fine-grained locking mechanism, ConcurrentHashMap is able to achieve high performance, even under heavy concurrent access.

Disadvantages of ConcurrentHashMap:

1. Higher memory overhead: The fine-grained locking mechanism used by ConcurrentHashMap requires additional memory overhead compared to other synchronization mechanisms.
2. Complexity: The fine-grained locking mechanism used by ConcurrentHashMap can make the code more complex, especially for developers who are not familiar with concurrent programming.