**Java 8 Features :**

There are a few major Java 8 features mentioned below:

* **Lambda Expressions**: Concise functional code using ->.
* **Functional Interfaces**: Single-method interfaces.
* **Introduced and Improved APIs:**
  1. **Stream API**: Efficient Data Manipulation.
  2. **Date/Time API**: Robust Date and Time Handling.
  3. **Collection API Improvements**: Enhanced Methods for Collections (e.g., removeIf, replaceAll).
  4. **Concurrency API Improvements**: New classes for parallel processing (e.g., CompletableFuture).
* **Optional Class**: Handle null values safely.
* **forEach() Method in Iterable Interface**: Executes an action for each element in a Collection.
* **Default Methods**: Evolve interfaces without breaking compatibility.
* **Static Methods**: Allows adding methods with default implementations to interfaces.
* **Method References**: Refer to methods easily.

**Lambda Expressions**

* Lambda Expression basically expresses an instance of the functional interface, in other words, you can say it provides a clear and concise way to represent a method of the functional interface using an expression. Lambda Expressions are added in Java 8.

**Functional Interfaces**

* An interface that contains only one abstract method is known as a functional interface, but there is no restriction, you can have **n**number of default and static methods inside a functional interface.

**Method Reference**

Method reference is a shorthand notation of a lambda expression to call a method. There are four types of method references that are as follows:

* Static Method Reference
* Instance Method Reference of a particular object.
* Instance Method Reference of an arbitrary object of a particular type.
* Constructor Reference.

**Example:**

numList.stream().filter(n -> n > 5).sorted().forEach(System.out::println);

**Java Stream Programs**

Java Streams provide a powerful and expressive way to process sequences of elements in Java, enabling functional-style operations on collections of data. The Stream API, introduced in Java 8, facilitates efficient data manipulation and transformation using a sequence of operations that can be performed in parallel or sequentially.

**Java 8 Optional Class**

Every Java Programmer is familiar with [NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/). It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

Below are the method which can be used for optional class:



**New Date-Time API in Java 8**

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :

1. Not thread safe : Unlike old java.util.Date which is not thread safe the new date-time API is immutable and doesn’t have setter methods.
2. Less operations : In old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

1. Local : Simplified date-time API with no complexity of timezone handling.
2. Zoned : Specialized date-time API to deal with various timezones.

* LocalDate/LocalTime and LocalDateTime API : Use it when time zones are NOT required.

**What are Immutable objects?**

Immutable objects are objects which once declared elements can’t be modified after it.

**How are String Immutable?**

A String in Java that is specified as immutable, as the content shared storage in a single pool to minimize creating a copy of the same value. String class and all wrapper classes in Java that include Boolean, Character, Byte, Short, Integer, Long, Float, and Double are immutable. A user is free to create immutable classes of their own.

**StringBuffer class in Java**

StringBuffer is a class in Java that represents a mutable sequence of characters. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

Here are some important features and methods of the StringBuffer class:

* StringBuffer objects are mutable, meaning that you can change the contents of the buffer without creating a new object.
* The initial capacity of a StringBuffer can be specified when it is created, or it can be set later with the ensureCapacity() method.
* The append() method is used to add characters, strings, or other objects to the end of the buffer.
* The insert() method is used to insert characters, strings, or other objects at a specified position in the buffer.
* The delete() method is used to remove characters from the buffer.
* The reverse() method is used to reverse the order of the characters in the buffer.
* **Methods of Java StringBuffer class**

| **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. |

**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. However, the StringBuilder class differs from the StringBuffer class on the basis of synchronization. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does. Therefore this class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations. Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used. String Builder is not thread-safe and high in performance compared to String buffer.

**Methods in Java StringBuilder**

[**StringBuilder append(X x)**](https://www.geeksforgeeks.org/stringbuilder-append-method-in-java-with-examples/)**: This method appends the string representation of the X type argument to the sequence.**

1. [**StringBuilder appendCodePoint(int codePoint)**](https://www.geeksforgeeks.org/stringbuilder-appendcodepoint-method-in-java-with-examples/): This method appends the string representation of the codePoint argument to this sequence.
2. [**int capacity()**](https://www.geeksforgeeks.org/stringbuilder-capacity-in-java-with-examples/): This method returns the current capacity.
3. [**char charAt(int index)**](https://www.geeksforgeeks.org/stringbuilder-charat-in-java-with-examples/): This method returns the char value in this sequence at the specified index.
4. **IntStream chars()**: This method returns a stream of int zero-extending the char values from this sequence.
5. [**int codePointAt(int index)**](https://www.geeksforgeeks.org/stringbuilder-codepointat-in-java-with-examples/): This method returns the character (Unicode code point) at the specified index.
6. [**int codePointBefore(int index)**](https://www.geeksforgeeks.org/stringbuilder-codepointbefore-in-java-with-examples/): This method returns the character (Unicode code point) before the specified index.
7. [**int codePointCount(int beginIndex, int endIndex)**](https://www.geeksforgeeks.org/stringbuilder-codepointcount-in-java-with-examples/): This method returns the number of Unicode code points in the specified text range of this sequence.
8. **IntStream codePoints()**: This method returns a stream of code point values from this sequence.
9. [**StringBuilder delete(int start, int end)**](https://www.geeksforgeeks.org/stringbuilder-delete-in-java-with-examples/): This method removes the characters in a substring of this sequence.
10. **StringBuilder deleteCharAt(int index)**: This method removes the char at the specified position in this sequence.
11. [**void ensureCapacity(int minimumCapacity)**](https://www.geeksforgeeks.org/stringbuilder-ensurecapacity-in-java-with-examples/): This method ensures that the capacity is at least equal to the specified minimum.
12. [**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**](https://www.geeksforgeeks.org/stringbuilder-getchars-in-java-with-examples/): This method characters are copied from this sequence into the destination character array dst.
13. [**int indexOf()**](https://www.geeksforgeeks.org/stringbuilder-indexof-method-in-java-with-examples/): This method returns the index within this string of the first occurrence of the specified substring.
14. **StringBuilder insert(int offset, boolean b)**: This method inserts the string representation of the boolean alternate argument into this sequence.
15. **StringBuilder insert()**: This method inserts the string representation of the char argument into this sequence.
16. [**int lastIndexOf()**](https://www.geeksforgeeks.org/stringbuilder-lastindexof-method-in-java-with-examples/): This method returns the index within this string of the last occurrence of the specified substring.
17. [**int length()**](https://www.geeksforgeeks.org/stringbuilder-length-in-java-with-examples/): This method returns the length (character count).
18. **int offsetByCodePoints(int index, int codePointOffset)**: This method returns the index within this sequence that is offset from the given index by codePointOffset code points.
19. [**StringBuilder replace(int start, int end, String str)**](https://www.geeksforgeeks.org/stringbuilder-replace-in-java-with-examples/): This method replaces the characters in a substring of this sequence with characters in the specified String.
20. [**StringBuilder reverse()**](https://www.geeksforgeeks.org/stringbuilder-reverse-in-java-with-examples/): This method causes this character sequence to be replaced by the reverse of the sequence.
21. [**void setCharAt(int index, char ch)**](https://www.geeksforgeeks.org/stringbuilder-setcharat-in-java-with-examples/): In this method, the character at the specified index is set to ch.
22. [**void setLength(int newLength)**](https://www.geeksforgeeks.org/stringbuilder-setlength-in-java-with-examples/): This method sets the length of the character sequence.
23. [**CharSequence subSequence(int start, int end)**](https://www.geeksforgeeks.org/stringbuilder-subsequence-in-java-with-examples/): This method returns a new character sequence that is a subsequence of this sequence.
24. [**String substring()**](https://www.geeksforgeeks.org/stringbuilder-substring-method-in-java-with-examples/): This method returns a new String that contains a subsequence of characters currently contained in this character sequence.
25. [**String toString()**](https://www.geeksforgeeks.org/stringbuilder-tostring-method-in-java-with-examples/): This method returns a string representing the data in this sequence.
26. [**void trimToSize()**](https://www.geeksforgeeks.org/stringbuilder-trimtosize-method-in-java-with-examples/): This method attempts to reduce storage used for the character sequence.

**Types of Access Modifiers in Java**

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

**Default Access Modifier**

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**.

**Private Access Modifier**

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

* Any other **class of**the **same package will not be able to access** these members.
* Top-level classes or interfaces can not be declared as private because
  + private means “only visible within the enclosing class”.
  + protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

**Protected Access Modifier**

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

**Public Access modifier**

The public access modifier is specified using the keyword **public**.

* The public access modifier has the **widest scope** among all other access modifiers.
* Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

**Inheritance in Java**

Java, Inheritance is an important pillar of OOP (Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

**Why Do We Need Java Inheritance?**

* **Code Reusability:**The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
* **Method Overriding:**[Method Overriding](https://www.geeksforgeeks.org/overriding-in-java) is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
* **Abstraction:**The concept of abstract where we do not have to provide all details is achieved through inheritance. [Abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2)only shows the functionality to the user.

**Abstraction in Java**

**Abstraction** in Java is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

In Java, abstraction is achieved by[**interfaces**](https://www.geeksforgeeks.org/interfaces-in-java/)and [**abstract classes**](https://www.geeksforgeeks.org/abstract-classes-in-java/). We can achieve 100% abstraction using interfaces.

**Java Abstract classes and Java Abstract methods**

1. An abstract class is a class that is declared with an [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. **A method-defined abstract must always be redefined in the subclass, thus making**[**overriding**](https://www.geeksforgeeks.org/overriding-in-java/)**compulsory or making the subclass itself abstract.**
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/).
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

**Algorithm to implement abstraction in Java**

1. Determine the classes or interfaces that will be part of the abstraction.
2. Create an abstract class or interface that defines the common behaviors and properties of these classes.
3. Define abstract methods within the abstract class or interface that do not have any implementation details.
4. Implement concrete classes that extend the abstract class or implement the interface.
5. Override the abstract methods in the concrete classes to provide their specific implementations.
6. Use the concrete classes to implement the program logic.

**When to use abstract classes and abstract methods?**

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

**Encapsulation in Java**

Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

**Polymorphism in Java**

The word polymorphism means having many forms. In simple words, we can define Java Polymorphism as the ability of a message to be displayed in more than one form.

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**Compile-Time Polymorphism in Java**

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

***Note:****But Java doesn’t support the Operator Overloading.*

**Method Overloading**

When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be **overloaded** by changes in the number of arguments or/and a change in the type of arguments.

**Subtypes of Compile-time Polymorphism**

**1. Function Overloading**

It is a feature in C++ where multiple functions can have the same name but with different parameter lists. The compiler will decide which function to call based on the number and types of arguments passed to the function.

**2. Operator Overloading**

It is a feature in C++ where the operators such as +, -, \*, etc. can be given additional meanings when applied to user-defined data types.

**3. Template**

it is a powerful feature in C++ that allows us to write generic functions and classes. A template is a blueprint for creating a family of functions or classes.

[**Runtime Polymorphism in Java**](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/)

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. [**Method overriding**](https://www.geeksforgeeks.org/overriding-in-java/), on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

* When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
* At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
* A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

**Interfaces in Java**

**Java Functional Interfaces**

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/) can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***,*and****Comparable*** are some of the examples of functional interfaces.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called *@FunctionalInterface***. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods

**Nested Interface in Java**

We can declare interfaces as members of a class or another interface. Such an interface is called a member interface or nested interface. Interface in a class Interfaces (or classes) can have only public and default access specifiers when declared outside any other class.

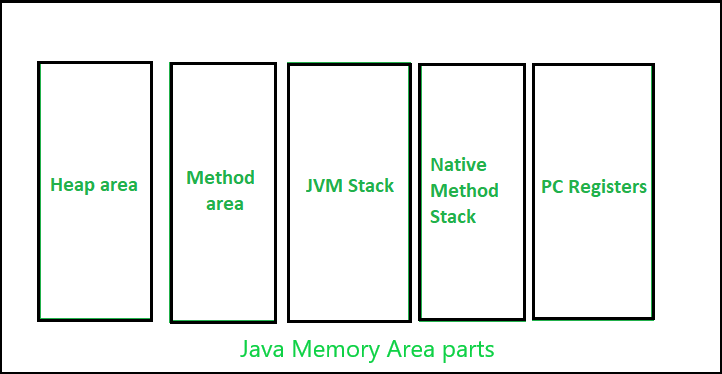
This interface declared in a class can either default, public, protected not private. While implementing the interface, we mention the interface as**c\_name.i\_name** where **c\_name** is the name of the class in which it is nested and **i\_name** is the name of the interface itself.

**Marker interface in Java**

It is an empty interface (no field or methods). Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

**Java Memory Structure:**

JVM defines various run time data area which are used during execution of a program. Some of the areas are created by the JVM whereas some are created by the threads that are used in a program. However, the memory area created by JVM is destroyed only when the JVM exits. The data areas of thread are created during instantiation and destroyed when the thread exits.



**Heap :**

* It is a shared runtime data area and stores the actual object in a memory. It is instantiated during the virtual machine startup.
* This memory is allocated for all class instances and array. Heap can be of fixed or dynamic size depending upon the system’s configuration.

**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.

**JVM Stacks:**

* 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.

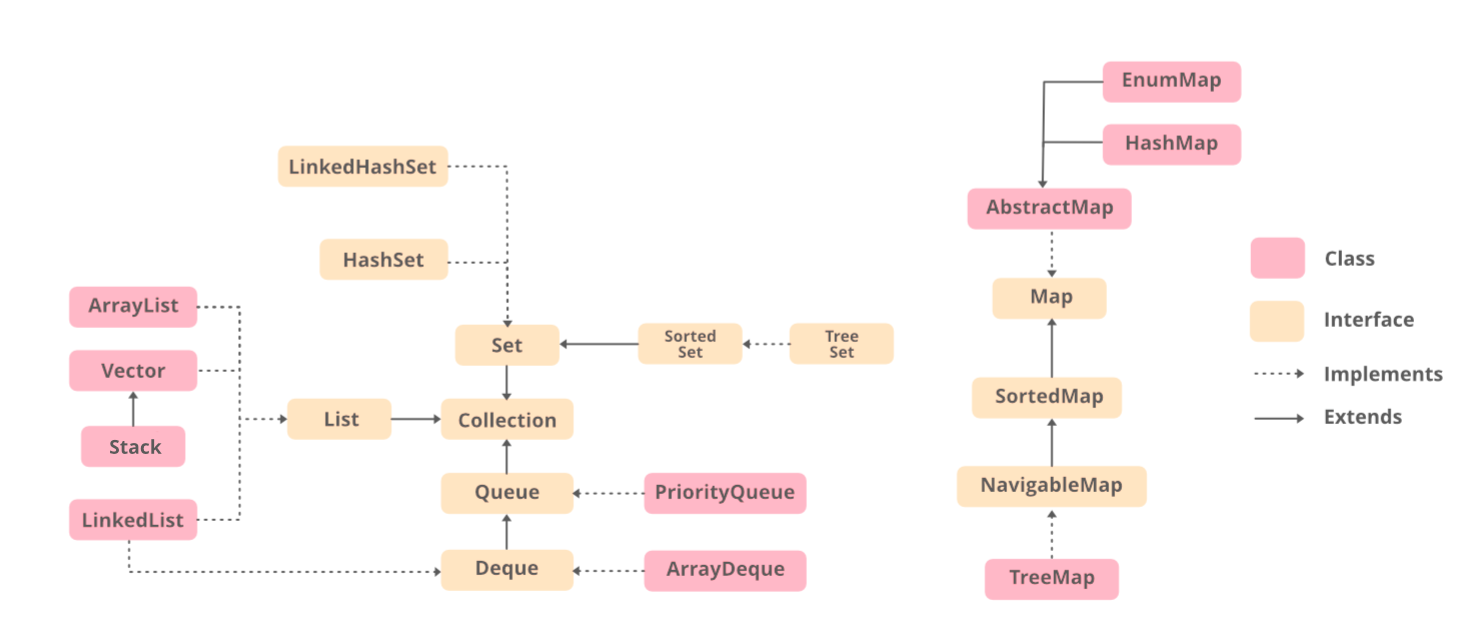
**Native method Stacks:**

* 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.

**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 nonnative method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register can store the return address or a native pointer on some specific platform

**Collection Framework**



[**Iterable interface**](https://www.geeksforgeeks.org/iterable-interface-in-java/)**is** the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections.

The following 6 interfaces are described below first later on been discussed with clean java programs as in implementation.

* Collection interface
* List interface
* Queue interface
* Deque interface (Double-ended queue)
* Set interface
* Map

[**Collection Interface**](https://www.geeksforgeeks.org/collection-interface-in-java-with-examples/) contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation.

[List Interface](https://www.geeksforgeeks.org/list-interface-java-examples/)

It is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collections of the objects. This deals with the index or position-specific functions like getting an element or setting an element. It deals with the arrays and lists types of operations like ArrayList, LinkedList, Vector, and Stack.

1. **[ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/)**

provides us with dynamic arrays in Java. The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collections.

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

cars.stream().sorted().collect(Collections.toList());

* [Add an Element in an ArrayList](https://www.geeksforgeeks.org/java-util-arraylist-add-method-java/)
* [Remove an Element in an ArrayList](https://www.geeksforgeeks.org/remove-element-arraylist-java/)
* [Update an Element in an ArrayList](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/)
* [Remove duplicates from an ArrayList](https://www.geeksforgeeks.org/how-to-remove-duplicates-from-arraylist-in-java/)
* [Reverse an ArrayList](https://www.geeksforgeeks.org/reverse-an-arraylist-in-java/)
* [Converting ArrayList to String](https://www.geeksforgeeks.org/convert-an-arraylist-of-string-to-a-string-array-in-java/)
* [Getting a sub-list from an ArrayList](https://www.geeksforgeeks.org/arraylist-sublist-method-in-java-with-examples/)

For more, go through [Java ArrayList](https://www.geeksforgeeks.org/tag/java-arraylist/)

2. [**LinkedList**](https://www.geeksforgeeks.org/linked-list-in-java/)

is class is an implementation of a doubly-linked list data structure.

LinkedList<String> cars = new LinkedList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

* [Access Elements of a LinkedList](https://www.geeksforgeeks.org/linkedlist-get-method-in-java/)
* [Add Elements to a LinkedList](https://www.geeksforgeeks.org/java-util-linkedlist-add-method-in-java/)
* [Check if the LinkedList contains the Element](https://www.geeksforgeeks.org/linkedlist-contains-method-in-java/)
* [Change Elements of a LinkedList](https://www.geeksforgeeks.org/linkedlist-set-method-in-java/)
* [Remove All Elements from a LinkedList](https://www.geeksforgeeks.org/linkedlist-clear-method-in-java/)
* [Iterate over LinkedList using Iterator](https://www.geeksforgeeks.org/linkedlist-listiterator-method-in-java/)

For more, go through [Java LinkedList](https://www.geeksforgeeks.org/tag/java-linkedlist/)

3. [**Vector**](https://www.geeksforgeeks.org/java-util-vector-class-java/)

provides us with dynamic arrays in Java. This is a legacy class. It is a thread-safe class. This is not recommended being used in a single-threaded environment as it might cause extra overheads. However, to overcome this in Vectors place one can readily use ArrayList.

* [Adding Elements in Vector](https://www.geeksforgeeks.org/vector-add-method-in-java/)
* [Removing Elements From Vector](https://www.geeksforgeeks.org/vector-clear-method-in-java/)
* [Replacing Elements in the Vector](https://www.geeksforgeeks.org/vector-set-method-in-java/)
* [Check if Vector Has Elements or not](https://www.geeksforgeeks.org/vector-isempty-method-in-java/)
* [Iterating the Vector Elements in Reverse Order](https://www.geeksforgeeks.org/how-to-iterate-the-vector-elements-in-the-reverse-order-in-java/)
* [Copying Elements of One Java Vector to Another](https://www.geeksforgeeks.org/copy-elements-of-one-java-vector-to-another-vector-in-java/)
* [Maximum and Minimum Element From Vector](https://www.geeksforgeeks.org/how-to-find-the-minimum-or-maximum-element-from-the-vector-in-java/)

For more, go through [Java Vector](https://www.geeksforgeeks.org/tag/java-vector/)

4. [**Stack**](https://www.geeksforgeeks.org/stack-class-in-java/)

is a class is based on the basic principle of last-in-first-out. This is a legacy class. This inherits from a Vector class. It is also a thread-safe class. This is not recommended being used in a single-threaded environment as it might cause extra overheads. However, to overcome this in Vectors place one can readily use [ArrayDeque](https://www.geeksforgeeks.org/arraydeque-in-java/).

* [Check Whether the Stack is Empty or Not](https://www.geeksforgeeks.org/stack-empty-method-in-java/)
* [Search Element in Stacks](https://www.geeksforgeeks.org/stack-search-method-in-java/)
* [Access Top Element at the Top of Stack](https://www.geeksforgeeks.org/stack-peek-method-in-java/)
* [Add the Element in the Stack](https://www.geeksforgeeks.org/stack-push-method-in-java/)
* [Removing of Element From the Stack](https://www.geeksforgeeks.org/stack-pop-method-in-java/)

For more, go through [Java Stack](https://www.geeksforgeeks.org/tag/java-stack/)

5. **[AbstractList](https://www.geeksforgeeks.org/abstractlist-in-java-with-examples/)**

class in Java provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a Random Access data store (such as an array). For sequential access data (such as a linked list), AbstractSequentialList should be used in preference to this class. It implements the Collection interface and the AbstractCollection class.

6. **[AbstractSequentialList](https://www.geeksforgeeks.org/abstractsequentiallist-in-java-with-examples/)**

class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a “sequential access” data store (such as a linked list). For random access data (such as an array), AbstractList should be used in preference to this class. It is used to implement an unmodifiable list, for which one needs to only extend this AbstractList Class and implement only the get() and the size() methods.

[Set Interface](https://www.geeksforgeeks.org/set-in-java/)

It is an unordered collection of objects in which duplicate values cannot be stored. This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc.

[**Map Interface**](https://www.geeksforgeeks.org/map-interface-java-examples/)

Map is a data structure that supports the key-value pair mapping for the data. It is mainly used in the scenarios where Students are needed to be searched or removed or added according to the roll numbers or in item-frequency scenarios where items are stored along with their frequencies

**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.

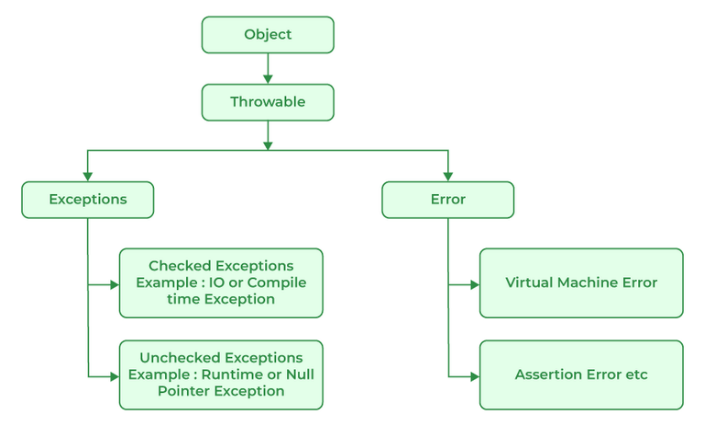
**Major reasons why an exception Occurs**

* 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

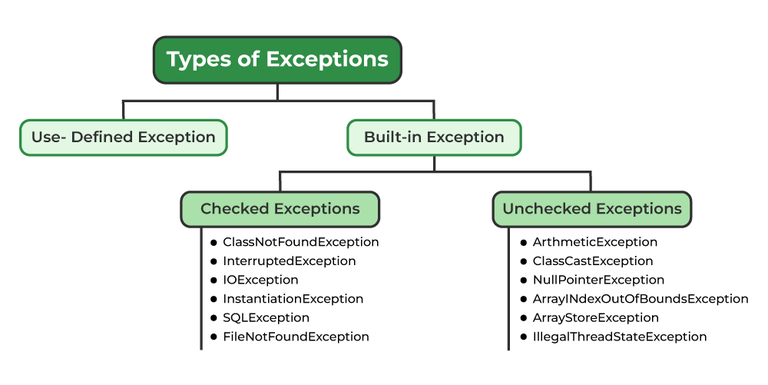
**Exception Hierarchy**

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](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) 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**
   * Checked Exception
   * Unchecked Exception
2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

**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*](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/)

**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:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

**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**.

