Java

**Principles of object-oriented programming -** Four basic OOPs principles:

1. **Encapsulation: (Access Modifier)**

* Hide the implementation details of object.
* Protect the integrity of an object’s data.

Encapsulation is achieved by defining fields in a class as private and providing public getter and setter methods.

2. **Abstraction: (Interfaces)**

* Simplify complex systems by hiding unnecessary details.
* Provide a clear and simple interface.

Abstraction is implemented using interfaces. For example, creating a service layer interface for handling user requests to create new event, show all events etc

Also used Spring data jpa repository interface which abstracts the DAO layer

3. **Inheritance: (Extend keyword)**

* Promote code reuse.

Inheritance is used in the Repository interface to extend the JpaRepository for utilizing common methods. For example, findAll() method implementation is provided by JpaRepository that is used by my EventRepository.

4. **Polymorphism**: (**Method overriding runtime, Method overloading compile time**)

* Allow objects of different classes to be treated as objects of a common superclass.
* Provide flexibility and reusability in code.

Polymorphism is applied through method overriding. For example: EventService.java two overridden methods.

public List<EventDTO> fetchEvents() throws DataAccessException;

public List<EventDTO> fetchEvents(String satelliteName) throws DataAccessException;

**Java Development Kit (JDK**) is a software development environment used for **developing Java applications** and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

**Java Runtime Environment (JRE)** provides the minimum requirements for executing a Java application **(only run (not develop)**). It consists of the *Java Virtual Machine (JVM), core classes*, and *supporting files*

**Java Virtual Machine (JVM)** is responsible for executing the java program line by line. Hence it is known as **Runtime Interpreter,** which is responsible for loading, verifying, and executing the bytecode created in Java.

JVM has specifications and its implementation has been provided by Sun (now acquired by Oracle Corporation), IBM, Redhat, Microsoft, Azul, SAP etc.

JVM is the one that calls the **main method** present in Java code.

The software of JVM is different for different Operating Systems. It plays a vital role in making Java platform Independent.

**Java Platform Independent -** javac compiles the program to form a bytecode or .class file. This .class file is independent of the software or hardware running but needs a JVM (Java Virtual Machine) preinstalled in the operating system for further execution of the bytecode.

**Just-in-Time (JIT) compiler** is a part of JRE (Java Runtime Environment) and is responsible for compiling the bytecode into native machine code at run time.

A diagram of a program

Description automatically generated

**Memory Areas Allocated By the JVM**

1. **Class Area** - It stores class-level data of every class such as static variable, field, runtime constant, method code, and the constructor.
2. **Heap Space** - Objects are created and stored. It allocates memory to objects during run time.

*Till Java 8 static methods and variables were stored in Class Area. But after Java8 static variables and methods are stored in Heap Memory.*

1. **Stack** **Memory** - Contains local primitive variables and reference variables. Each thread has a private JVM stack, created at the time of the thread’s creation.

It stores data and partial results which will be needed while returning value for method and performing dynamic linking.

*Java Stack stores frames and a new frame is created during method invocation. A frame is destroyed when its method invocation completes.*

1. **Program Counter (PC) Register** - stores the address of the Java virtual machine (JVM) instruction currently being executed.
2. **Native Method Stack** - stores all the native methods used in the application.

A diagram of a class loader

Description automatically generated

**Java8 Memory Optimization**

Starting Java8 **PermGen** memory region is replaced by **Metaspace** memory region. **PermGen (Permanent Generation)** is a **special heap space** separated from the **main memory heap space**. The **JVM** keeps track of **loaded class metadata,** stores all the **static content** like **static methods**, **primitive variables**, and **references to static objects in the PermGen**.

Furthermore, **PermGen** contains data about **bytecode**, **names**, and **JIT information**. Before Java7, the **String Pool** was also part of PermGen memory.

The default maximum PermGen memory size for 32-bit JVM is 64MB and for 64-bit JVM is 82MB.y

-XX:PermSize=[size] is the initial or minimum size of the PermGen space.

-XX:MaxPermSize=[size] is the maximum size.

*Due to limited heap memory size,* ***PermGen*** *was responsible for* ***OutOfMemoryError****, as the Class Loaders weren’t properly garbage collected resulting into memory leak.*

**Metaspace** is a new memory space, and **its native memory region grows automatically by default**.

**MetaspaceSize** and **MaxMetaspaceSize –** set the Metaspace upper bounds.

**MinMetaspaceFreeRatio –** is the minimum percentage of **class metadata capacity free after garbage collection.**

**MaxMetaspaceFreeRatio -** is the maximum percentage of **class metadata capacity free after garbage collection** to avoid reduction in amount of space.

**JVM Memory Model**

Java memory model depicts how Java virtual machine (VM) works with the memory (RAM). JVM divides memory between **thread stacks** and the **heap**.

Each thread running in the JVM has its own thread stack. Thread stack contains information about the **methods** a thread has called and all the **local variables** for each method being executed.

A thread can only access it's own thread stack. Local variables created by a thread are invisible to all other threads even if two threads are executing the exact same code. Each thread will create the **local variables of the same code in their own thread stack.**

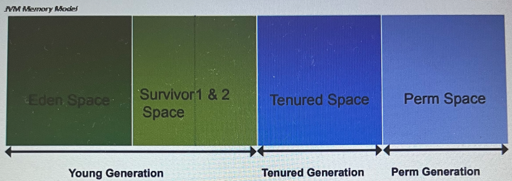
All local variables of primitive types (boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads for sharing.

The reference of the local variables of Object type is stored on the thread stack, but the object itself is stored on the heap. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.).

An object's **member variables** are stored on the **heap** along with the object itself, irrespective of the member variable is of a primitive type or reference to an object.

Static class variables are also stored on the heap along with the class definition.

**Objects on the heap can be accessed by all threads that have a reference to the object.** If a thread has access to an object, it can access its member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but **each thread will have its own copy of the local variables.**



**Young generation (heap)** - consists of one Eden Space and Two survivor spaces. The VM initially assigns all objects to Eden space, and most objects die there. When VM performs a minor GC, it moves any remaining objects from the Eden space to one of the survivor spaces.

**Old/Tenured Generation (heap)**- VM moves objects that live long enough in the survivor spaces to the "tenured" space in the old generation. When the tenured generation fills up, there is a full GC that is often much slower because it involves all live objects.

**Permanent Generation (non-heap)**- The permanent generation holds all the reflective data of the virtual machine itself, such as class and method objects.

**Code Cache (non-heap)** - memory that is used for compilation and storage of native code.

**ClassLoader** is an integral part of the Java Runtime Environment (JRE) that dynamically loads Java classes into the Java Virtual Machine (JVM).

Java classes aren’t loaded into memory all at once, but only when required by an application. The Java Runtime Environment (JRE) calls the Java ClassLoader and then only these ClassLoaders load classes into memory dynamically.

**Types of Class Loaders in Java** each responsible for loading classes from specific locations.

1. **Bootstrap ClassLoader (Primordial Class Loader):** Up to Java8, it loaded core Java files from rt.jar.Java 9 onwards, it loads core Java files from the Java Runtime Image (JRT).
2. **Platform ClassLoader (Extension Class Loader):** It loads platform-specific extensions from the JDK’s module system specified by the system property java.platform or –module-path.
3. **System ClassLoader (Application Class Loader):**It loads classes from the application’s classpath.

**Methods of java.lang.ClassLoader**

1. **loadClass(String name, boolean resolve):** Loads classes referenced by the JVM, resolving them if necessary.
2. **defineClass()**: Defines a byte array as an instance of a class. Throws ClassFormatError if the class is invalid.
3. **findClass(String name):** Finds a specified class without loading it.
4. **findLoadedClass(String name):** Verifies if the class is already loaded.
5. **Class.forName(String name, boolean initialize, ClassLoader loader):** Loads and initializes a class. This method gives the option to choose any one of the ClassLoaders.

If the ClassLoader parameter is null, the Bootstrap ClassLoader is used by default.

Example

Class<?> class = ClassLoader.getSystemClassLoader().loadClass("com.example.MyClass");

**Functionality Principles of Java Class Loader**

1. **Delegation Model:** When the JVM encounters a class, it checks if it’s already loaded in the Class (Method) Area Memory then only the JVM proceeds with execution.

If not, it delegates the loading process through a chain of Class Loaders called the **Delegation Hierarchy Algorithm.**

The Java ClassLoader Sub-System hands over the control to System (Application) ClassLoader which delegates the request to Platform (Extension) ClassLoader which in turn delegates the request to Bootstrap (Primordial) ClassLoader.

The Bootstrap (Primordial) ClassLoader will search in the Bootstrap classpath (JDK/JRE/LIB) example rt.jar or Java Runtime Image (JRT). If the class is available, then it is loaded.

If not, then the request is delegated to Platform (Extension) ClassLoader to search for the class in the Extension Classpath (JDK/JRE/LIB/EXT). If the class is available, then it is loaded.

If not, then the request is delegated to the System (Application) ClassLoader to search for the class in the Application Classpath. If the class is available, then it is loaded.

If not, then a ***java.lang.*ClassNotFoundException** exception is generated.

A diagram of a class loader

Description automatically generated

1. **Visibility Principle:** Classes loaded by parent ClassLoaders are visible to child ClassLoaders, but not vice versa to prevent conflicts between classes loaded by different ClassLoaders.
2. **Uniqueness Property:** ClassLoaders ensure that classes are loaded only once to maintain uniqueness.

**Design Principles and Design Patterns**

**SOLID Design principles –**

1. **Single Responsibility** - A class should only have one responsibility.

***Benefit*** - It will have far fewer test cases, fewer dependencies and will be easier to search for than monolithic ones.

1. **Open/Closed** - classes should be open for extension but closed for modification.

***Benefit*** - We stop ourselves from modifying existing code and causing potential new bugs. By extending the class, and then writing new methods, we can be sure that our existing application won’t be affected. The only exception to the rule is when fixing bugs in existing code.

1. **Liskov Substitution** - If class A is a subtype of class B, we should be able to replace B with A without disrupting the behavior of our program.

***Way to Implement*** - One possible solution would be to rework our model into interfaces that consider the engine-less state of our Car.

1. **Interface Segregation** - larger interfaces should be split into smaller ones.

***Benefit*** - By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

1. **Dependency Inversion** - refers to the decoupling of software modules.

***Way to Implement*** - classes are decoupled and communicate through the abstraction.

**public static void main(String args[]) in Java**

**public** so that JVM can invoke it from outside the class.

**static** so that we can invoke the main method without initiating (initializing) the class, to avoid the unnecessary allocation of the memory due to Object creation.

**void** as the main function doesn’t return anything.

**main** as the JVM looks for the main method name.

**String args[]** stores Java command-line arguments as an array java.lang.String.

*If we don’t declare the main as static that main method will not be treated as the entry point to the application.*

**Wrapper classes** are object classes that encapsulate primitive data types.

1. Wrapper classes are final and immutable.
2. It provides the features of autoboxing and unboxing.

***Autoboxing*** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

For example, converting an int to an Integer, a double to a Double, and so on.

Here is the simplest example of autoboxing:

Character ch = 'a';

***Unboxing*** is converting an object of a wrapper type (Integer) to its corresponding primitive (int) value. The Java compiler applies unboxing when an object of a wrapper class is:

* Passed as a parameter to a method that expects a value of the corresponding primitive type.
* Assigned to a variable of the corresponding primitive type.

**Reader/Writer** class method accepts character arrays as parameters.

**InputStream/OutputStream** class methods accept byte arrays as parameters.

**BufferedInputStream and BufferedOutputStream** classes increase the Input/Output performance of the program by providing the capability of buffering which means that the data will be stored in a buffer before writing to a file or reading it from a stream.

**print** prints all the elements and the cursor remains in the same line.

**println** prints all the elements and shifts the cursor to next line.

**printf** we can use format identifiers to print the elements.

**Operators** are the special types of symbols used for performing operations over variables and values.

**operators in Java are** -

1. **Arithmetic Operators** - mathematical operations. Example **+ - \* / %**
2. **Unary Operators** - only one operand to perform any operation.

Example NOT (!) Increment (++) Decrement (--) Bitwise Complement (~)

1. **Assignment Operator** - assign values to a variable.  Example Simple Assignment Operator (=)

Compound Assignment Operator where +,-,\*, and / is used along with the = operator

a += 10 This means, a = a + 10

1. **Relational Operators** - binary operators used to check for relations between two operands.

Example Equal to (==) Not equal to (!=) Less than(<) Greater than or equal to (>=) Less than or equal to (<=)

1. **Logical Operators** - perform logical “AND” (**&&**), “OR” (**||**) and “NOT” (**!**) operations.
2. **Ternary Operator** - conditional operator that takes three operands.

Example variable = Expression1 ? Expression2: Expression3.

*Exression2* is executed if *Expression1* is true else *Expression3* is executed.

***Time Complexity:****O(1)* ***Auxiliary Space:****O(1)*

1. **Bitwise Operators**
2. **Shift Operators**
3. **instance of operator** - used to test if an object is an instance of a class

**Dot operator in Java** is used to access classes and sub-packages from the package.

***volatile***modifier guarantees that any thread that reads a field will see the most recently written value.

*volatile keywords in Java can only be applied to individual variables but not to arrays or collections.*

| **Sleep()** | **Wait()** |
| --- | --- |
| sleep() method belongs to the **Thread** class. | Wait() method belongs to the **Object** class. |
| Sleep **does not release the lock** that the current thread holds. | wait() **release the lock** which allows other threads to acquire it. |
| This method is a **static** method. | This method is **non static** method. |
| Mainly used to delay a thread for some specific time duration. | Mainly used to pause a thread until notified by another thread. |
| Sleep() Has Two Overloaded Methods:   * sleep(long millis)millis: milliseconds * sleep(long millis, int nanos) nanos: Nanoseconds | Wait() Has Three Overloaded Methods:   * wait() * wait(long timeout) * wait(long timeout, int nanos) |

| **String** | **StringBuffer** |
| --- | --- |
| Store of a sequence of characters. | Provides functionality to work with the strings. |
| It is immutable. | It is mutable (can be modified and other string operations could be performed on them.) |
| No thread operations in a string. | It is thread-safe (two threads can’t call the methods of StringBuffer simultaneously) |

| **StringBuffer** | **StringBuilder** |
| --- | --- |
| It is thread-safe (two threads can’t call the methods of StringBuffer simultaneously) | It is not thread-safe (two threads can call the methods concurrently) |
| Comparatively slow as it is synchronized. | Being non-synchronized, implementation is faster |

*StringBuffer is an alternative to the immutable String class by enabling us to change a string’s contents without constantly creating new objects.*

**String literal** vs **new String(“”)** String literal resides in the String pool. The same address is returned for every reference to a String literal from the String pool.

Whereas on creating String object using new() it allocates a dynamic memory in the heap memory.

*Object gets created in the heap memory even if the same content object is present.*

**StringJoiner** is introduced in Java8 under java.util package is *used to create string by passing delimiters like comma(,), hyphen(-) etc and can also add prefix and suffix to the string*.

final class StringJoiner {//class code}

public StringJoiner(CharSequence delimeter)

public StringJoiner(CharSequence delimeter, CharSequence prefix, CharSequence suffix)

public StringJoiner add(CharSequence element)

public StringJoiner merge(StringJoiner element)

public int length()

**String Pool** in java is a pool of Strings stored in Java Heap Memory. Variable referencing the String is stored in the Stack memory.

JVM checks for the presence of the object in the String pool, If String is available in the pool, the same object reference is shared with the variable, else a new object is created.

A close-up of a computer code

Description automatically generated

**Arrays** in Java are created in heap memory dynamically during the runtime of the program.

The size of the array is specified during the declaration of the array, and it cannot be changed once the array is created.

*The index of an array signifies the distance from the start of the array. So, the first element has 0 distance therefore the starting index is 0.*

int[] Arr = new int[5];

**Jagged Array** in Java is just a 2d array in which each row of the array can have a different length compared to traditional 2-d Array that have the same length of each row.

*This Jagged Array is useful in conditions where the data has varying lengths or when memory usage needs to be optimized.*

A rectangular array with numbers

Description automatically generated

*Advantage of Array* -

1. The size of an array is known at compile time since its elements are stored in contiguous memory regions. Due to which arrays provide quick data retrieval, the Time Complexity to search an element in the Array is O(1).

*Disadvantage of Array* -

1. Arrays are created with a predetermined size so if the array’s size needs to be extended, a new array will need to be made, and the data will need to be copied from the old array to the new array, which can take a lot of time and memory.
2. Unused memory space in an array’s memory space if the array is not completely occupied.

**new operator vs newInstance() method** both are used for object creation. If we know the type of object to be created, then we can use a new operator but if we do not know the type of object to be created in the beginning and is passed at runtime.

Class.newInstance() can only invoke the zero-argument constructor, while Constructor.newInstance() may invoke any constructor, regardless of the number of parameters.

**Class (static) variables** are declared with the static keyword, and they are shared by all instances (objects) of the class as well as by the class itself.

| **Static(Class) method** | **Instance method** |
| --- | --- |
| Static method is associated with a class rather than an object. | The instance method is associated with an object rather than a class. |
| Static methods can be **called using the class name** only without creating an instance of a class. | The instance methods can be called on a specific instance of a class using the object reference. |
| Static methods **do not have access** to **this**keyword**.** | Instance methods have access to **this**keyword**.** |
| **Static methods can access only static members of the class.** | Instance methods can access both static and non-static methods of the class. |
| **Static methods cannot be overridden because they are resolved at compile time, not at run time.** This means that the compiler decides which method to call based on the reference type, not on the object type. | Instance methods can be overridden because they are resolved at run time, not at compile time. This means that the compiler decides which method to call based on the object type, not on the reference type. |

| **Instance Variable** | **Local Variable** |
| --- | --- |
| Declared outside the method, directly invoked by the method. | Declared within the method. |
| Has a default value. | No default value |
| It can be used throughout the class. | The scope is limited to the method. |

**Access Specifiers** in Java help to restrict the scope of a class, constructor, variable, method, or data member. Four types of Access Specifiers in Java: Public, Private, Protected, Default.

**different ways to create objects in Java**

1. Using new keyword
2. Using newInstance() method of Class class.
3. Using clone() method
4. Using deserialization
5. Using the newInstance() method of the Constructor class

**this** is a keyword used to reference a variable that refers to the current object.

**Advantages of passing this into a method instead of the current class object reference**

* this is **final** variable because of which this cannot be assigned to any new value whereas the current class object might not be final and can be changed.
* this can be **used in the synchronized block**.

**Copy constructors** in Java allow the creation of a new object by copying the values of an already present object of the same class.

First, we need to declare a constructor that takes an object of the same type as a parameter. Then, we copy each field of the input object into the new instance.

In the case of primitive types or immutable types Copy constructor will create *shallow copy* but for mutable fields it will make deep copy.

**private constructor** restricts any other class to instantiate the object to avoid subclassing.

**constructors vs methods:**

1. Constructors do not have a return type, whereas methods have a return type.
2. Constructors are only called when the object is created but other methods can be called multiple times during the life of an object.

**Interface** in Java **(Has-A relationship)** is a collection of static final variables and abstract methods. Java8 onwards Interface can have default and static methods as well.

The **static method** of an Interface belongs to the Interface and can only be invoked using the Interface name. static methods in Interface help to provide common utility methods.

Example – doing null checks or sorting logic of collection.

The **default method** in Interface **need not** to be redefined always. It provides backward compatibility that means it does not force existing implementations to be altered unnecessarily.

Example forEach() is a default method in the List Interface whose inclusion does not force the existing implementation like ArrayList to be modified.

*A class can extend only one class but can implement multiple interfaces. Interface allows multiple inheritances in Java.* Java overcomes multiple inheritances diamond problem using below rule:

**Rule1: Classes always win:** A method defined in a class, or its superclass takes precedence over the default method definition in the Interface.

**Rule2: Otherwise, sub-interface wins**: the method with the exact signature in the most specific default-functionality providing interface will be selected.

Rule3: If the choice continues to be ambiguous, **the class that inherits multiple interfaces should explicitly select the default method implementation** to be used just by overriding it and the overridden method should have an explicit call to the desired default behavior.

**Abstract class** in Java **(Is-A relationship)** can have ***both abstract and non-abstract methods***. Declaring a ***class abstract means that it cannot be instantiated on its own and can only be sub classed***. Declaring a ***method abstract means that the method will be defined in the subclass***.

**Need for abstract class** despite Interfaces having default method implementation is that:

1. An **abstract class can override Object class methods, but interface can’t** as interface can extend another interface, but interface cannot extend a class.
2. Interfaces can have variables that are explicitly public, static and final. This means **Interfaces can only declare constants but cannot declare instance variables.** While abstract class cannot be instantiated on its own, an **abstract class can declare instance variables, with all possible access modifiers**, and the instance of the sub-class can access the non-abstract method and the instance variable.
3. An **abstract class can have constructors**, allowing us to initialize the fields on object creation.

**Marker Interface** is an empty interface without any field or methods.

Example - Serializable, Cloneable, and Remote

Classes that do not implement **Serializable** interface will not have any of their state serialized or deserialized.

The serialization runtime associates with each serializable class a version number, called a **serialVersionUID**, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

Example - **private** **static** **final** **long** ***serialVersionUID*** = 14568433L; // Must be static, final of type long

**Functional Interfaces** are Interface with ***only one abstract method*** however they can have any number of default and static method.

**@FunctionalInterface** annotation helps designing an interface as functional interface. This will make the compiler raise an error if the count of abstract methods in the interface exceeds one.

Example – Functional Interface Runnable, Callable and Comparator has one abstract method run(), call() and compare(o1, o2)

**Lambda expressions** provide implementation logic for functional Interfaces and add the essence of functional programming in Java.

*Lambda expressions are functional constructs without classes, which can be passed like objects and executed as required. They make the access modifiers, return type and parameter types optional.*

Syntax **(arguments) -> (body)**

Predefined @FunctionalInterface in the **java.util.function** package**.**

**Predicate<T>** functional Interface has **Boolean test(T t)** method

**Function<T, R>** functional Interface has **R apply(T t)** method

**Consumer<T>** functional Interface has **void accept(T t)** method

**Supplier<T>** functional Interface has **T get()** method

**Streams** denote the flow of a group of elements in sequence from a specific source like collections, I/O resources or arrays.

Data processing operations like filter, map, sort, count etc. can be easily used.

Java8 Stream support has been added by introducing **default stream() method** in **Collection Interface**.

Example – Stream<Employee> empStream = lstEmp.stream();

Whenever a **terminal operation is called on Stream object, the instance gets consumed and closed.** *If we try to consume the Stream object again, we will get IllegalStateException.*

Examples Stream.of("B", "A", "C" , "B").**forEach**(System.out::print);

Stream.of("B", "A", "C" , "B").**collect**(Collectors.toList());

So, to avoid *IllegalStateException* because the Stream object got consumed and closedbetter use Supplier functional interface to get the Stream object and then perform any operation.

Supplier<Stream<String>> streamSup = Stream.of("B", "A", "C" , "B");

Optional<String> list = streamSup.get().**collect**(Collectors.toList());

Stream operations can be pipelined without having to maintain intermediate results using **Intermediate operations** like filter(), sorted() map() etc.

Optional<String> list = streamSup.get().filter(//filter condition).sorted(//sorted condition).collect(Collectors.toList());

**Stream vs Collection** Both are data structures representing a set of values, but **Collection is an in-memory data structure that holds all the data, whereas Streams are data structures whose elements are computed on demand.**

**Parallel Stream** utilizes multithreading. It divides its elements into several chunks and processes each chunk on different thread. By default parallelStream() creates threads whose count equals the number of processors available.

**Constructors** are **called from the top to down hierarchy** in case of Inheritance. Example –

*A is superclass of B. Creating an instance of new B() will invoke B’s constructor, the compiler will in turn call super() and thus constructor of A() will get invoked. So instance of A is created first, and if the constructor of A contains any method which is overridden in subclass B, the subclass method version will be invoked except for static method call.*

*Default no-argument is must in the superclass for the compiler to call otherwise the programmer need to write logic to invoke call to parametrized constructor.*

**Rules for Overriding**

* **The argument list must exactly match** that of the overridden method. If they don't then overloading will be the result instead of overriding.
* The return type must be of covariant type (same or sub-class) - **narrow down return type**
* Overriding method can only throw the same or narrowed checked exception - **narrow down exception**
* **Access level can be less restrictive** - broader visibility of the method
* **Private, Static, and final methods are not inherited and hence can't be overridden** -> No inheritance no overriding
* Free to throw any kind of Runtime exception
* The type of the actual object on the heap decides which method is selected at runtime

**Rules for Overloading**

1. ﻿﻿﻿**Method name must be the same.**
2. ﻿﻿﻿**Argument List must change** in overloaded method
3. ﻿﻿﻿Overloaded method **can change the return type.**
4. ﻿﻿﻿Overloaded method **can change Access modifier.**
5. **New or Broader exceptions can be thrown** by overloaded method
6. Inherited method from the super class (non private) can be overloaded in the subclass.
7. Reference type determines which overloaded version is selected based on argument types at compile time.

**Java Executor Framework**

java.util.concurrent.**Executor** **Interface** released in JDK 5 is used to run the Runnable objects without creating new threads every time and re-using the already created threads from the **ThreadPool**.

*Any excess tasks flowing in, that the threads in the pool can’t handle are held in a* ***Queue****. Once any of the threads get free, they pick up the next task from this queue.*

**Executor Interface <- extends ExecutorService Interface <- implements AbstractExecutorService class**

ThreadPool creation:

ExecutorService executorService = Executors.**newSingleThreadExecutor();**

// thread pool of single thread

ExecutorService executorService = Executors.**newFixedThreadPool(2);**

// thread pool of fixed number of threads.

ExecutorService executorService = Executors.**newCachedThreadPool();**

// If no existing thread is available, a new thread will be created and added to the pool.

ScheduledExecutorService executorService = Executors.**newScheduledThreadPool(1);**

// task that needs to be run at regular intervals or delay a certain task.

scheduleAtFixedRate() and scheduleWithFixedDelay()

The result of the task submitted for execution to an executor can be accessed using the java.util.concurrent.**Future** object

Future<String> result = **executorService.submit(callableTask);**

Callable and Runnable are both designed for classes whose instances are potentially executed by another thread,however **Runnable run() method, does not return a result and cannot throw a checked exception but Callable call() method can return a result and throw a checked exception.**

**Java Design Pattern**

**Creational Design Patterns:** Make the process of object creation more flexible and efficient.

**A diagram of a design

Description automatically generated**