**OOPS Concepts**

**Class:**

* In java, Class is a blueprint or template that defines the properties (fields) and behaviours (methods) common to a group of objects.
* It serves as a model for creating objects with specific attributes and functionalities.
* Class is a blueprint for creating objects in Java.

**Object:**

* An object is an instance of a class that represents a real-world entity. It encapsulates data (fields) and behaviour (methods) defined in its class, allowing manipulation and interaction within a program.
* objects are instances of classes.

**Encapsulation:**

* Encapsulation is the concept of bundling data (fields) and methods (behaviours) within a class, where data is accessed and modified only through methods, ensuring data integrity and providing control over access levels (public, private, protected).
* Encapsulation is the process of bundling data and methods within a class to restrict access and ensure data integrity.

**This keyword:**

* **This** keyword refers to the current object in a method or constructor.

**Constructors:**

* A constructor in Java is a special method that is used to initialize objects.
* The constructor is called when an object of a class is created.
* It can be used to set initial values for object attributes

**Access Modifiers:**

* Controls the access level for classes, attributes, methods and constructors.
* Public – Accessed by other classes. (Class)
* Default – If we don’t provide anything, default is used. It provides access to class in the same package. (Class).

**For Attributes, methods and constructors**

* Public - The code is accessible for all classes
* Private- The code is only accessible within the declared class
* default - The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter
* protected-The code is accessible in the same package and subclasses

**Non-Access Modifiers:**

**Classes**

* Final - The class cannot be inherited by other classes
* Abstract - The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.

**For attributes and methods**

* Final- Attributes and methods cannot be overridden/modified
* Static- Attributes and methods belongs to the class, rather than an object
* Abstract- Can only be used in an abstract class and can only be used on methods. The method does not have a body.
* Transient- Attributes and methods are skipped when serializing the object containing them
* Synchronized- Methods can only be accessed by one thread at a time
* Volatile- The value of an attribute is not cached thread-locally, and is always read from the "main memory"

**Inheritance:**

* In Java, it is possible to inherit attributes and methods from one class to another.
* It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.
* The done using **extends** keyword.
* Superclass – parent class from which we can inherit
* Subclass/child class - the class that inherits from another class.

**Polymorphism:**

* Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.
* With method overloading, multiple methods can have the same name with different parameters.
* The **print ()** method is also an example of polymorphism. It is used to print values of different types like char, int, string, etc.

**Method Overriding:**

* If the same method is present in both the superclass and the subclass. Then, the method in the subclass overrides the same method in the superclass. This is called method overriding

**Method Overloading:**

* we can create methods with the same name if they differ in parameters.

**Java Super:**

* The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).

**Data Abstraction:**

* Data abstraction is the process of hiding certain details and showing only essential information to the user.
* Abstraction can be achieved with either abstract classes or interfaces.

**Abstract Method:**

* A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods.

**THREADS**

**Synchronization:**

* Synchronization in Java is the capability to control the access of multiple threads to any shared resource.
* Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Thread.currentThread().getName() – To get thread information

**Synchronized Method:**

* Locks the instance (object) on which the method is called, allowing only one thread to execute the synchronized method on that instance at a time

**Synchronized Block:**

* Allows for more granular synchronization by locking a specific block of code within a method, typically using synchronized(this) or a shared object to control access to critical sections.

**Static Synchronized:**

* Synchronizes a static method or block at the class level, preventing concurrent execution by multiple threads across all instances of the class by locking the class's monitor.

**wait ():**

* Causes the current thread to wait until another thread calls notify () or notifyAll() on the same object.

**notify ():**

* Wakes up one waiting thread that is waiting on the same object, allowing it to continue execution.

**notifyAll():**

* Wakes up all waiting threads that are waiting on the same object, allowing them to compete for the object's lock and resume execution.

**FILES**

* **createNewFile()** – To create new file
* **Scanner** – To read the file contents
* **getName()** – To get name of the file
* **Write()** – To write inti the file
* **FileWriter** – used along with write to write into the file.
* **Delete()** – to delete the file.

**BINARY FILES**

Binary format refers to representing data using the binary number system, which uses only two digits: 0 and 1. In computer systems, data is typically stored, processed, and transmitted in binary format because it directly corresponds to the on-off states of electronic switches in digital circuits, making it efficient for electronic devices to work with.

* **FileOutputStream:** Used for writing bytes to a file. It's part of Java's IO package (java.io).
* **FileInputStream:** Used for reading bytes from a file. Also part of Java's IO package (java.io).
* **ObjectOutputStream:** Used for writing serialized Java objects to an output stream (e.g., a file). It's used with FileOutputStream to save objects to a file.
* **ObjectInputStream:** Used for reading serialized Java objects from an input stream (e.g., a file). It's used with FileInputStream to read objects from a file.
* **FileOutputStream** writes bytes to a file.
* **FileInputStream** reads bytes from a file.
* **ObjectOutputStream** writes serialized Java objects to a stream. It "serializes" the objects, which means it converts them into a format that can be saved to a file.
* **ObjectInputStream** reads serialized Java objects from a stream

**Text File vs Binary File**

* A text file is a type of file that contains human-readable characters, such as letters, numbers, and symbols.
* Text files store data using character encoding, such as ASCII or Unicode, where each character corresponds to a specific binary value.
* binary file is a type of file that contains data in binary format, represented as sequences of 0s and 1s.
* Binary files store data in its raw binary form, without encoding characters directly. They are used for storing non-textual data, such as images, audio, video, or serialized Java objects.

**serialization and De-serialization**:

* Serialization in Java is the process of converting an object into a byte stream for storing in file or transport over network. while deserialization is the reverse process of reconstructing an object from a byte stream.

**transient**

* The transient keyword in Java is used to indicate that a variable should not be serialized when an object is serialized.

**OBJECT CLASS**

The object class is the parent class of all classes in Java by default. It has 11 methods

**Clone()** –

* To create exact copy of an object. When we create object with new it will take lot of processing time. Using clone, it will be faster.
* It should implement Cloneable interface and throw CloneNotSupportedException.

**toString()**

* Converts object to string.

**hashCode()**

* For every object, JVM generates a unique number which is a hashcode. It returns distinct integers for distinct objects.

**finalize()**

* This method is called just before an object is garbage collected. It is called the Garbage Collector on an object when the garbage collector determines that there are no more references to the object.

getClass()

The getClass() method in Java returns the runtime class of an object, which is an instance of the Class class representing the object's type."

1. **equals(Object obj):** Compares two objects for equality.
2. **hashCode():** Returns a hash code value for the object.
3. **toString():** Returns a string representation of the object.
4. **getClass():** Returns the runtime class of the object.
5. **notify():** Wakes up a single thread waiting on this object's monitor.
6. **notifyAll():** Wakes up all threads waiting on this object's monitor.
7. **wait():** Causes the current thread to wait until another thread notifies the calling object.
8. **wait(long timeout):** Causes the current thread to wait until another thread notifies the calling object or a specified amount of time elapses.
9. **wait(long timeout, int nanos):** Causes the current thread to wait until another thread notifies the calling object or a specified amount of time elapses, with nanosecond precision.
10. **finalize():** Called by the garbage collector before the object is reclaimed to perform clean-up.
11. **clone():** Creates and returns a copy of the object

**Reflection API:**

Reflection is an API that is used to examine or modify the behaviour of methods, classes, and interfaces at runtime.

**String vs String Buffer**

**1.Mutability:**

**String**:

* Immutable: Once created, String objects cannot be changed. Any modification creates a new String object.

**StringBuffer:**

* Mutable: StringBuffer objects can be changed after creation. Modifications are made directly to the existing object.

Example:

**2. Thread Safety:**

**String**:

* Thread-safe: String objects are inherently thread-safe due to their immutability. They can be shared across multiple threads without synchronization.

**StringBuffer:**

* Thread-safe: Methods in StringBuffer are synchronized, making it safe to use in multi-threaded environments. Only one thread can execute a method at a time on a particular instance.

**3. Performance:**

**String:**

* Performance Issues with Modifications: Modifying a String creates new objects, leading to potential performance overhead if many modifications are performed.

**StringBuffer:**

* Better Performance for Modifications: StringBuffer is more efficient for extensive string manipulations as it modifies the existing object without creating new ones.

**4. Usage Scenarios:**

**String**:

* Constant Strings: Use String for fixed sequences of characters that do not change.

**StringBuffer**

* Strings with Frequent Modifications: Use StringBuffer for strings that undergo frequent changes, especially in a multi-threaded environment.

Example:

**5. Memory Usage:**

**String:**

* Higher Memory Usage for Modifications: Since each modification creates a new String object, memory usage can be higher and may lead to more frequent garbage collection.

**StringBuffer:**

* Lower Memory Usage for Modifications: Modifications to a StringBuffer do not create new objects, resulting in lower memory usage and less strain on the garbage collector.

**StringBuffer vs StringBuilder**

**1. Mutability:**

**StringBuffer**:

* Mutable: StringBuffer objects can be changed after creation. Modifications are made directly to the existing object.

**StringBuilder**:

* Mutable: Similar to StringBuffer, StringBuilder objects can also be changed after creation.

2**. Thread Safety:**

**StringBuffer**:

* Thread-safe: Methods in StringBuffer are synchronized, which means they are safe to use in multi-threaded environments. Only one thread can execute a method at a time on a particular instance.

**StringBuilder**:

* Not Thread-safe: Methods in StringBuilder are not synchronized, meaning they are not safe for use in multi-threaded environments without additional synchronization.

3. **Performance**:

**StringBuffer**:

* Slower Performance: Due to synchronization, StringBuffer has an overhead, making it slightly slower than StringBuilder in single-threaded environments.

**StringBuilder**:

* Faster Performance: Without synchronization overhead, StringBuilder is faster and preferred in single-threaded environments.

**4. Usage Scenarios:**

**StringBuffer**:

* Multi-threaded Environment: Use StringBuffer when working with strings in a multi-threaded environment to ensure thread safety.

**StringBuilder**:

* Single-threaded Environment: Use StringBuilder when working with strings in a single-threaded environment for better performance.

**5. Method Synchronization:**

**StringBuffer**:

* Synchronized Methods: Each method in StringBuffer is synchronized, making it thread-safe but with additional overhead.

**StringBuilder**:

* Non-Synchronized Methods: Methods in StringBuilder are not synchronized, which avoids the synchronization overhead.

**StringBuffer and StringBuilder important methods**

1. append(String s): Appends the specified string to this string buffer.
2. insert(int offset, String s): Inserts the specified string at the specified position in this string buffer.
3. replace(int startIndex, int endIndex, String str): Replaces the characters in a substring of this string buffer with characters in the specified string.
4. delete(int startIndex, int endIndex): Deletes the characters in a substring of this string buffer.
5. reverse(): Reverses the sequence of characters in this string buffer.
6. capacity(): Returns the current capacity of the string buffer.
7. ensureCapacity(int minimumCapacity): Ensures that the capacity is at least equal to the specified minimum capacity.
8. charAt(int index): Returns the character at the specified position in this string buffer.
9. length(): Returns the length (number of characters) of this string buffer.
10. substring(int beginIndex): Returns a new string that is a substring of this string buffer, starting from the specified begin index.
11. substring(int beginIndex, int endIndex): Returns a new string that is a substring of this string buffer, starting from the specified begin index and extending to the character at index end index - 1.