OOPs:

object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Object:

An object in Java is a fundamental concept in object-oriented programming (OOP). It is a self-contained entity that encapsulates data (attributes) and behavior (methods).

Objects are instances of classes, which serve as blueprints defining the common characteristics of a particular type of object.

For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake

Key aspects of objects in Java:

- **State:** Represented by attributes or instance variables, which store data specific to the object.
- **Behavior:** Defined by methods, which are functions that can operate on the object's data or interact with other objects.
- **Identity:** Each object has a unique identity, allowing it to be distinguished from other objects, even if they have the same state.
- **Responsibility:** object's responsibilities define what it is expected to do and serve within a system

Creating Objects:

Objects are created using the new keyword followed by the class name and constructor arguments.

ClassName objectName = new ClassName(arguments);

Accessing Object Members:

Object members (attributes and methods) are accessed using the dot operator (.).

objectName.attributeName;
objectName.methodName(arguments);

Class	Object
Class is the blueprint of an object. It is used to create objects.	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
A class is a group of similar objects.	An object is a real-world entity such as a book, car, etc.
Class is a logical entity.	An object is a physical entity.
A class can only be declared once.	Objects can be created many times as per the requirement.
An example of class can be a **car**.	Objects of the class **car** can be BMW, Mercedes, Ferrari, etc.

Class:

A class in Java serves as a blueprint for creating objects. It encapsulates data (fields or attributes) and methods (behaviors or actions) into a single unit. It is a fundamental concept in object-oriented programming

Class is not a real-world entity.

It is just a template or blueprint or prototype from which objects are created.

Class does not occupy memory.

Class is a group of variables of different data types and a group of methods.

A Class in Java can contain:

1. Data member
2. Method
3. Constructor
4. Nested Class
5. Interface

- When an object of a class is created, the class is said to be instantiated.

Abstract classes:

An abstract class in Java serves as a blueprint for other classes. It cannot be instantiated directly, meaning you can't create objects from it. Instead, it's designed to be subclassed (inherited from) by other classes, which then provide concrete implementations for its abstract methods

Key characteristics of abstract classes:

- **Declaration:** Declared using the abstract keyword.
- **Abstract Methods:** Can contain abstract methods, which are methods without a body (no implementation). These methods must be implemented by subclasses.
- **Concrete Methods:** Can also contain concrete methods (methods with a body), which subclasses can inherit or override.
- **Constructors:** Can have constructors, which are called when a subclass is instantiated.
- **Instantiation:** Cannot be instantiated directly using the new keyword.

An abstract method in Java is a method declared without an implementation. It's defined using the abstract keyword and doesn't have a body (no curly braces). Instead, it ends with a semicolon. Abstract methods can only exist within abstract classes or interfaces.

Subclasses of an abstract class containing abstract methods are obligated to provide concrete implementations for those methods. If a subclass fails to implement all abstract methods from its parent class, the subclass itself must also be declared abstract.

Use of Abstract Class in Java

Code Reusability: Abstract classes facilitate code reuse by allowing common methods to be implemented once and inherited by multiple subclasses.

Defining a Common Interface: Abstract classes can define a common interface for a group of related classes, ensuring consistency in their structure and behavior.

Enforcing Method Implementation: Abstract classes can enforce the implementation of certain methods by declaring them as abstract, thereby ensuring that subclasses provide necessary functionality. The abstract class can also be used to provide some implementation of the interface

Interface:

An interface in Java is a blueprint of a class. It is an abstract type that specifies a contract that classes can implement. Interfaces define a set of methods that implementing classes must provide implementations for. They cannot be instantiated directly and can only contain constants, method signatures, default methods, static methods, and nested types.

It is used to achieve abstraction and multiple inheritance in Java.

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance

Compile-Time Polymorphism in Java is also known as static polymorphism. This type of polymorphism is achieved by function 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.

Runtime polymorphism in Java 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 will 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.

If a Parent type reference refers to a Child object . This is called RUN TIME POLYMORPHISM. Parent p=new Child();

A constructor in Java is a special method that is used to initialize objects. It has the same name as the class and does not have a return type, not even void. Constructors are called automatically when an object is created using the new keyword.

Types of Constructors

- **Default Constructor**:

If a class does not explicitly define a constructor, Java provides a default constructor with no arguments. It initializes instance variables with default values (e.g., 0 for int, null for objects).

- **Parameterized Constructor**:

A constructor that accepts arguments, allowing for initialization of instance variables with specific values during object creation.

- **Copy Constructor**:

A constructor that creates a new object as a copy of an existing object. Java does not provide a built-in copy constructor; it must be implemented manually.

Constructor Overloading

A class can have multiple constructors with different parameter lists. This is known as constructor overloading. It allows objects to be created in different ways, depending on the provided arguments.

Use of this Keyword

The “this” keyword refers to the current object instance. It is used to access instance variables, especially when parameter names are the same as instance variable names, and to call other constructors within the same class.

Static:

In Java, the static keyword is a non-access modifier used to define class-level members. It indicates that a variable, method, block, or nested class belongs to the class itself rather than to any specific instance of the class. This means there's only one copy of the static member, shared among all objects of that class.

Types of static members:

- **Static Variables:**

These variables are shared among all instances of a class. They are initialized only once when the class is loaded into memory.

- **Static Methods:**

These methods can be called without creating an instance of the class. They can only access static variables and other static methods within the same class.

- **Static Blocks:**

These blocks of code are executed only once when the class is loaded into memory. They are often used to initialize static variables.

- **Static Classes:**

Nested classes can be declared as static. Static nested classes do not have access to the instance members of the outer class.

Usage and benefits:

- **Memory Management:**

Static members are stored in a common memory location, making memory usage more efficient, especially when dealing with a large number of objects.

- **Global Access:**

Static members can be accessed directly using the class name, without needing to create an object instance.

- **Constants:**

The combination of static and final keywords can be used to define constants that are shared across the entire program.

- **Utility Methods:**

Static methods are useful for creating utility functions that don't depend on the state of any particular object.

Restrictions:

- Static methods can only access static members (variables and methods) directly.
- Static methods cannot use the this keyword, as it refers to the current instance of the class.
- Static methods cannot be overridden in subclasses.
- Non-static methods can access both static and non-static members.

String is a sequence of characters. In Java, objects of the **String class** are immutable which means they cannot be changed once created. In this article, we will learn about the **String class in Java**.

The **StringBuffer class in Java** represents a sequence of characters that can be modified, which means we can change the content of the StringBuffer without creating a new object every time. It represents a mutable sequence of characters.

StringBuffer s = **new** StringBuffer();

In Java, the **StringBuilder class** is a part of the java.lang package that provides a mutable sequence of characters. Unlike String (which is immutable), StringBuilder allows in-place modifications, making it memory-efficient and faster for frequent string operations.

StringBuilder sb = new StringBuilder("Initial String");

Exception:

An **Exception** is an unwanted or unexpected event that occurs during the execution of a program (i.e., at **runtime**) and disrupts the normal flow of the program's instructions. It occurs when something unexpected happens, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.

Exception in Java is an error condition that occurs when something wrong happens during the program execution.

Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
    - Checked Exception
    - Unchecked Exception
2. **User-Defined Exceptions**

1. Built-in Exception

Build-in Exception are pre-defined exception classes provided by Java to handle common errors during program execution.

1.1 Checked Exceptions

Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples of Checked Exception are listed below:

1. **ClassNotFoundException:** Throws when the program tries to load a class at runtime but the class is not found because it's**belong** not present in the correct location or it is missing from the project.
2. **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.
3. **IOException:** Throws when input/output operation fails
4. **InstantiationException:** Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface, or has no default constructor.
5. **SQLException:** Throws when there's an error with the database.
6. **FileNotFoundException:** Thrown when the program tries to open a file that doesn't exist

1.2 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. Examples of Unchecked Exception are listed below:

1. **ArithmeticException:** It is thrown when there's an illegal math operation.
2. **ClassCastException:** It is thrown when you try to cast an object to a class it does not belong**This** to.
3. **NullPointerException:** It is thrown when you try to use a null object (e.g. accessing its methods or fields)
4. **ArrayIndexOutOfBoundsException:** This**This** occurs when we try to access an array element with an invalid index.
5. **ArrayStoreException:** This**handle h**appens when you store an object of the wrong type in an array.
6. **IllegalThreadStateException:** It is thrown when a thread operation is not allowed in its current state

2. User-Defined Exception

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".

. Errors

Errors represent exceptional conditions that are not expected to be caught under normal circumstances. They are typically caused by issues outside the control of the application, such as system failures or resource exhaustion. Errors are not meant to be caught or handled by application code. Examples of errors include:

OutOfMemoryError: It occurs when the Java Virtual Machine (JVM) cannot allocate enough memory for the application.

StackOverflowError: It is thrown when the stack memory is exhausted due to excessive recursion.

NoClassDefFoundError: It indicates that the JVM cannot find the definition of a class that was available at compile-time.

Understanding the different types of exceptions in Java is crucial for writing robust and reliable code. By handling exceptions appropriately, you can improve the resilience of your applications and provide better user experiences.hierarchy of exception handling

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

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 FileWriter class of the **java.io** package is used to write **data in character** form to a file. The FileWriter class in Java is used to write character-oriented data to a file. It is a character-oriented class that is used for file handling in Java.

- This Class inherits from OutputStreamWriter class which in turn inherits from the Writer class.
- The Constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
- FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
    - FileWriter creates the output file if it is not present already.

Serialization in Java refers to the mechanism of converting the state of an object into a byte stream. This allows the object to be easily stored, transmitted, or reconstructed later, which is particularly useful for tasks like saving data to a file, sending objects over a network, or caching. **Deserialization** is the reverse process, converting the byte stream back into an object.

To make a class serializable in Java, it must implement the java.io.Serializable interface. This interface is a marker interface, meaning it doesn't contain any methods. It simply signals to the Java Virtual Machine (JVM) that objects of this class can be serialized.

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)

- Deserialization in Java is the reverse process of serialization. It converts a byte stream back into an object. This process is essential when you need to retrieve objects that have been previously serialized and stored or transmitted. Deserialization is achieved using the ObjectInputStream class.
- To deserialize an object, you first need to have the serialized data, typically in a file or received over a network. Then, you can use the ObjectInputStream to read the byte stream and reconstruct the object