**What is Java?**

Java is object-oriented programming language -> write once, run anywhere philosophy (platform independent)

Java can run on any platform which has JVM installed.

**Why do we say Java object oriented?**

It follows the principles of object-oriented programming (it is a way to model real world objects into software objects which has both data and code).

example: it has classes and objects -> an object here is an instance of the class.

**What is an instance in Java?**

Class it is a blueprint or a template that you have created for one entity. It has properties/state (attributes) and behavior (methods).

* A software object stores its state in fields it is called as variables or attributes.
* And an object exposes its behavior through methods.

A screenshot of a computer program

Description automatically generated A black screen with a white border

Description automatically generated

An object is the instance of the class which has its own set of data stored in the attributes and performs certain actions based on the methods defined.

A screenshot of a computer

Description automatically generated

**What is JDK?**

JDK (Java development kit) has all the tools required for developing a Java application it has a compiler (convert Java source code to Java bytecode), debugger it also has copy of the Java runtime environment (JRE).

* **Java runtime environment (JRE)** - it just has some libraries and classes and a JVM which helps to execute Java bytecode.
* it does not have any development tools like debuggers or compilers.
* **JVM** executes Java bytecode and produces output of the program. It is responsible for converting the platform independent bytecode to a machine code for the specific platform (operating system) it is running on.
* JVM also manages memory allocation and garbage collection to ensure efficient memory usage by Java programs.
* **Memory allocation**: The type of memory managed by JVM is called a heap memory (RAM), This heap memory is dynamically allocated and deallocated as needed during the program's execution.
* The objects that are created with new keywords in Java are allocated onto the heap memory.
* **Garbage collection -** JVM checks if there are any objects (like variables arrays or any other data structures) that are no longer being used in the program and it removes all those objects so that the program can execute without running out of memory.

**What is IDE (integrated development environment)?**

An IDE is just a text editor that facilitates a lot of features for writing an application.

* Code editing i.e., syntax highlighting, code completion and automatic error checking.
* Project management i.e., it also integrates with version control systems like git and allows developers to comment, push or pull changes.
* Debugging, i.e., it includes debugging tools for error identification or like to create like breakpoints or watch variables or like understand step by step execution.
* IDE can also integrate with development tools like version control like Git, testing frameworks like Junit and also tools for importing third party libraries into the projects.

**What are class members**?

In object-oriented programming class members are variables (fields or properties) or methods.

* Variables can be fields or properties which represent the state or data associated with object of the class.
* These variables can be primitive types (int, double, long, Boolean) or reference types (objects, arrays)

A white paper with black text

Description automatically generated

variables can be classified into different types based upon their scope and usage.

**Difference blw primitive and reference data types?**

* Reference data type include (classes, array, interfaces and enumerations). They hold reference to the object in memory unlike primitive data type that hold simple values directly.
* Strings are reference data type. Meaning they hold references of objects rather than values.

**Static variables (representing common data or variable across all the instances of a class)**:

* Static variables and methods belong to a class rather than an individual instance.
* Constants like π, counters that remain constant across all the instances of class are defined with a static keyword.
* Static variables have only one copy that is shared by all instances of the class.
* Static variables exist if the class is loaded into the memory.A screenshot of a computer program

  Description automatically generated

**Instance variables (belong to individual instances of the class, each object may have different values stored):**

* These are associated with the instances of the class and have a different copy for each instance.
* instance variables exist if the instance of the class exists.

A screen shot of a computer

Description automatically generated

**static variables versus instance variables?**

A computer code with green and black text

Description automatically generated A computer code with text

Description automatically generated

**What are access modifiers in Java?**

Java provides 4 main access modifiers, these are essential for encapsulation (protect data from unauthorized access or modification)

* **public:** class, method, variable are accessible from any class (even from different packages).
* **protected:** Accessible from classes in the same package and subclasses (even if they are in different packages).
* **default** (no modifier): Accessible only from classes in the same package.
* **private:** Accessible only within the same class.

**What is the interface in Java? Difference between interface and a class?**

Interface is a blueprint or a contract that other classes must implement. Interface has method signatures without any implementation details, and they are declared using interface keyword in Java (mainly used for polymorphism and abstraction).

A screen shot of a computer code

Description automatically generated

Classes can contain properties (variables), behaviors (methods), constructors, and other members. Classes can implement interfaces by giving concrete implementation to the methods declared in the interface.

A screen shot of a computer code

Description automatically generated

**Explain the concepts of object-oriented programming in Java?**

* Classes and objects
* Inheritance
* Polymorphism
* Encapsulation
* Abstraction
* It's sociation, aggregation, and composition.

**Explain encapsulation (**restricting access to the data (variables) through access modifiers (PPDP) and making them only available through the methods (getters or setters)**)?**

* It means bundling of data (variables) and its behavior (methods) into a single unit known as class.
* it hides the internal state of an object from the outside world and only exposes the necessary functionality through method call (it is a practice of hiding fields and methods from public access).

Encapsulating a class means setting your attributes private and then setting up setters and getters for those attributes to be accessed by other classes. For example if you take a bank account it has attributes like account number, balance, customer name and Phone number, you set all these attributes private and then there are behaviors /methods like withdrawing funds and depositing funds where you set validations like if the account deposit is less than the funds it shouldn't allow you to withdraw so you're restricting the access.

**Why do we need encapsulation?**

* No encapsulation means allowing direct access to the data on object, this bypasses all the checks you placed in a class for validation.
* Any changes to the object’s fields mean the calling code also need to be changed in all places.
* If you do not restrict access to the field members, then instead of constructor’s initialization the data, the calling code is responsible for setting up those data for instances individually.

**What are getters and setters?**

Getters and setters are the methods used to provide access and modify the private fields of a class; this is a fundamental part of encapsulation.

* Getter’s method provides access to a private variable outside the class.
* Setters’ method allows you to modify the private value outside the class.
* We can specify validations and business logic rules through getters and setter’s methods and can access the private variables outside the encapsulated class.

**What is constructor?**

Constructor is a special method in Java which has the same name as its class.

Constructors in Java can be overloaded meaning a class can have several constructors with different parameters it gives the flexibility to initialize an object based on the convenience of a user.

**Difference between getters/setters versus constructor?**

* Constructors are used for initializing objects and setting up their initial state To ensure objects are created in a valid and consistent state.
* Getters and setters provide access to the attributes.

When initializing an object, constructor helps us to set all the required attributes in a single place promoting readability whereas with getters and setters you must make several methods calls which makes the code less readable and harder to understand.

We can use constructors and setters both for having a validation logic. One advantage of having validation logic in constructors is that since constructors use all valid parameter values to create an object, it draws an error saying the object is not created in case of invalid parameter values. However, if you use the validation logic through setters the object will be created. However, you will change the state of that variable after creation.

It is often recommended to use constructors with getters instead of setters to favor more control ways of accessing and modifying an object state. Because in terms of inheritance modifying a setter in superclass can lead to unintended consequences in subclass which can potentially break the functionality of a subclass that rely on superclass interface.

**What is constructor chaining?**

It refers to the process of a constructor calling other constructors in the same class or its super class it allows code reuse and avoid duplication of initialization logic across multiple constructors in the class.

constructor chaining is achieved using the this() keyword to invoke another constructor within the same class or the super() keyword to invoke a constructor in the superclass.

A computer screen shot of a program

Description automatically generated A computer screen shot of a program

Description automatically generated

**static versus instance methods?**

**Static methods**: These belong to the class rather than the instances of the class; these are often stateless and operate only on their parameters or other static variables. You don't need to create an instance to access the static method, you can directly access the static method from its class.

A computer screen shot of a program

Description automatically generated

The main method in Java is a special static method that serves as the entry point for a Java application. It's called by the Java Virtual Machine (JVM) when it starts executing the Java program. The JVM looks for the main method with the signature: **public static void main(String[] args)**

* public: This keyword indicates that the main method is accessible from outside the class. It allows the JVM to call the main method from anywhere.
* static: the main method in Java is declared as static to serve as the entry point of the program, allowing the JVM to execute the program without needing to create an instance of the class containing the main method.
* void: This keyword indicates that the main method does not return any value.
* main: This is the name of the method. The JVM expects to find a method named main to serve as the entry point for the application.
* String [] args (an array of strings): This parameter allows you to pass command-line arguments to the Java program. Users can provide data as arguments while running the program to perform some analysis or a computation (example: java SumCalculator 10 20 30).

Examples of inbuilt static methods are math class, Arrays class and string class and so on

* Static methods are executed first during the initialization of the class, as they are associated with the class itself. Non-static methods are executed within the context of objects created from the class and require the class to be initialized before they can be called. Within the main method, both static and non-static methods can be called, but the order of execution depends on the specific calls made within the main method.
* In Java, you cannot declare a top-level class (i.e., a class that is not an inner class) as static. If you attempt to do so, it will result in a compilation error. Static keyword only implies variables and methods and subclasses within a class, these variables and methods can directly be accessed by class name without the need of Instantiating a class.

**Instance method**: to use the instance method, we first need to instantiate the class using a new keyword. These are accessed using instance references.

* Instant methods can directly access instant variables and other instance methods at the same time they can also access static variable and methods.

**What is a plain old Java object (POJO)?**

* POJO is a class that only has instance fields. It is a straightforward structure without any complex dependencies or framework specific annotations.
* A simple Java class with few instant fields, constructors, getters and setters and a tostring() method for debugging purpose is a POJO
* It represents a simple entity with basic data attribute.

**What are annotations?**

Annotations provide additional information about the code to the compiler, runtime behavior or instructions for tools or frameworks.

**Built-in Annotations:** Java provides several built-in annotations, such as @Override, @Deprecated, and @SuppressWarnings, which are commonly used to convey information about the code to the compiler or other tools.

**@Override:** Itis used to indicate that a method in subclass is overriding a method with the same signature in the superclass. The main purpose is to improve code readability and also serves as a marker for developers’ intent.

It acts as a compile time check, when a method is annotated with ‘@override’ the compiler goes and checks for the method with same signature in the superclass. If it does not find the method with same signature, the compiler gives an error.

Preventing Errors: The @Override annotation helps prevent errors that can occur when a developer mistakenly attempts to override a method but misspells the method name, uses incorrect parameters, or forgets to include the @Override annotation.

It also prevents errors, in cases where developers misspell the method name or give wrong parameter types or forget to include the annotation.

**What is the record in Java?**

Record is a new type of class introduced in Java 14. It is similar to POJO but with less boilerplate code as the compiler generates the constructors, the accesses and the tostring() methods automatically.

* Records are very easy to define. The syntax includes record keywords followed by the record name and the list of components(attributes). Records are by default immutable. The difference between for POJO and record is you cannot use setters’ method to change the attribute value in the record.

public record Point(int x, int y) {} . The compiler automatically generates a constructor that initializes the x and y components, accessor methods (getX() and getY()), equals(), hashCode(), and toString() methods. They mainly reduce the boilerplate code and are useful for representing data centric classes.

* equals(): Used to compare two objects for equality based on their fields or properties.
* hashCode(): Used to generate a hash code value for an object, often based on its fields or properties.
* toString(): Used to obtain a string representation of an object, often including information about its state for debugging or logging purpose.

**What is inheritance? extends keyword (subclass extends superclass)**

Inheritance is used for code reuse. it is a way to organize classes in parent child hierarchy there a child class will inherit all the attributes and methods of a parent class.

* Inheritance allows you to add new functionality by either adding new methods or overriding the existing methods in the subclasses.
* Inheritance also supports encapsulation it helps to hide the internal details of a class and only exposes the necessary information like attributes and behaviors of a public interface.

Key points about the **super () keyword**:

Super keyword is used to access parent class variables or methods.

* If a subclass constructor does not explicitly call super(), the compiler automatically inserts a call to the parameterless constructor of the superclass (super() with no arguments). If the superclass does not have a parameterless constructor or if you want to call a different superclass constructor with arguments, you must explicitly use super() with the appropriate arguments.
* Must Be First Statement: If super() is used in a constructor, it must be the first statement in the constructor body. This is because superclass initialization must occur before any subclass-specific initialization.
* Access to Superclass Constructors: The super() keyword provides access to all constructors of the superclass, allowing you to choose which superclass constructor to call based on the subclass constructor's requirements.

Key points about the **this () keyword**:

This is used to call current class members (variables or methods)

* this() This followed by a parenthesis is used for calling another constructor from the constructor within the same class it enables constructor chaining and reduces code duplication.
* this() must be the first statement in the constructor body to ensure constructor chaining occurs before any other initialization logic in the constructor.
* this() allows constructor overloading with different initialization parameters

A screenshot of a web page

Description automatically generated

* this keyword followed by a parameter is mainly used to distinguish between instance variables and method parameters with the same name.

**Method Overriding(@Override):** when a method in subclass has same method name, return type and parameters as a method in its superclass it is called method overriding.

* The subclass usually overrides the methods that it inherited from superclass to either have a new implementation logic or to extend the functionality of the method it inherited from the superclass.
* When overriding a method in a subclass, the super() keyword can be used to explicitly call the overridden method from the superclass. This is useful when you want to extend the behavior of the superclass method rather than completely replace it.

Method overriding is also known as runtime polymorphism or dynamic method dispatch (JVM decides which method it is going to call at the runtime).

**Method overriding rules:**

* Static methods cannot be overridden, only instance methods can be overridden.
* Constructors and private methods cannot be overridden.
* Methods that are final also cannot be overridden.

**Method overloading**: two or more methods having the same method name but different parameters.

* These methods may have the same or different return type
* May or may not have different access modifiers
* May or may not throw different checked and unchecked exceptions.
* We can overload both static and instance methods.

Method overloading is also known as compile time polymorphism (this means compiler is determining the right method to call based on the name and arguments list)

A table with text on it

Description automatically generated

**Covariant return type**:

When a subclass overrides a method from its superclass it can specify a return type that is a subtype of the return type of the overridden method in the superclass.

* This means that the subclass method can return a more specific (or narrower) type than the superclass method-
* subclass overrides a method from its superclass it can specify the return type (It can be more specific type than the superclass method).

A computer screen shot of a black screen

Description automatically generated A text on a white background

Description automatically generated

**What is Java.lang.object?**

java.lang.Object. is the root class in Java class hierarchy. Every other Java class either directly or indirectly inherits from java.lang.Object.

* If a class does not explicitly extend any other class, it implicitly extends Object. This means that even if a class definition does not include an explicit extends clause, it still inherits the methods defined in Object.

Common Methods: Object defines several common methods that are inherited by all Java classes, including:

* toString(): Returns a string representation of the object.
* finalize(): Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Object provides default implementations for these methods, but they can be overridden in subclasses to provide custom behavior.

**Polymorphism**:

polymorphism is the ability to execute different behavior for different types which can be determined only at the runtime. It allows different implementations of a method to be invoked based on the context in which they are used.

Example :

A screenshot of a computer program

Description automatically generated

public static void main(String[] args) {

Shape circle = new Circle(5.0);

Shape rectangle = new Rectangle(4.0, 6.0);

System.out.println("Area of circle: " + circle.calculateArea());

System.out.println("Area of rectangle: " + rectangle.calculateArea());

}

}

In this example, circle and rectangle are both declared as type Shape, but they point to instances of Circle and Rectangle respectively. When calculateArea() is called on each object, Java invokes the overridden method corresponding to its actual class, demonstrating polymorphic behavior.

Polymorphism simply means many forms; it enables code extensibility and reusability.

**Compile time polymorphism/method overloading:** When compiler interprets source code to Java bytecode, It determines which method to call based on number

and type of arguments provided during method call.

**Runtime polymorphism/ method overriding**: The bytecode is being executed by Java virtual machine it determines the appropriate overridden method to call.

Compiler looks for static methods while starting a program (hence main method is static) and they can be resolved at the compile time itself. Since we cannot override a static method, we perform method overriding on instance methods only.

* Static methods are associated with the class itself not with the individual objects or their runtime types hence they cannot be overridden in the same way as instance methods.
* When a method is invoked on an object reference, the JVM will look for actual runtime type of that object to determine which method to implement during the method call and since matter over overriding happens only on instance methods and that it is runtime polymorphism.

**What is an exception?**

Exception is an error occurred in the code.

* In Java we catch exceptions using try catch blocks, the line of code which throws an error can be written under try block and we can catch those exceptions under the catch block by defining the type of exceptions that can occur in catch. We can also use finally block in cases where we want some piece of code to run/execute no matter what. If you have a return statement in both try and finally blocks, the ultimate return statement that will be executed will be from the finally block.

**Exception hierarchy:**

**A diagram of an error

Description automatically generated**

**Checked versus unchecked exceptions**:

* Checked exceptions are exceptions that are checked at compile time by the compiler.
* They represent exceptional conditions that a well-behaved program should anticipate and handle.
* Checked exceptions must be either caught (handled) using a try-catch block or declared in the method signature using the throws keyword.

Examples of checked exceptions include IOException, FileNotFoundException, SQLException, etc.

* Unchecked exceptions are also known as runtime exceptions. These are usually caused by programming or logical errors in the code.

Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, etc.

**Text Block (“””):** Text blocks are introduced in Java 13 (preview feature) to write multiline strings in Java with out use of several escape characters.

* At compile time, text blocks are considered as string literals and escape sequences are also processed.
* At runtime, text blocks are considered as instances of String class.
* Text blocks are quoted with “”” … “”” and can span multiple lines.

**Escape Sequences:** Below are commonly used escape sequences –

A white paper with black text

Description automatically generated

**Format Specifiers**: %[argument\_index$][flags][width][.precision]conversion.

URL - <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Formatter.html>

**String Methods**:

String Inspection Methods”

A screenshot of a computer

Description automatically generated

String Comparison Methods : contentEquals() is used to compare content of string with stringBuffer or stringBuilder.

A screenshot of a computer

Description automatically generated A screenshot of a phone

Description automatically generated

String Manipulation Methods

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

**String vs stringBuilder:**

* Once a string object is created it cannot be modified, any operations performed on the object will create a new string object. Strings are thread safe and can be used in multithreaded environment without synchronization concerns as they are immutable.
* String builders are mutable and any operations on stringBuilder will modify the existing object instead of creating a new object.

**StringBuilder vs StringBuffers**: Both can create mutable sequence of characters, and provide methods like append(), insert(), delete().

* StringBuilder are introduced in Java 5 as a part of java.lang package, they are immutable and are used in single thread environments (not synchronous )

A diagram of a computer

Description automatically generated

* StringBuffer are synchronous and can be used in multithread environments, however due to synchronization overhead they have slightly lower performance than the stringBuilder().

**Composition vs inheritance**:

* Inheritance defines **IS A** relationship (uses extend keyword).
* Composition defines **HAS A** relationship.
* composition is creating classes that contain instance of other classes as its member variables. (Instead of inheriting attributes and methods from a parent class/superclass, Composition involves creating complex objects with simple objects

**Why composition?**

It also avoids fragile base class problems (where changes in superclass can affect subclasses inadvertently). Inheritances can lead to complex class hierarchies, where changes in one class may affect its subclasses and their functionalities.

**Disadvantages of composition?**

* Composition can lead to increased complexity in class relationships in the case of multiple levels of composition. And any changes to structure of composed classes may lead to modification of multiple classes.

**What is a factory method**?

* A factory method is used to create the instances of objects without needing to know what's the concrete class of those objects is
* Instead of directly instantiating the objects we create a factory method which already created those instances of the objects for us to use through a method.

A black screen with white text

Description automatically generatedA screen shot of a computer program

Description automatically generated

In the given example Movie is a super class, adventure, sci-fi, comedy all are subclasses you can access the instances of these subclasses through getMovie() method instead of us initializing it in the main class.

**Var Keyword and LVTI (local variable type inference):**

* The var keyword and local variable type inference are the features introduced in Java 10.

These allow the developers to declare variables without explicitly specifying their type. Here the compiler infers the type based on the expression used to initialize it on the right side of the variable declaration.

A black screen with white text

Description automatically generated

Note: only non-primitive types can be inferred primitive types cannot be inferred

* LVTI is the underlying feature that enables the use of var keyword it only applies to local variables(variables declared within a method or a block) and does not affect method signatures or class fields.

**Compile time versus runtime typing**:

* Determining the type of variables expressions and values during the compilation phase of the program is called compile time typing whereas if we determine the type of objects variables and expressions at runtime in the program is executing it is called runtime typing.
* In LVTI, we do not declare the type for the compiled reference type (it can be variable reference, method return type) it get inferred as if we had already declared it.
* In many cases the compile time type is declared on the left side of the assignment operator. However in certain cases we can only determine the return type of an expression or a variable or a method at the runtime only, that is runtime typing
* We can know runtime type of an object/instance with. getClass() or instanceOf from an object class

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**Pattern Matching for InstanceOf operator** - If the JVM can identify that the object matches the type, it can directly extract data of object without casting. It became part of Java 16 features. 

The object sci-fi is already typed as science fiction we do not need to cast it again.

**Enumerations (Keyword enum)**: it is a special type in Java that has list of constants which define fixed set of related values.

E.g.: DaysOfWeek, enum PaymentMenthod {Credit card, Debit card, Paypal}.

Note: enumerations can only hold a value that was declared in the list, other values are not allowed. We can say that enumeration gives us the way to explicitly specify the only values that are data type can legally have and are usually used to define a set of values that represent a specific collection.

We can define methods, instance variables and constructors in our enumerations. The constant created in the enumerator are the objects of enumeration type, so if we create a constructor in the enumerator, the constructor is called when each enumeration constant is created.so each enumeration constant has its own copy of instance variable defined by the enumeration,

* Enum has a list of constants which are implicitly defined as public static final. The type of these constants is same as the type of enumeration they are defined, i.e., these constants are self-typed.

A screenshot of a computer program

Description automatically generated

* Here we have defined an enum payment method, with the three constants (credit card, debit card, PayPal).
* Each constant is implicitly declared as public static final member of PaymentMethod.
* The type of the constants is the type of enumeration in which they are declared which is PaymentMethod in this case.
* Even though enumerations are a class type we do not instantiate an enum with **new** keyword, because enums are immutable and have predefined set of constant instances defined within the enum declaration itself.
* We define them same as primitive types (example PaymentMethod paymentMethod)
* Enums work well with switch statements compared to integer and string constants.

values() vs valuesOf()

* The values () method returns an array containing all the constants declared in the enum declaration.
* The valueOf() method returns the enum constant with specific name.

All enumerations automatically inherit one superclass java,lang.enum.

* It also inherits several methods that are available for use by all enumerations.
* ordinal( ) method - it returns the position in its enum declaration where initial constant is assigned as 0
* compareTo( ) - based on the ordinal value it compares 1 enum constant with another
* equals( ) - equals( ) method can compare enumeration constant with any other object however the method will return only true if the two objects contains the same value and they are from the same enumeration. Simply having ordinal values in common will not cause equals( ) to return true if the two constants are from different enumerations.
* A white paper with black text

  Description automatically generated A white screen with black text

  Description automatically generated

**Wrapper Classes:** these are the classes in Java that encapsulate primitive data types within an object. Since primitive data types (int, double, float, char) are not objects but it is necessary to treat them as objects in certain scenarios. We can also say wrapper class provides a way to use primitive data types as reference data types (contains useful methods).

A screenshot of a computer

Description automatically generated

* Working with Collections: Java collections (like ArrayList, LinkedList, etc.) can only store objects, not primitives. Wrapper classes allow you to store primitive values in collections.
* Utilizing Generic Types: Generic types in Java cannot accept primitive types as type arguments. Wrapper classes can be used as type arguments in generics.
* In order to obtain the wrapper object we need to use one of the valueOf() methods.

e.g. Integer a = Integer.valueOf(100) . After the statement executes the value 100 is represented by an integer instance, thus ‘a’ wraps the value 100 within an object.

* Byte, Short, Integer, Long, Float, and Double, these are the wrapper classes that represent numeric values, and they inherit the abstract class Number.

**Autoboxing & auto unboxing**:

* Autoboxing is automatic conversion of a primitive type to its wrapper class whenever an object of wrapper class is needed. With the auto boxing you don't need to manually construct an object to wrap a primitive type with auto boxing you don't need to manually construct an object to wrap around a primitive type you just have to a sign the value to type wrapper reference Java automatically creates an object for you.
*  (This is automatic auto boxing.) Here instance of an integer is created and its value is set to 100.
* However, there are static factory methods which can also be used for autoboxing -  **valueOf(int)**
* similarly, auto unboxing is a process of converting wrapper class to its corresponding primitive type whenever a primitive value is required.
* 
* A close-up of a black and white text

  Description automatically generatedA close up of a sign

  Description automatically generated

**Why can't we use wrapper classes instead of primitive types?**

Because each autobox and auto-unbox add overhead that is not present if the primitive type is used.

Autoboxing/ auto unboxing are not added in Java as a backdoor way of eliminating the primitive types. We only need to use wrapper classes in cases where object representation of a primitive type is required.

**Advantages of using wrapper classes**

Nullable Values: Wrapper classes allow you to represent nullable values by assigning null. For example, if a patient's weight or height is not available, you can set the corresponding attribute to null, which may be useful in certain scenarios.

Integration with Collections: If you need to store patient records in a collection like ArrayList or HashMap, you must use wrapper classes because these collections can only store objects, not primitives.

**Disadvantages of wrapper classes -** memory overhead performance overhead boxing and unboxing overhead nullity complexity potential for unintended consequences

Memory Overhead: Wrapper classes consume more memory than their corresponding primitive types. Each instance of a wrapper class has additional overhead due to the object header, reference to the object, and other metadata associated with objects in memory.

Performance Overhead: Because wrapper classes are objects, working with them can introduce performance overhead compared to primitives. Operations such as autoboxing, auto-unboxing, and method calls on wrapper objects may incur additional overhead compared to working directly with primitives.

Nullability Complexity: Wrapper classes can be null, which adds complexity when handling null values. You need to handle null checks to avoid NullPointerExceptions.

Potential for Unintended Consequences: Because wrapper classes are objects, they behave differently from primitives in certain contexts. For example, comparing two wrapper objects with == checks for reference equality, not value equality. This can lead to unexpected behavior if not handled carefully.

**Implicit Type Casting (Widening Conversion):**

* Use implicit type casting when converting from smaller to larger data types to ensure data integrity and precision.
* int intValue = 10; long longValue = intValue; // Implicit casting from int to long

**Explicit type casting, also known as narrowing conversion,**

* Use explicit type casting when converting from larger to smaller data types, but exercise caution to avoid data loss or unexpected behavior.
* double doubleValue = 10.5; float floatValue = (float) doubleValue; // Explicit casting from double to float

**Java I/O Package:**

* Java performs input output operations through streams and stream is an abstraction that either produces or consumes information.
* Input stream: a stream can extract information from any different form of sources keyboard, disk file or a network socket.
* Output stream: this refers to a disk file console or a network connection.

Streams are categorized into two types

* Byte streams (I/O of bytes, reading and writing binary data)

Character streams(input and output of characters)

In cases, if we want to work on files that holds characters we use character stream, where we do not what kind of data that file contains (data from database, image, audio and serialized objects (process of converting an object from its memory representation into a format that can be easily stored transmitted or reconstructed later -> saving and restoring the state of objects in a persistent storage like files and databases )) then we use Byte Streams.

Internally character streams also use byte streams, it is much simpler and easier to use character streams if we know the file has only characters. Also the character streams read char by character.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

**Byte Stream:** the two important abstract classes are **InputStream** and **OutputStream** which implement **read( )** and **write( )** methods to read and write bytes of data. Each method has a form that is abstract and must be overridden by derived stream classes.

Byte streams are used to perform I/O of 8-bit bytes, which can represent binary data. The classes for byte streams are typically suffixed with InputStream or OutputStream. Examples include FileInputStream, FileOutputStream, ByteArrayInputStream, and ByteArrayOutputStream

**Character stream:** it has two abstract classes named **Reader** and **Writer**.

Character streams handle I/O of 16-bit Unicode characters, making them suitable for text-based data. They are typically suffixed with Reader or Writer. Examples include FileReader, FileWriter, BufferedReader, and BufferedWriter.

FileReader and FileWriter: They are straightforward to use when you need to read from or write to a file character by character or string by string. However, for improved performance, it's often recommended to wrap them with BufferedReader and BufferedWriter (They are typically used for reading and writing text files line by line or in chunks. They are especially useful when dealing with large files or when efficiency is important.).

**Note:** all Java programs automatically import java.lang. package. This package defines a class called system. System contains 3 predefined stream variables in, out, and err these are declared as public static and final within the system i.e., they can be used in any other part of Java program without reference to a specific system object.

**System.out** : this is a standard output stream

**System.in:** this is a standard input stream(keyboard by default)

**System.err:** it refers to the standard error stream maybe redirect it to any compatible IO device.

* **System.out and System.err** are objects of **PrintStream**
* **System.in** is an object of **InputStream**

**Reading console input:** we can read console input by reading from System.in

-One way to obtain a character-based stream that is attached to the console is to wrap **System.in** in a **BufferedReader**. The BufferedReader class supports a buffered input stream

BufferedReader(Reader *inputReader*)

* Reading characters: To read a character from a BufferedReader, use read( ). The version of read( ) that we will be using is
* int read( ) throws IOException
* Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1, when an attempt is made to read at the end of the stream. As you can see, it can throw an IOException.

In the below code, InputStreamReader acts as a bridge between byte-oriented streams (such as InputStream) and character-oriented streams (such as Reader). It converts bytes read from the input stream into characters that can be processed by character-oriented classes like BufferedReader.

BufferedReader adds buffering functionality to the underlying reader. It reads data from the input stream in chunks and stores it in an internal buffer. This reduces the number of I/O operations and improves performance, especially when reading large amounts of data.

System.in is line buffered, by default. This means that no input is actually passed until you press enter, so here instead of read() we can use readLine() method, which is also a member of buffered reader class, which reads and displays lines of text until you specify certain conditions for the program to stop in the console

A screen shot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generated

**PrintWriter:** in cases where we wanted to write formatted text to a character based output streams we use print writer example for files sockets or any other character based streams. In the below example we are writing hello print writer to a new file called output.txt. In another example via printing the output to the console, the second parameter true enables auto flush. If flushingOn is true, flushing automatically takes place. If false, flushing is not automatic.

A computer screen with text

Description automatically generatedA black screen with colorful text

Description automatically generated

**Transient Modifiers:** in cases where an instance variable is specified as transient it will be excluded from Java serialization mechanism when an object is serialized to be stored to the output streams like files and databases (serialization is the process of converting an object from its in memory representation into a format where it can be easily stored transmitted and reconstructed later)

static variables are not part of the object's state and are not serialized.

* It does not prevent the instance fields from being allocated memory or initialized when the object is created.
* When an object is serialized (converted into a byte stream for storage or transmission), only the values of non-transient fields are saved.

**Serialization:** Serialization is the process of converting an object from its in-memory representation into a format that can be easily stored, transmitted, or reconstructed later. In Java, serialization is achieved using the **java.io.Serializable interface**, which allows objects to be converted into a stream of bytes and then reconstructed back into an object

* Saving and restoring the state of objects in a persistent storage (e.g., files, databases).
* Transmitting objects between different applications or across a network.
* Caching and sharing objects in distributed systems
* A black text on a white background

  Description automatically generated A screenshot of a computer

  Description automatically generated

**Why do we use transient modifier in healthcare applications**

* Certain fields within a patient record might contain sensitive information, such as social security numbers, financial data, or personal identification numbers. To protect this sensitive data from being serialized and stored persistently, you could mark these fields as transient. This ensures that they are not included when the patient record objects are serialized and stored in the database.
* Some fields within a patient record might be derived from other fields or calculated dynamically. Examples include fields representing computed values like age, BMI (Body Mass Index), or risk scores. Since these fields can be recalculated or derived from other fields when needed, there may be no need to serialize them.
* By marking these fields as transient, you ensure that they are excluded from the serialization process, saving storage space and avoiding potential inconsistencies between the serialized data and the recalculated values.

**Volatile modifiers:** volatile is often used for flags or variables that are shared among multiple threads and whose values may change asynchronously.

* In multithreaded programming sometimes two or more threads share the same variable value, the volatile modifier is used to indicate that a variable's value may be modified by multiple threads that are executing concurrently.
* Volatile modifier specifies the compiler that the variables value may be modified by some other parts of the program,so it tells the compiler that it always must use the master copy of a volatile variable in order to avoid inconsistencies.

**Why do we use volatile modifier in healthcare applications**

* In multi-threaded healthcare applications, different threads may access and modify shared data concurrently. For example, in a real-time monitoring system, one thread might be responsible for updating patient vital signs while another thread is processing alarms or notifications.
* By marking shared variables as volatile, you ensure that changes made by one thread are immediately visible to other threads. This helps prevent data inconsistencies and ensures that all threads see the most up-to-date values of shared variables.
* For example, in a healthcare application where multiple threads are updating a shared counter representing the number of patients admitted to a hospital, marking the counter variable as volatile ensures that all threads see the latest count without the need for explicit synchronization.

**instanceOf:** This operator in Java is used to know the type of an object. The syntax of the instanceof operator is: **object instanceof Class.**

Here, object is the reference variable to be tested, and Class is the class or interface being tested against.

* An invalid casting can cause runtime errors we can identify the cast of an object using instance of and prevent the program from running into errors.

A white paper with black text

Description automatically generated

* It's also used in type casting to ensure type safety before performing the casting operation.

**Strictfp:**

The strictfp keyword in Java is a modifier that can be applied to classes, interfaces, and methods. It stands for "strict floating-point" and affects the floating-point calculations performed within the scope of the entity to which it is applied.

A screen shot of a computer program

Description automatically generated

In this example, the MathOperations class and the calculate method are declared with strictfp, ensuring that all floating-point calculations within the class/ method adhere to strict floating-point rules.

* Java program can run on any platform with JVM installed (platform independent). However different platforms may have different implementations of the JVM which could lead to variations in floating point arithmetic behavior.
* Strictfp specifies formats for representing floating-point numbers, rules for rounding, handling of special values (such as NaN and infinity), and requirements for arithmetic operations.

**Package:**

Packages lets us organize our classes by functionality or relationships.

We can define a package by just including **package** command as the first statement in Java source file. If we omit the package keyword, all the classes. Will be put in The default package, which has no name.

* package mypackage; (here mypackage is the name of package)
* package pkg1[.pkg2[.pkg3]] – e.g., package a.b.c; We just have to specify a period Between each package name in order To create a hierarchy.

**Packages and Member Access**:

* Packages act as container for classes and other packages. Whereas classes act as container for data and code. The class is Java smallest unit of abstraction.
* Java addresses 4 categories of visibility for class members.
  + Subclass in the same package.
  + Subclass in the different package.
  + Non subclasses in the different package.
  + Classes that are neither in the same package nor subclasses.

A table with text on it

Description automatically generated

Java has four access modifiers and they are public, private, protected and no modifier.

* As a thumb rule, anything declared as public can be accessed from different classes and different packages.
* Whereas anything declared as private cannot be accessed outside its class.
* In cases where no access specifier is explicitly mentioned, these can be accessed by Other classes and subclasses in the same package, but not from a different package. This is no modifier.
* If you want your elements to be seen outside your current package, but only to those classes that subclass your class directly, then we need to use protected access modifier.

**Arrays:** It is a data structure that allows you to store sequence of values all of same type.

We can instantiate a new array using new Keyword. We have array declaration on the left side of = array creation expression on the right side.

A white rectangular object with black text

Description automatically generated

* You cannot change the size of an array once the array is instantiated.
* We can't add or delete elements. We can only assign values to one of the elements assigned in the array.
* In the example we use new keyword to create an array. And specified the length of the array as 10.

A close-up of a number

Description automatically generated

**Array initializer as an anonymous array:**

* Above is an example of anonymous array initializer which can be only used in declaration statement. Here the size of array is determined by the number of elements specified in {}
* We cannot declare an array in one line And initialize the array elements in another line without specifying the size of an array. However, we can Initialize an array as an anonymous array in a single line.

Array is a special class in Java which inherits from java.lang.Object. Java array type is very basic and it comes with limited built in functionality. However, Java provides a helper class named Java.Util.Arrays providing all the common functionalities and some static methods on arrays which can be used.

**Array initialization to default values –**

* For primitive types, this is zero for any kind of numeric primitive, like int, double or short.
* For booleans, the default value will be false.
* And for any class type, the elements will be initialized to null.

**Reference types versus value types –**

When you assign an object to a variable, the variable becomes reference to that object.

A screenshot of a computer

Description automatically generated

* Here we have instantiated an array called myIntArray using new keyword which is of length 5 elements.
* Here the variables myIntArray and anotherArray are referencing to the same array in memory.

**Variable arguments.** - We can replace the brackets after the String type in the main function with three periods. This is a special designation in Java that means Java will either take 0 or 1 or many strings as arguments to this method.

However, we need to remember 2 important concepts while using variable arguments.

* There can be only one variable argument in a method.
* The variable argument must be the last argument.

A screenshot of a computer

Description automatically generated

The above code throws an error because there is more than one variable argument as a parameter in the method.

A screen shot of a computer

Description automatically generated

We can resolve the error by having only one variable argument in the method as a parameter. And that parameter will be the last argument.

**Two-dimensional and multi-dimensional arrays :** Java supports 2 dimensional and three-dimensional arrays of varying dimensions using a concept called nested array.

Multidimensional arrays are implemented as arrays of arrays. We can specify each additional index using another set of square brackets.

* As shown in the below example, when we allocate memory for multi-dimensional array, we only need to specify the memory of first leftmost dimension. We can allocate the remaining dimensions separately.
* The only advantage of allocating the second dimensions Arrays individually is when we have different sizes of second dimension. However, the use of uneven or irregular multidimensional arrays may be not appropriate for many applications.

We can also initialize the two-dimensional array without specifying the size of nested arrays.

A close up of words

Description automatically generated

* These are the two different ways of initializing a 2-dimensional array. However, first method is preferred over the second to avoid confusion.

A white paper with text and numbers

Description automatically generated with medium confidence

**Limitations.:**

* One of the biggest limitations of arrays is that we won't be able to change the number of elements in an array after being instantiated.

Java has an entire library for Java containers which they called Collections. They take the arrays to the next level, they allow you to change the number of elements defined in an array. Two of the most important classes are.

* ArrayList
* LinkedList.

**List**: List is a special type in Java called interface., We can say that List interface describes a set of method signatures that all List classes are expected to have.

* It is an ordered collection, also known as sequence. The user of this interface has precise control over where in the list each element is inserted, and users can access elements by their integer index and search for the elements in the list. Unlike sets, lists typically allow duplicate elements.

The List interface extends SequencedCollection, **interface List<E>** Here, E specifies the type of objects that the list will hold.

**ArrayList:** Resizable array implementation of the List interface. This implements all optional list operations and permits all elements including null. In addition to implementing List interface, this class provides method to manipulate the size of array that is used internally to store the list.

Each arrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As the elements are added to the array list, its capacity grows automatically.

A screenshot of a computer program

Description automatically generated A screen shot of a computer program

Description automatically generated

* The first screenshot is using array list which is resizable. The second screenshot uses integer array where we need to instantiate the size at the beginning of declaration. The array list also lets us specify the type of elements/Parameters in the list. See the example. Grocery items are specified in the array list.
* Also, in order to print the arrays, we need a helper class From Java (Ararys.toString()). However, We can directly print array list.

**Common mistake while using array list** - Whenever we used parameterized class of Arraylists, we need to make sure we have <> diamond operator at the end of instantiation.

Unmodifiable list: The List.of and List.copyOf - List created using these static factory methods are immutable. That is, they are unmodifiable, i.e., elements cannot be added, removed or replaced. Calling any muter method on this list will always cause **UnsupportedOperationException** through be thrown.

ClassCastException generated when you attempt to add an incompatible object to the list.

IndexOutOfBoundsException in cases where an invalid index is used to retrieve the Elements.

NullPointerException is generated whenever you attempt to store a null object, and null elements are not allowed in the list.

A screen shot of a computer program

Description automatically generated A black screen with white text

Description automatically generated

**Arrays versus arrayLists**:

Arrays can store primitive data types (such as int, car and double) whereas ArrayList can only store objects, not primitive types. For an example, if you want to store integers in array list, you will use Integer wrapper object instead of int primitives.

A black screen with white text

Description automatically generatedA screen shot of a computer code

Description automatically generated

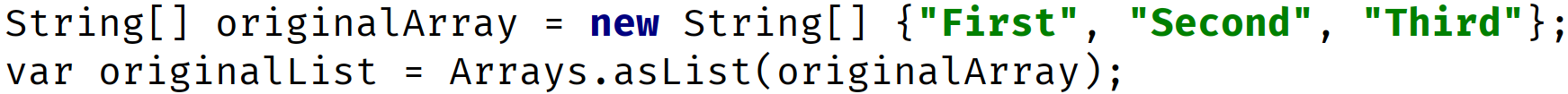
A white and pink box with black text

Description automatically generated with medium confidence

An ArrayList can be instantiated by passing another list to it as we show below - We can use the List.of() factory method, which uses variable arguments, to create a pass-through immutable list.A close-up of a computer screen

Description automatically generated

**Array as an array list –**



* The Arrays.asList method returns an ArrayList, backed by an array.
* The code uses the Arrays.asList method, passing it the array, and assign it to a variable, using the var keyword, to a variable named originalList.
* Any change made to the list is a change to the array that backs it. This means an array list created by this method is not resizable.

A close-up of a table

Description automatically generated

In this example, newList is a List returned by Arrays.asList(), but when we try to add an element to it, we get an UnsupportedOperationException because the list's size cannot be changed. However, we can modify the first element of the list returned by Arrays.asList() using the set() method. The list is mutable, so we can change its elements.

In this example, listOne is an immutable list returned by List.of(). Trying to add an element or modifying an element using set() also results in an UnsupportedOperationException.

* // Create a new ArrayList and pass the elements from the immutable list
* List<String> mutableList = new ArrayList<>(listOne);

In this example, we first create an immutable list using List.of(). Then, we create a new ArrayList named mutableList and pass the elements of the immutable list to it using the constructor ArrayList<>( listOne). Now, mutableList is a mutable list backed by an ArrayList, and we can modify its elements using methods like set().

**Arrays vs ArrayList vs LinkedList**:

A table with a list of different cases

Description automatically generated with medium confidence

**LinkedList:**

* Internally implemented as double linked list. Each element is stored in a separate node and nodes are linked to each other.
* Provides fast insertion and deletion of elements in the middle of the list because it just requires updating the link between nodes.
* Slower access time for retrieving elements by index because it needs to travel the list from head to tail or from tail to head to reach the desired index.
* It requires more memory overhead per element because each element is stored in a separate node and each node has additional pointers to the previous and next nodes.
* Used in scenarios where the number of elements are unpredictable and changes frequently.
* O(1) - constant time - operation's cost (time) should be constant regardless of no. of elements.
* O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.
* O(1)\* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.
* Use ArrayList when frequent random access to elements by index is required and the number of insertions or deletions in the middle is relatively low.
* Use LinkedList when frequent insertions or deletions in the middle of the list are required and random access to elements by index is less important.

Additionally, an array list is implemented on top of an array, but a linked list is a doubly linked list. Both implement all of list method but linked list also implements the queue and stack methods as well.

**ArrayList:**

* Elements are stored in continuous blocks of memory.
* Provides fast random access to elements using indexed based retrieval.
* Slower for adding and removing elements from the middle due to potential array resizing and element shifting.
* Requires less memory overhead per element. Since it only needs to store elements and an array. However, it waste a lot of memory if the initial capacity is set too high.
* Often used in scenarios where memory efficiency is a concern. It is usually better choice for a list, especially if the list is used predominantly for storing and reading data.

Linked lists can be used as a stack, a queue and a general list or as a double linked list. The LinkedList class extends AbstractSequentialList and implements the List, Deque and Queue Interfaces.

* **Stack:** A stack follows last in first out (LIFO) principle, you can use linked list as a stag by performing stack operation in its elements like push, pop or peak.
* **Queue**: A queue follows first in First Out FIFO principle, You can use linked list as a queue by performing queue operations such as enqueue, dequeue, peak.
* **General list**: Unlinked links can be used as a general purpose list by adding, removing, getting and setting elements.

A screenshot of a computer

Description automatically generated

**Stack:**

* Push: Add an element to the top of the stack (addFirst() or offerFirst()).
* Pop: Remove and return the element from the top of the stack (pollFirst() or removeFirst()).
* Peek: Return the element from the top of the stack without removing it (getFirst()).

**Queue**:

* Enqueue: Add an element to the end of the queue (addLast() or offerLast()).
* Dequeue: Remove and return the element from the front of the queue (pollFirst() or removeFirst()).
* Peek: Return the element from the front of the queue without removing it (getFirst()).

**General List:**

* Add: Add an element at a specified position in the list (add(int index, E element)).
* Remove: Remove an element from the list based on its index (remove(int index)).
* Get: Retrieve an element from the list based on its index (get(int index)).
* Set: Update an element at a specified position in the list (set(int index, E element)).
* We can obtain a sub list of a list by calling subList () specifying the beginning and ending indexes of the sub list.
* Starting JDK 9. List includes the of() factory method which has number of overloads. One takes no argument and creates empty list. In other cases, we can take 10 arguments and create a list that contains 10 elements.

**ListIterator:**

* We can use the traditional for loop and an index, to index into a list.
* We can use the enhanced for loop and a collection, to step through the elements, one at a time.

Java provides other means to traverse list: One is Iterator and the other is ListIterator. The iterator can be thought of as database cursor.

* When we create an instance of an iterator, we can call next method to get the next element in the list. We can use hasnext method to check if there are any elements remained to be processed.
* Difference between Iterator versus ListIterator is: iterator only moves forward and only supports the remove method whereas the list iterator can be used to go both forward (hasNext(), next()) and backward (hasPrevious(), previous()) and in addition to the remove method it also supports add and set methods.
* Linked list provides support for both forward and backward iteration through its elements, Using ListIterator, making it suitable for scenarios where bidirectional travel is required.

**Abstraction**: Terms like abstraction and generalization are used when we try to model real world things in software. When we start to model objects for our application, we identify common features and behaviors that objects have, so we generalize them. We create a class hierarchy; the base class is the most general class which has common things for objects.

* Part of generalization is using abstraction. For example, if we consider Octopus, Dog and Penguin, you'd probably say all these are animals. Here Animal is an abstract concept which doesn't exist, except you can use it to describe a set of specific things. Here we describe a set of traits and behavior and group them into a particular class based on their functionality.
* Similarly, think of an example car. It has sets of thousands of individual parts, however we think of it as a well-defined object with its own unique behavior.
* As a thumb rule, we manage complexity through abstraction. We hide all the implementation details under an abstract class which has its own unique behavior.

Java supports abstraction in several ways.

* It allows us to create class hierarchy where the top of hierarchy which is base class is usually an abstract concept whether it is an abstract class or not. Java lets us create abstract classes.
* Java lets us also create abstract methods.
  + An abstract method has a method signature and a written type but doesn't have a method body. In a way, we can say that abstract method is unimplemented.
  + The purpose of creating an abstract method is to describe behaviour which any object of that type will always have.

Abstract classes and interfaces can have a mix of both concrete and abstract methods.

* Concrete method: A concrete method has a method body with usually at least one statement. It has an operational code that is executed under the right conditions. It is said that a concrete method implements an abstract method if it overrides 1.

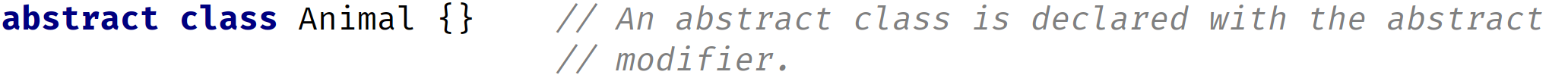
In addition to access modifiers (Public, private, protected), we also have method modifiers.

A screenshot of a computer

Description automatically generated

**Abstract Class:**

* An abstract class is declared with abstract modifier.



* We can say an abstract class is incomplete, hence we can't create instance of an abstract class.



* However, an abstract class can still have a constructor which will be called by its subclasses during their construction.
* **An abstract class** purpose is to define the behavior its subclasses are required to have, so it always participates in inheritance.
* Classes extending abstract classes can be an abstract class or a concrete class. Similarly, classes extending concrete class can be a concrete class or an abstract class.

Abstract method: Similar to abstract class, abstract method is also declared with modifier abstract, As mentioned, it can have a written type, but it always ends with semi colon; It doesn't have a body, not even curly braces.

**Note:** An abstract method can only be declared on an abstract class or interface. It cannot be declared on a concrete class.

**Why do we need an abstract class? What good is a class without any code in it?**

* Abstract classes define method signatures without providing implementation.

An abstract class in the class hierarchy forces the designers of subclasses to think about and create unique and targeted implementation for abstract methods. Since an abstract class can't be instantiated, if you are using abstract classes to design a framework for implementation, this is definitely an advantage..

* Subclasses of abstract methods must either implement all abstract methods or be declared abstract themselves. If a subclass failed to implement all abstract methods of its superclass, a compilation error occurs.
* This ensures that all subclasses adhere to the contract defined by the abstract class, guaranteeing the availability of required behavior in all subclasses.
* Abstraction helps in defining common behaviour across related classes while hiding the implementation details. Additionally, abstract classes can also achieve polymorphism, where objects of different concrete subclasses can be treated uniformly through their common abstract superclass.

**Interface**: Interface in Java is similar to abstract class, but it isn't a class, it is a reference type. It only contains constants (no instance variables), method signatures, default methods and static methods. It is more like a contract between the class and the client code that the compiler enforces. By declaring its using an interface, your class must implement all the abstract methods on the interface. However, each class is free to determine the details of its own implementation.

A screen shot of a computer code

Description automatically generated

By providing interface keyword, Java allows you to fully utilize one interface, multiple methods aspect of polymorphism. 

**Interface Declaration:** We declare an interface using interface keyword instead of using class keyword.

* Usually many interfaces will end in ‘able’ like Comparable, Iterable, Flight enabled. They are named according to the set of behavior’s it describe.
* While defining an interface, if no access modifier is included, then default access is given to the interface. That means it is only available to other members of the package in which it is declared. Whereas if it is declared as public, the interface can be used by code outside its package.

Using an interface (implements):

A close-up of a logo

Description automatically generated

* A class is associated to an interface by using **implements** clause in the class declaration.
* Because of the above declaration, we can now use FlightEnabled as the reference type and assign it to the instance of a bird. See below example.



**Note:** A class can only extend single class. This is why Java is called single inheritance whereas a class can implement many interfaces. This gives us plug and play functionality. A class can extend to another class and implement one or more interfaces.

* All interfaces are implicitly declared as abstract, you don't have to specify it explicitly.
* Similarly, all methods declared without a body of an interface are implicitly declared as both public and abstract.
* If we omit the access modifier on interface member, it implicitly is public, Whereas if we omit an access modifier on a class member, it implicitly is package private. A field/variable declared on an interface is implicitly public, static and final (This means the values of these variables cannot be changed by the class that is implementing the interface).
* If we try to change access modifier of our interface method to protected, it gives a compilation error. Only concrete methods can have private access.
* Any class that implements an interface, it has to implement all the abstract methods of those interfaces. Additionally, this class can also have methods of its own.

**Final keyword**: When you declare a field on interface, it actually means final static field (A field which cannot be reassigned or given a different value after its initialization.)

* A field declared as final static means the objects field cannot be reassigned or given different value after its initialization.
* A final variable in a block of code. This means once it is assigned a value, any remaining code in the block can't change it.
* A final variable in the method: It also means final method parameter, which means we can't assign a different value to a parameter in the method Code block.
* A final method means it cannot be overridden by a subclass.
* A final class can’t be overridden meaning no class can use it in the extents class.

**Variables in Interfaces: Constants in Java?**

Constant in Java is a variable that can't be changed. Constant in Java are usually named with uppercase letters and with underscores between words. A constant variable is a final variable of either primitive type or type String that is initialized with a constant expression.

E.g. INTEGER.MAX\_VALUE, and the INTEGER.MIN\_VALUE fields.

**Accessing implementations through interface reference**:

Here we can see that the class implementing interface is implementing all the abstract methods of the interface. Additionally, it is also having its own methods.

We can also declare variables as object references that uses an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. We can see that, variable calls the callback() via an interface reference variable.

We can see that variable C is declared to be of the interface type Callback, yet it was assigned to an instance of class Client.

Interface reference variables e.g. **C** do not have access to other members of the class. They can only access methods declared by interface declaration.

A close up of a text

Description automatically generated

A screenshot of a computer code

Description automatically generated

A white background with black text

Description automatically generated

Output:



**Partial implementation:** Any class that implements interface must implement all the abstract methods of interface. Otherwise the class must declare itself as abstract.

**Nested interface**: An interface can be declared as a member of a class or another interface. These interfaces. Are called member interfaces or nested interface. And this nested interface can be declared as public, private or protected. This is different from top level interface which either must be public or use default access level.

**Note:** when a nested interface is used outside of closing scope, It must be qualified by the name of the class or interface of which it is a member.

A white text with black text

Description automatically generatedA computer code with black text

Description automatically generated

Notice that A defines a member interface called NestedIF and that it is declared public. Next, B implements the nested interface by specifying implements **A.NestedIF**. Here the name is fully qualified by using enclosed class name.

**Interfaces can be extended:**

* For a class to implement interface we use implements keyword.
* For an interface to implement another interface we use extends keyword instead of implements keyword.
* A class cannot inherit multiple super classes using extends keyword. However an interface can use the extends expression with multiple interfaces.
* In this scenario, if a class implements an interface that inherits another interface, it must provide implementations for all the abstract methods required by interface inheritance chain. If not implemented, it causes compilation error.

A screen shot of a computer program

Description automatically generated A screen shot of a computer program

Description automatically generated

**Default interface methods**: In JDK 8 version, a new capability is added to interface called default method. A default method lets you define a default implementation for interface methods, means it allows you to provide a body rather than being abstract. By providing the default, the interface makes the implementation of default methods by class optional.

* It is important to note that default methods do not change the key aspect of interface. The interface still cannot have instance variables. This is still the major difference between the interface and a class. It is still not possible to create an instance for an interface by itself. An interface must be implemented by a class if an instance is to be created. The default method just gives you added flexibility.
* The default method in an interface is created by using default keyword before the method declaration.

**Multiple Inheritance Issue**: Classes can implement multiple interfaces, allowing them to inherit behavior from multiple sources without introducing conflict, because interfaces define contracts without providing implementation. However. Interfaces does not solve multiple inheritance issue because the difference between a class and interface is a class can maintain a state information especially using instant variables whereas interface cannot.

For example, let's consider a class that implements 2 interfaces and both the interfaces has same default method. In this case, which interface default method is used? What happens if a class has its own implementation of the method?

* A class implementation takes priority over interface default implementation. In this case both the interface default methods are overridden by class implementation.
* In case if one interface inherits other, both defining a common default method, the inheriting interface version of the method will take precedence.
* In case if one interface does not inherit other interface but both define a common default method And if class also does not override the default method, then an error will occur.

**Static interface methods:** Starting JDK 8, we can include one or more static methods in an interface.

**Note:** A static method defined by an interface can be called independently of any object. No implementation of interface is necessary. And no instance of interface is required.

Additionally, static interface methods are not inherited by either an implementing class or an sub interface.

**Private interface methods**: Starting JDK 9, Private methods can also be included in an interface. However, it cannot be used by code outside the interface in which it is defined. The restrictions also include some interfaces, because private interface methods are also not inherited by sub interface.

Starting Java 14, records and enums can implement interfaces. This allows them to provide implementation for the methods declared in those interfaces, providing additional flexibility in designing the code.

Interface

* In an interface, we define what kind of operation an object can perform. These operations are defined by the classes that implement the interface.
* Interfaces form a contract between the class, and the outside world, and this contract is enforced at build time, by the Java compiler.
* You can't instantiate interfaces, but they may contain a mix of methods declared with, or without an implementation.
* All methods on interfaces, declared without a method body, are automatically public and abstract.
* An interface can extend another interface.

Abstract Class

* With abstract classes, you can declare fields that aren't static and final, instance fields in other words.
* Also with abstract classes, we can use any of the four access modifiers for its concrete methods. We can also use all but the private access modifier, for its abstract methods.
* An abstract class can extend only one parent class, but it can implement multiple interfaces.
* When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it doesn't, then the subclass must also be declared abstract.

A white sheet with black text

Description automatically generated

**Generics**: It refers to a way of writing code that allowed types (Strings, integers, objects.) to be used as parameters while defining classes, interfaces or methods. We use generics to create classes, interfaces and methods that will work in a type safe manner with various kinds of data.

We can also say that the term generics means parameterized types. Parameterized types enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Here, T is the name of type parameter, which can be used As a placeholder for the actual type That will be passed to Gen class when an object is created.

A screenshot of a computer code

Description automatically generated

Gen uses a type parameter, Gen is a generic class, which is also called a parameterized type. In the declaration of generic class there is no Specific significance to the name T, any valid identifier could have been used, but is traditional.

The most commonly used type parameter identifiers are:

* E for Element (used extensively by the Java Collections Framework).
* K for Key (used for mapped types).
* N for Number.
* T for Type.
* V for Value.
* S, U, V etc. for 2nd, 3rd, 4th types.

Now consider Gen’s constructor: Notice that its parameter, o, is of type T. This means that the actual type of o is determined by the type passed to T when a Gen object is created. Also, because both the parameter o and the member variable ob are of type T, they will both be of the same actual type when a Gen object is created.

When you use generic classes, either referencing them or instantiating them, it's definitely recommended that you include a type parameter. But you can still use them without specifying one. This is called the Raw Use of the reference type.

* Generics allow the compiler to do compile-time type checking, when adding and processing elements in the list.
* Generics simplify code, because we don't have to do our own type checking and casting, as we would, if the type of our elements was Object.

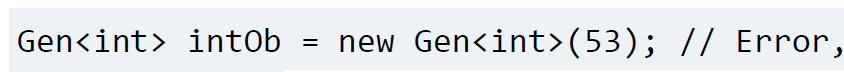
**Note**: **Generics only work with reference type** (String, Integer, Boolean, Double). With generic it is possible to pass any class type to T, but you cannot pass a primitive type (int, double, float, char) to a type parameter.

A screenshot of a computer code

Description automatically generated

A white paper with black text

Description automatically generated



**Generic types differ based on their type of argument:**

**A close-up of a letter

Description automatically generated**Even though both iOb and strOb are of type Gen<T>, they are references to different types because their type arguments differ. This is part of the way that generics add type safety and prevent errors.

**How generics improve type safety.?**

* With generics, type information is checked at compile time. This means that if you try to use a data type that is not compatible with the one expected by the generic code, the compiler will catch it and generate an error. This prevents type-related errors from occurring at runtime, which can be harder to debug.

A close up of text

Description automatically generated

* By using generics, we eliminate the need for manual typecasting, which simplifies the code and reduces the more chances of errors. You can simply specify the data type when creating an instance of the generic class, and the compiler ensures that only compatible types are used. If the parameterized type is not used for the class Gen, the return type of getOb( ) is Object, the cast to

A close-up of a text

Description automatically generated

* Integer is necessary to enable that value to be auto-unboxed and stored in v. If you remove the cast, the program will not compile.

**Generic classes with two type parameters:**  You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. so we could do T1, T2, T3. But again t instead of using type parameters like this, it's easier to read the code with alternate letter selections. And these are usually S, U, and V, in that order. If we had three types, we'd probably want to declare this class as shown here, with T, S, and U.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A black and white text

Description automatically generated with medium confidence





**Bounded Types:** When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived.  
 We can achieve this by using extends clause when specifying the type parameter.

* Here T extends superclass: This specifies that T can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit (In this example, T can be any class that extends Number, such as Integer, Double, or BigDecimal.)
* Lower Bound: T can be Number, Object, or any other class that Integer extends.
* In addition to using a class type as a bond, you can also use an interface type. In fact, you can specify multiple interfaces as bounds.
* A bound can include both class type and one or more interfaces.
  + In this case class type must be first.
  + When specifying a bound that has class and an interface or multiple interfaces, use the & operator to connect them. This creates an intersection type.

A screenshot of a computer

Description automatically generated



**Using wild card arguments:** Denoted by ? extends Type, where Type is a specific type. This wildcard allows any type that is a subtype of Type or Type itself.

**Generic Exception Restriction**: A generic class cannot extend Throwable. This means that you cannot create generic exception classes.

A close-up of a list

Description automatically generated

**Nested Classes:**

* A class usually contains attributes and methods. However, a class can contain other types within the class body, such as other classes, interfaces, enums, and records. These are called nested types or nested classes.

Note : Only static nested classes were allowed to have static methods before JDK 16. Now all four types of nested classes can have static members of any type, including static methods.

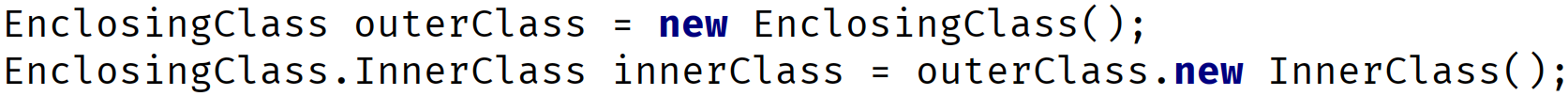


A computer screen shot of text

Description automatically generated

**Static Nested Classes:**

* It is a class enclosed in the structure of another class declaring it as static.
* To access this class externally, it requires the outer class name as a part of qualifying name. Here the static nested class is Employee Comparator. In order to access it, we require the outer class name that is Employee.
* The major advantage of static nested class is being to able to access private attributes of the outer class.
* Similarly, the outer class can access any attributes of the nested static class, including private attributes.





* Starting master classes doesn't have access to the instance variables and methods of their enclosing class.

Static nested classes can be used to logically group related functionality together. And also improve encapsulation (Nested static classes can access private members of their enclosing class, providing better encapsulation and access control. This allows the utility methods to interact with private members of the enclosing class, which might not be accessible from outside the class).

Lets take an example

* The PatientRecord class represents patient information such as patientId, name, age, and diagnosis.
* We have a nested static class DataUtils within PatientRecord to encapsulate utility methods for data validation and processing. These methods include isValidAge, isValidDiagnosis, and formatName.
* By grouping these utility methods within a nested static class, we improve the organization and clarity of our code. This is particularly useful because the PatientRecord class may have other non-static members and methods related to patient information, and we want to separate the utility methods from the rest of the class's functionality.

**Inner classes**: These are the non-static classes declared on an enclosing class at a member level.

* Unlike started nested classes, inner classes have access to the instance variables and methods of the enclosing class.
* To create an instance of an inner class, you first must have an instance of the enclosed class. From that instance you call .new keyword followed by the inner class name and the parenthesis taking any constructor arguments.

Inner classes are useful when you need to represent a relationship where one class is part of another class or when you want to encapsulate related functionality within the scope of an instance of the enclosing class.

nested static classes and inner classes are useful when you need to define classes that are closely related to an existing class and benefit from access to its members or when you want to logically group related classes together within a single class file.

**Local Classes:** classes are also inner classes, but they are directly declared in the code block, usually a method body.

* Since they are declared in a method body, they don't have access modifiers. They're only accessible to that method body while it's executing.
* However, like an inner class, local classes also have access to all the fields and methods on their enclosing class. Additionally they can also access local variables and method arguments that are final or effectively final

**Final vs effectively final:** Once you declare a variable final, you cannot assign a different value once these are initialized. These are explicitly declared final variables. If you assign a value to them and never change after that. These are called as effectively final variables.

**Why do we need to have local classes?**

If the functionality provided by the class is specific to certain methods and is not relevant outside of that context. We can use a local class to keep the implementation details close to where they are used. Here we are limiting the visibility to only the code within a particular method. This will help us to prevent unintended usage of a class elsewhere in the code base.

Local classes are meant to be localized implementation specific to particular method or context, whereas interfaces are designed to be global contracts that can be implemented by various classes. The reason you cannot define an interface inside a method (local interface) is because interfaces are meant to be contracts that can be implemented by multiple classes. Placing an interface inside a method would restrict its visibility and make it inaccessible to other parts of code, defeating the purpose of interfaces. Whereas local classes, including local interfaces, can be useful for encapsulating functionality within a specific method scope. They are tightly coupled with the method in which they are defined and are not meant to be accessed from outside that method. Local classes can access variables and parameters from the enclosing method, which can be useful for certain scenarios where you need to encapsulate logic within a specific context.

**Anonymous Class**: Anonymous Class is a local class that doesn't have a name. It is never created with a class declaration, but always instantiated as a part of expression. Anonymous class are replaced by lambda expression starting JDK 8.

Anonymous classes can extend the class or implement the interface. They're often used for one time implementation of interfaces or abstract classes.

Usually anonymous class is created inline without being assigned to a variable, and its purpose is to provide the implementation of the abstract method of an interface

* We can create anonymous class either by extending a class or by implementing an interface. We only use it in cases where we only need a Single object that implements interface, we can do it in one single statement and get the object we need for one time use.

**Lambda Expressions**: Lambda expression is essentially an anonymous method (Unnamed method.). However, this method is not executed on its own. Instead, it is used to implement a method defined by the functional interface/ we can say it forms the implementation of the abstract method defined by the functional interface. As a result, a Lambda expression can only be specified only in a context in which target type is defined. Clear. Go back.Lambda expression is similar to anonymous class. They're also referred as closures.

**Where's the method in the lambda expression?** For a lambda expression, the method is inferred by Java and this is an abstract method in the functional interface.

**Functional Interface:** It is an interface that has one and only abstract method. It is sometimes referred as SAM type where SAM stands for Single Abstract Method.

The standard example of functional interface is interface Runnable. This functional interface only defines one method called run()

* Functional interface defines the target type of a Lambda expression. And the Lambda expression can only be used in context in which its target type is specified.
* A Lambda expression consists of formal parameter list, usually not always declared in parenthesis; the arrow token; and then either an expression or a code block after the arrow token.

The Lambda operator or arrow operator divides Lambda expression into two parts. The left side specifies any parameters required by the Lambda expression. If no parameters are required an empty parameter list is used, on the right side is the Lambda body which specifies the action of Lambda expressions.

We define a Lambda expression to generate a random number and we assign the Lambda expression to a variable of type DoubleSupplier. Here DoubleSupplier is the functional interface which matches the signature of a Lambda expression.

Lambda bodies are of two types. One consists of a single expression; the other type consists of block of code.

A black background with white text

Description automatically generated

One key difference between single expression and the block of code expression in lambdas is you must explicitly use a return statement to return a value for a block code. For a single expression, you should never mention written statement.

**Lambda expression for a single/double parameter:**

In order for a Lambda expression to be used in a target type context, the type of abstract method from functional interface and the type of Lambda expression must be compatible. For example, if the abstract method specifies 2 int parameters, then the Lambda expression must specify 2 parameters which type is either explicitly int or implicitly referred as int by the context. Hello. Cortana. None of it. Hey, Cortana. What will the weather be for? Set my clock. And then the heater. Perfect. The string operations method has two parameters. The first type is.

A close-up of a computer code

Description automatically generated A white box with text and words

Description automatically generated with medium confidence

A close-up of a box

Description automatically generated A close-up of a white box

Description automatically generated

**A white sheet with black text

Description automatically generated**

A close up of a text

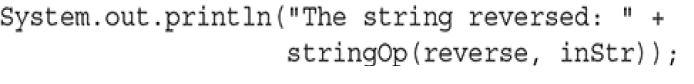
Description automatically generated

A close up of a word

Description automatically generated

A white background with black text

Description automatically generated



**Passing Lambda expressions as arguments**:

The stringOp( ) method has two parameters. The first is of type StringFunc, which is a functional interface. Thus, this parameter can receive a reference to any instance of StringFunc, including one created by a lambda expression. The second argument of stringOp( ) is of type String, and this is the string operated on.

The Lambda expression is assigned to reverse variable which is a reference to StringFunc instance All functional interface. Here we are passing Lambda expression as an argument to the method.

**java.util.function**: Java provides a library of functional interfaces in the java.util.function package.

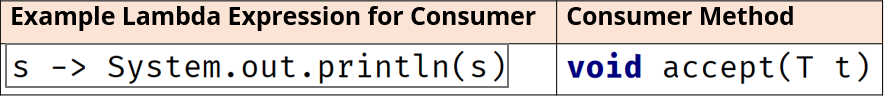
A close-up of a document

Description automatically generated

A close-up of a white box

Description automatically generatedA close-up of a sign

Description automatically generated

A close up of a sign

Description automatically generated

A table with text on it

Description automatically generated

* There are four basic types of functional interfaces in the java.util.function package.
* The Consumer interface takes one argument of any type similarlyBiConsumer interface take two arguments or two different. In the consumer example shown, it takes one argument and executes a single statement without returning any result.
* The Predicate interface takes one argument of any type and BIPredicate interface accepts argument of two different types. In the predicate example shown, the expression takes a string and tests if it is equal to the literal text, ignoring the case, and returns either true or false.

**Lambda expressions and exceptions:** A Lambda expression can throw an exception. However, if it throws a checked exception, then that exception must be compatible with the exceptions listed in the throws clause of abstract method in the functional interface.

A computer code with black text

Description automatically generated

We have created a custom functional interface that throws a custom empty array exception which extends the Exceptions class.

* Unchecked exceptions: Unchecked exceptions do not need special handling in lambdas.
* Checked exceptions: There are three ways to handle checked exceptions.
  + We handle checked exceptions within a Lambda expression using a try catch block.
  + Or we can read through checked exceptions as unchecked exceptions within the Lambda expression.
  + Or we can create a custom functional interface that allows for checked exceptions.

**Lambda expressions and variable capture**: when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a variable capture. In this case, a lambda expression may only use local variables that are effectively final. An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as final, although doing so would not be an error.

A computer screen shot of a code

Description automatically generated

The use of prefix variable from enclosing scope in our Lambda expression is valid. However, modifying the value of prefix variable inside the Lambda expression or after the Lambda expression is invalid.

As Lambda expressions may only use variables that are final or effectively final, so we cannot change the value of prefix. If modified, it causes an error.

**Method references**: Method references in Java are shorthand syntax for expressing A Lambda expression that executes just one method.

* A method reference abstracts the Lambda expression even further, eliminating the need to declare formal parameters. We also don't have to pass arguments to the method in question.
* A method reference has double columns between the qualifying type/ object and the method name.

A close up of a text

Description automatically generated

* It relates to Lambda expression because it also requires a target type context that consists of compatible functional interface. In simpler words, methods which can be used are based on the context of Lambda expression. This means the method reference is again dependent on the targeted interface method.
* Method references can be used to refer to a method in a class or object.

There are different types of method references:

* **Reference to a Static Method on a class :***ClassName::methodName*

Here sum method is the static method on Integer wrapper class.

* Reference to an Instance Method of a Particular Object(instance)
* Reference to an Instance Method of an Arbitrary Object of a Particular Type
* Reference to a Constructor ( by using new as the method)

**Why do we need to use Lambda expressions**?

Java is traditionally using Object oriented programming. With use of Lambda expressions, we can introduce functional programming concepts and techniques like map, filter and reduce on collections.

Lambda expressions also reduce the amount of boilerplate code needed for using single method interfaces (functional interfaces). Which is extremely useful in using Java Streams API (Where operations on collections can be chained together more fluidly.).