OOPs

Classes:

A class is a template for an object, and an object is an instance of a class.

A class creates a new data type that can be used to create objects.

When you declare an object of a class, you are creating an instance of that class.

Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.

Objects:

Objects are characterized by three essential properties: state, identity, and behavior.

The state of an object is a value from its data type. The identity of an object distinguishes one object from another.

It is useful to think of an object's identity as the place where its value is stored in memory.

The behavior of an object is the effect of data-type operations.

Dot Operator:

The dot operator links the name of the object with the name of an instance variable.

Although commonly referred to as the dot operator, the formal specification for Java categorizes the . as a separator.

New Keyword:

The 'new' keyword dynamically allocates(that is, allocates at run time)memory for an object & returns a reference to it.

This reference is, more or less, the address in memory of the object allocated by new.

This reference is then stored in the variable.

Thus, in Java, all class objects must be dynamically allocated.

This Keyword on primitive

You might be wondering why you do not need to use new for such things as integers or characters.

The answer is that Java's primitive types are not implemented as objects.

Rather, they are implemented as "normal" variables.

This is done in the interest of efficiency.

Inheritance:

"Inheritance in Java allows a class (subclass/child class) to acquire properties and behaviors (fields and methods) of another class (superclass/parent class). It promotes code reusability and establishes a relationship between classes."

Real-World Analogy:

"Think of inheritance like a family tree. For example, a Car class can be a parent class, and a Sedan or SUV can be child classes. These child classes inherit common properties like engine, wheels, and methods like start() from the Car class, but can have their own specialized features as well."

Example:

Parent myCar = new Child();

Child myCar = new Child();

Parent myCar : What one to use depends on this

new Child(): which one of the version to use depends on this

1. Parent myCar = new Child();

This is an example of upcasting and polymorphism in Java.

Explanation:

- Parent is the reference type (parent class).
- new Child() is the object type (child class).

This means that a reference of the parent class (Parent) is pointing to an object of the child class (Child). Java allows this because the child class inherits from the parent class, and thus a child object can be treated as an instance of the parent class.

2. Child myCar = new Child();

This is a direct reference to the child class.

Explanation:

- Child is both the reference type and the object type, meaning the reference and the object are of the same class (Child).

In this case, you can access both the inherited methods from the parent class and the methods or properties specific to the child class.

Concept	Parent myCar = new Child();	Child myCar = new Child();
Type of reference	Parent (Superclass)	Child (Subclass)
Type of object	Child (Subclass)	Child (Subclass)
Polymorphism	Yes (enables runtime polymorphism)	No (direct reference)
Method access	Can only access parent class methods (or overridden ones)	Can access both parent and child class methods
Child-specific methods	Not accessible unless cast to Child	Accessible directly

Super Keyword:

"In Java, the super keyword is used to refer to the parent (superclass) of the current object. It allows you to access methods, constructors, and variables from the parent class that may be overridden or hidden by the child class."

Use Cases for super:

- **Access Parent Class Methods:** Call a parent class's method that has been overridden in the child class.
- **Access Parent Class Variables:** Access a parent class's variable if it's hidden by a variable in the child class.
- **Call Parent Class Constructor:** Invoke a parent class constructor to initialize the parent class's properties.

Polymorphism:

Polymorphism in Java refers to the ability of a single interface or method to operate in multiple forms. It allows objects of different classes to be treated as objects of a common superclass, enabling a single action to behave differently based on the object that it is acting upon.

Types of Polymorphism:

1. **Compile-Time Polymorphism (Static Binding):** This type occurs when the method to be executed is determined at compile time. It is achieved through method overloading, where multiple methods have the same name but differ in parameters (type, number, or both).

2. **Runtime Polymorphism (Dynamic Binding):** This occurs when the method to be executed is determined at runtime. It is achieved through method overriding, where a subclass provides a specific implementation of a method already defined in its superclass.

Encapsulation:

Encapsulation is a fundamental concept in object-oriented programming that involves bundling the data and methods that operate on that data into a single unit, typically a class. It restricts direct access to some of an object's components, helping to protect the integrity of the data.

Real-World Example: A Bank Account:

In a bank account system, we want to manage sensitive information like the account balance and account holder's details. We need to ensure that this information is protected and can only be accessed or modified in controlled ways.

In encapsulation getter and setter methods are used to reterive the information.

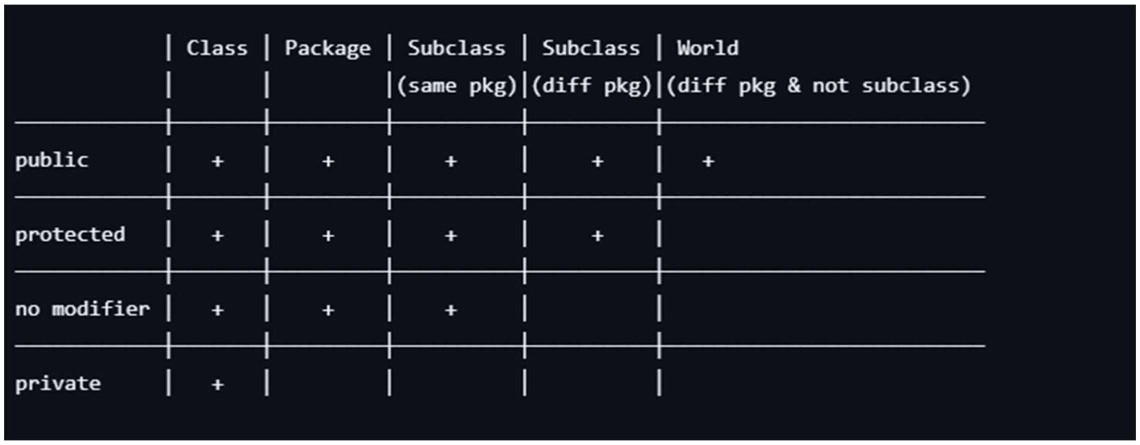
Abstraction:

Abstraction is an object-oriented programming principle in Java that focuses on hiding the implementation details and exposing only the essential functionalities. It allows you to define abstract concepts or interfaces, making your code more flexible and maintainable by separating 'what' an object does from 'how' it does it.

Real-World Example: A Car

Let's consider a real-world example of a car. When you drive a car, you interact with the steering wheel, pedals, and dashboard controls. You don't need to know how the engine works or how the fuel system operates—that's all abstracted away from you.

Access Control



	Class	Package	Subclass	Subclass	World
			(same pkg)	(diff pkg)	(diff pkg & not subclass)
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

Abstract Class:

An abstract class in Java is a class that cannot be instantiated on its own. It is designed to be extended by other classes and can contain both abstract methods (without a body) and concrete methods (with a body).

Purpose: Abstract classes provide a base structure for other classes to build upon. They allow you to define general behavior that all subclasses must follow, while giving subclasses the flexibility to provide their specific implementations.

Real-World Example: Parents Allowing Their Child to Choose Their Own Profession (Abstract Class Example)

In this real-world analogy, we can think of **parents** as an **abstract class**, and their **children** as subclasses that choose their own profession. The parents set up a general framework (like basic education, values, etc.), but the children are free to choose their own specific professions (like becoming a doctor, engineer, or artist).

Interface:

An interface in Java is a reference type that defines a contract for classes. It specifies a set of methods that a class must implement, but without providing the actual implementation of those methods. Unlike abstract classes, interfaces provide 100% abstraction, meaning they do not have any concrete (implemented) methods (until Java 8 introduced default methods).

Real world example:

Imagine the car having engine and media player both engine and media player as the method start method. Whenever the object is created for the car and start function is called java is confused to call which method interface helps to solve this problem it is the example for the multiple inheritance.

Abstract class vs Interface:

Type of methods:

Interface can have only abstract methods.

Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.

Final Variables:

Variables declared in a Java interface are by default final.

An abstract class may contain non-final variables.

Type of variables:

Abstract class can have final, non-final, static and non-static variables.

Interface has only static and final variables.

Implementation:

Abstract class can provide the implementation of interface.

Interface can’t provide the implementation of abstract class.

Inheritance vs Abstraction:

A Java interface can be implemented using keyword “implements”

and abstract class can be extended using keyword “extends”.

Multiple implementation:

An interface can extend another Java interface only,

an abstract class can extend another Java class and implement multiple Java interfaces.

Accessibility of Data Members:

Members of a Java interface are public by default.

A Java abstract class can have class members like private, protected, etc.

Generic:

Generics in Java allow you to create classes, interfaces, and methods that can operate on types specified by the user at compile time, without the need to cast or use raw types. Generics provide type safety and code reusability by allowing you to define methods or classes that can work with any type, while still enforcing compile-time type checks.

Real world example:

Imagine you are running an e-commerce store, and you need to ship various products, such as books, clothes, or electronics, to customers. You use a 'Box' to pack these items. The concept of generics is like using the same box to ship different types of items—Books, Clothes, or Electronics—without needing to create separate boxes for each type. This allows flexibility while ensuring that only the correct item is placed in the right box.

Object Cloning:

Object cloning in Java is the process of creating an exact copy (clone) of an existing object. Java provides a built-in mechanism to clone objects using the Cloneable interface and the clone() method from the Object class.

Shallow Copy:

- **"Shallow copying"** copies the object and its primitive fields. However, for reference fields, it copies only the reference, not the actual object the reference points to.

Deep Copy:

- **"Deep copying"** copies the object and also creates copies of any referenced objects (non-primitive fields). This ensures that all objects are fully duplicated, not just the references.

Real-World Example:

Shallow Copy: Photocopy of a Document with Attachments

- Imagine you have a document with some important pages and a few attached documents (like a report with papers stapled to it).
- If you make a shallow copy of this document, you're photocopying the main pages but NOT the attached documents. Instead, in your photocopy, you're using the same original attachments from the first document.
- In the shallow copy, changes to the attached documents affect both the original and the copied documents because they are still referring to the same attachments.

Deep Copy: Full Copy of Document and Attachments

- Now, imagine making a deep copy of the document. Not only do you photocopy the main pages, but you also make copies of each attached document.
- In this case, you have a complete and independent duplicate. Modifying the copied attachment won't affect the original document because it’s a separate, independent copy.