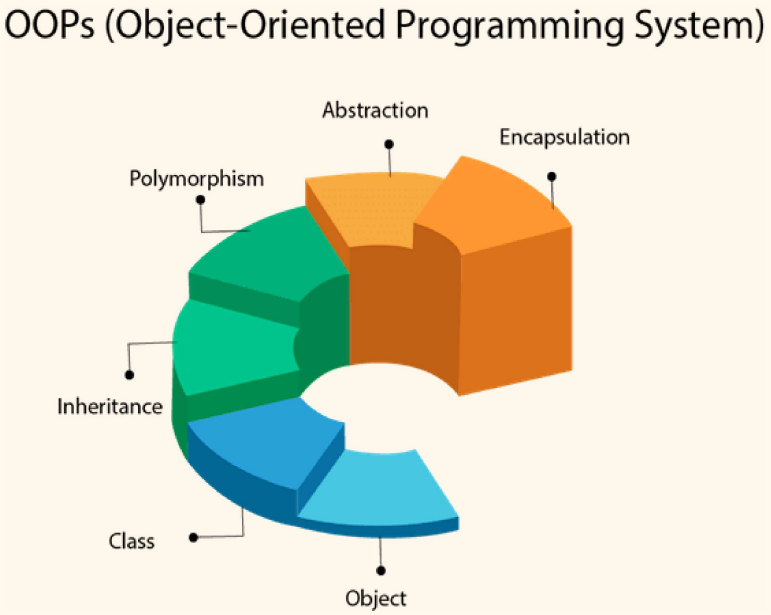
OBJECT ORIENTED PROGRAMMING

Object Oriented Programming CheatSheet - by Love Babbar (whimsical.com)
OOP stands for Object-Oriented Programming.
Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.
Object-oriented programming has several advantages over procedural programming:

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

Object means a real-world entity such as a pen, chair, table, computer, watch, etc.
Object-Oriented Programming (OOP) is a programming paradigm in computer science that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
It simplifies software development and maintenance by providing some concepts.

- Object
- Class
- Inheritance
- Polymorphism



- Abstraction
- Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

How object oriented programming is related to the real world?

- **Abstraction:** OOP allows developers to represent real-world entities and their characteristics in a simplified manner. Classes in OOP act as blueprints for objects, defining their attributes (data) and behaviors (methods). This abstraction enables programmers to focus on essential features and hide unnecessary details.
- **Encapsulation:** Encapsulation in OOP involves bundling the data (attributes) and methods (behaviors) that operate on the data into a single unit, known as an object. This mirrors the real-world scenario where objects encapsulate both state and behavior. Encapsulation enhances modularity and allows for more manageable and maintainable code.

- **Inheritance:** Inheritance in OOP allows one class to inherit the properties and behaviors of another class. This concept mirrors the hierarchical relationships found in the real world. For example, a "Vehicle" class may have sub-classes like "Car" and "Motorcycle" that inherit common features from the parent class. Inheritance promotes code reuse and reflects the natural structure of objects.
- **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. This mirrors the real-world scenario where different objects may exhibit similar behaviors or respond to the same interface. Polymorphism enhances flexibility and extensibility in the code.
- **Modeling Real-World Relationships:** OOP provides a natural way to model relationships between entities in the real world. Associations, compositions, and aggregations can be expressed through class relationships. For example, in a banking application, you might have classes like "Customer," "Account," and "Transaction" with relationships that mimic real-world connections.
- **Real-world Analogy:** OOP's design is inspired by the way humans perceive and interact with objects in the real world. By using familiar concepts such as objects, classes, and inheritance, developers can create software that is easier to understand, maintain, and extend. This helps bridge the gap between the conceptualization of a problem in the real world and its implementation in code.

Why do we need OOPs?

1. Duplicate code is Bad.
2. Code will always be changed.

So, OOP provides code reusability which reduces the duplication of code because once you have duplicate code, you have to make changes everywhere which leads to performance. Code can be changed anytime or requirement of application changed anytime so when you want to make changes in your application, OOP makes it easier.

Advantages of OOP:

1. OOP provides a clear modular structure for programs which makes it good for defining abstract data types where implementation details are hidden and the unit has a clearly defined interface.
2. OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
3. OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.

Limitations of OOP:

1. Steep Learning Curve: OOP concepts such as encapsulation, inheritance, and polymorphism can be challenging for beginners to grasp. The paradigm introduces a set of abstract and theoretical concepts that may take time for developers to fully understand and apply effectively.
2. Performance Overhead: OOP can introduce some performance overhead. The use of objects and dynamic dispatch (polymorphism) can lead to slower execution compared to more procedural or functional approaches, especially in resource-constrained environments.
3. Complexity and Verbosity: Large-scale OOP systems can become complex and verbose. The need for numerous classes, relationships, and design patterns can result in code that is harder to read, understand, and maintain. Overuse of inheritance, in particular, can lead to a deep and complex class hierarchy.

4. Difficulty in Parallel Processing: OOP systems may face challenges in parallel processing and concurrency. Managing shared state and coordinating activities among multiple objects can be complex, leading to potential issues such as race conditions and deadlocks.
5. Not Always Suitable for All Problems: While OOP is a great fit for many scenarios, it may not be the best choice for every problem. Some problems are better addressed using procedural programming, functional programming, or other paradigms. Over-reliance on OOP in inappropriate situations can lead to suboptimal solutions.
6. Overhead of Abstraction: Abstraction, a key feature of OOP, can sometimes result in an additional layer of complexity. The process of abstracting away details may make it harder to understand the inner workings of a system, especially for developers who did not participate in the original design.
7. Difficulty in Modeling Real-World Relationships: While OOP is designed to model real-world entities and their relationships, it may not always capture certain complex relationships accurately. For example, some real-world relationships may not fit neatly into the classical inheritance hierarchy, leading to design challenges.
8. Rigidity in Design: Once a system is designed and implemented using OOP principles, making significant changes to the architecture can be challenging. Modifications to the base classes or changes in the inheritance hierarchy may have a cascading effect, requiring extensive refactoring.
9. Not Ideal for All Types of Projects: Small, simple projects may not benefit significantly from the complexities introduced by OOP. In such cases, the overhead of designing and maintaining a system with multiple classes and objects may outweigh the advantages.

Approach to Object Oriented Design:

1. Start with the simple object which can be abstracted into individual classes.
2. Identify all the classes in the requirement specification.
3. Identify the commonalities between all or small groups of classes. Do not force fit generalization where it doesn’t make sense.
4. Keep all the data members private or protected
5. Identify all the member variables and methods the class should have
6. Ensure that the class is fully independent of other classes and contains all the necessary attributes and methods.
7. The methods in the class should be abstract.
8. Don't use the procedural code into a class for the methods in the class.
9. Inherit and extend classes from the base classes when required.
10. Define the "Has-A" or "Uses-A" relationships among the classes.

Class:

Classes are a blueprint or a set of instructions to build a specific type of object. It is a basic concept of Object-Oriented Programming which revolves around real-life entities. Class in Java determines how an object will behave and what the object will contain.

Properties of Java Classes:

- Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- Class does not occupy memory.
- Class is a group of variables of different data types and a group of methods.
- A Class in Java can contain:
    - Data member
    - Method
    - Constructor
    - Nested Class

- Interface

Components of Java Classes:

In general, class declarations can include these components, in order:

- Modifiers: A class can be public or has default access.
- Class keyword: class keyword is used to create a class.
- Class name: The name should begin with an initial letter (capitalized by convention).
- Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- Body: The class body is surrounded by braces, { }.

While creating a class, one must follow the following principles:

- Single Responsibility Principle (SRP)- A class should have only one reason to change
- Open Closed Responsibility (OCP)- It should be able to extend any classes without modifying it
- Liskov Substitution Responsibility (LSR)- Derived classes must be substitutable for their base classes
- Dependency Inversion Principle (DIP)- Depend on abstraction and not on concretions
- Interface Segregation Principle (ISP)- Prepare fine grained interfaces that are client specific.

Class vs Structure:

Class	Structure
1. Members of a class are private by default.	1. Members of a structure are public by default.
2. An instance of a class is called an 'object'.	2. An instance of structure is called the 'structure variable'.
3. It is declared using the class keyword.	3. It is declared using the struct keyword.
4. It is normally used for data abstraction and further inheritance.	4. It is normally used for the grouping of data
5. NULL values are possible in Class.	5. NULL values are not possible.

6. Syntax:
class class_name{
data_member;
member_function;
};
	6. Syntax:
	struct structure_name{
	type structure_member1;
	type structure_member2;
	};

Similarities between class and structure:

1. Both can have members, such as constructors, methods, properties, fields, constants, enumerations, events, and event handlers.
2. Both can declare some of their members private.

3. Both support inheritance mechanisms.
4. Both are user defined types.
5. Both can implement interfaces.

When to use struct over class?

1. Size and Copy Semantics: Use a struct when dealing with small, lightweight objects that have value semantics. In languages like C++, struct instances are often passed by value, which means that the entire object is copied when passed as a function argument or returned from a function.
2. Immutable Data: If your data structure is intended to be immutable (i.e., its state doesn't change after creation), a struct can be a good choice. Immutable structs are often used for representing mathematical vectors, points, or other data where mutation is not required.
3. Performance Considerations: For small, simple data structures that are frequently created and discarded, a struct can be more efficient than a class. This is because a struct is typically allocated on the stack, avoiding heap allocation and reducing the burden on the garbage collector.
4. Default Member Accessibility: The default member accessibility for a struct is public, while for a class, it's private. If you want all the members of your type to be public by default, a struct might be more convenient.
5. Logical Grouping: Use a struct when your data represents a logically grouped set of values, and there is no need for encapsulation or complex behavior associated with the data.
6. Interoperability: In some scenarios, interoperability with other languages or systems might favor the use of struct. For example, when working with platform invoke (P/Invoke) in C# to call native code, using a struct with sequential layout can be beneficial.

Access Modifiers:

Access modifiers in Java are the keywords that are used for controlling the use of the methods, constructors, fields. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Access Modifiers	Non-Access Modifiers
private default or No Modifier protected public	static final abstract synchronized transient volatile strictfp

There are four types of Java access modifiers:

1. Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package N0on-Subclass	No	No	No	Yes

Friend & Protected Friend:

Friend Class and Function in C++ - GeeksforGeeks

Friend: In C++, the friend keyword is used to grant a function or class access to the private and protected members of another class. This concept allows a specified function or class to access the private and protected members of the class that declares it as a friend. This is often used to provide certain classes or functions with privileged access to the internals of a class without making those members public.
Friend Functions is a reason why C++ is not called a pure Object Oriented language. Because it violates the concept of Encapsulation.

Protected Friend: The term "protected friend" is not a standard concept in C++. However, if you are referring to using the protected access specifier along with the friend keyword, it could imply a situation where a class is marked as a friend and granted access to protected members.

Member Functions of Classes in C++:

C++ Member Functions in Classes | C++ Tutorial | Studytonight

Member functions are the functions, which have their declaration inside the class definition and work on the data members of the class. The definition of member functions can be inside or outside the definition of class. If the member function is defined inside the class definition it can be defined directly, but if it's defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.
If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube
{
   public:
   int side;
   int getVolume()   // member function defined inside class definition
   {
      return side*side*side;
   }
};
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
class Cube
{
   public:
   int side;
   int getVolume();
```

```
}
int Cube :: getVolume()      // member function defined outside class definition
{
   return side*side*side;
}
```

Types of Class Member Functions in C++:

In C++, member functions can be classified into different types based on their properties and behavior. Here are some common types of member functions:

1. **Non-static Member Functions:** These are the regular member functions associated with instances of a class. They can access both the instance's data members and other member functions.

```
class MyClass {
public:
   void nonStaticFunction() {
      // Regular member function
   }
};
```

2. **Static Member Functions**: Static member functions are associated with the class itself rather than instances of the class. They can only access static members and other static functions.

```
class UtilityClass {
public:
   static void staticFunction() {
      // Static member function
   }
};
```

3. **Const Member Functions:** Const member functions are methods that promise not to modify the state of the object on which they are called. They are declared with the const keyword.

```
class ReadOnlyClass {
public:
   int getValue() const {
      return data;
   }

private:
 int data;
};
```

4. **Inline Member Functions:** An inline member function is a member function that the compiler is encouraged to expand inline rather than generating a separate function call. It is declared with the inline keyword.

```
class InlineClass {
public:
   inline void inlineFunction() {
      // Inline member function
   }
};
```

5. **Virtual Member Functions:** Virtual member functions are used in polymorphism. They allow derived classes to provide a specific implementation while being called through a base class pointer or reference.

```
class BaseClass {
public:
   virtual void virtualFunction() {
      // Virtual member function
   }
};
class DerivedClass : public BaseClass {
public:
   void virtualFunction() override {
      // Overridden virtual member function
   }
};
```

6. **Pure Virtual Functions:** A pure virtual function is a virtual function that has no implementation in the base class. It is marked with = 0 and must be implemented by any derived class.

```
class AbstractClass {
public:
   virtual void pureVirtualFunction() = 0;
};
```

7. **Friend Functions:** Friend functions are not member functions, but they are associated with a class by being declared as a friend inside the class. They can access private and protected members of the class.

```
class MyClass {
private:
   int data;

   friend void friendFunction(MyClass& obj);
};
void friendFunction(MyClass& obj) {
   // Accessing private member as a friend function
   int value = obj.data;
}
```

Inline keyword:

The inline keyword in C++ is a hint to the compiler that suggests it should perform inline expansion of a function, potentially replacing the function call with the actual body of the function at the call site. This can result in more efficient code by avoiding the overhead of a function call.

Inline Function Syntax:

To declare an inline function, you can use the inline keyword in the function declaration:

```
inline int add(int a, int b) {
   return a + b;
}
```

Here, the add function is declared as inline. The decision of whether the function is actually inlined is up to the compiler; it may choose to ignore the inline keyword in certain cases.

Advantages of Inline Functions:

1. Performance Improvement: Eliminates the overhead of a function call, which can lead to improved performance, especially for small and frequently called functions.
2. Reduced Function Call Overhead: Helps reduce the overhead associated with the function call, such as pushing and popping parameters and managing the function call stack.

3. Code Size Reduction: Can lead to smaller executable code size because the code of the function is inserted directly at the call site, eliminating the need for a separate function body.

Guidelines and Considerations:

1. Use for Small Functions: Inline functions are most effective for small functions. For larger functions, inlining may result in code bloat.
2. Definition in Header Files: In C++, inline functions are often defined in header files to allow the compiler to perform inline expansion in multiple translation units.
3. Compiler Discretion: The inline keyword is a hint to the compiler, not a strict command. The compiler may choose not to inline a function if it deems that inlining is not beneficial.
4. Balance Between Size and Speed: The decision to use inline should be a trade-off between code size and speed. Inlining too many functions might increase code size and reduce the effectiveness of the CPU cache.

```
#include <iostream>
inline int square(int x) {
   return x * x;
}
int main() {
   int result = square(5); // Compiler may inline the function
   std::cout << "Result: " << result << std::endl;
   return 0;
}
```

In this example, the square function is declared as inline. The compiler may choose to inline the function, leading to more efficient code.

Scope resolution operator (::):

The scope resolution operator (::) in C++ is used to access members (variables or functions) that are defined in a particular scope, such as a class or namespace. It allows you to qualify a member with the scope in which it is declared.

Usage in Classes: In the context of classes, the scope resolution operator is used to access members of a class, especially when there is a need to disambiguate between local and class members or when dealing with static members.

```
#include <iostream>
class MyClass {
public:
   static int staticVar; // Static member variable
   void display() {
      std::cout << "Display function in MyClass." << std::endl;
   }
};
int MyClass::staticVar = 42;          // Definition of static member variable
int main() {
   MyClass obj;
      obj.display();
      std::cout << "Static variable: " << MyClass::staticVar << std::endl;       // Accessing a static
member using the scope resolution operator
  return 0;
}
```

In this example, MyClass::staticVar is how you access the static member variable of the class outside the class scope.

Usage in Namespaces:

The scope resolution operator is also used to access members within a namespace.

```
#include <iostream>
namespace Math {
   const double PI = 3.14159;
   double calculateArea(double radius) {
      return PI * radius * radius;
   }
}
int main() {
   double radius = 5.0;
   std::cout << "Area: " << Math::calculateArea(radius) << std::endl; // Accessing a constant within
a namespace
   return 0;
}
```

In this example, Math::PI and Math::calculateArea demonstrate the use of the scope resolution operator to access members within a namespace.

Nested Classes: In the context of nested classes, the scope resolution operator is used to access members of the outer class from within the nested class.

```
#include <iostream>
class OuterClass {
public:
   int outerVar;
   class NestedClass {
   public:
      void displayOuterVar(OuterClass& obj) {
         std::cout << "Outer variable from nested class: " << obj.outerVar << std::endl;
      }
   };
};
int main() {
   OuterClass obj;
   obj.outerVar = 10;
   OuterClass::NestedClass nestedObj;
   nestedObj.displayOuterVar(obj);
   return
```

Here, OuterClass::NestedClass uses the scope resolution operator to access the outerVar member of the outer class.

In summary, the scope resolution operator in C++ is a versatile tool for accessing members in different scopes, including classes, namespaces, and nested classes. It allows you to qualify the scope of a member and disambiguate between local and external names.

Constructor:

A Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

How Java Constructors are Different From Java Methods?

- Constructors must have the same name as the class within which it is defined; it is not necessary for the method in Java.

- Constructors do not return any type while method(s) have the return type or void if it does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

- There are no “return value” statements in the constructor, but the constructor returns the current class instance. We can write ‘return’ inside a constructor.

Can a Java constructor be private?

Yes, a constructor can be declared private. A private constructor is used in restricting object creation.

Need of Constructor:

1. **Object Initialization:** Constructors are used to initialize the state of objects when they are created. They set the initial values of instance variables and perform any necessary setup for the object to be in a valid and usable state.

```
public class Person {
   private String name;
   private int age;
   // Constructor for initializing a Person object
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

2. **Encapsulation and Access Control:** Constructors can be used to enforce encapsulation and control the access to the internal state of an object. By making certain variables private and initializing them in a constructor, you can ensure that the object is properly set up according to the class's design.

```
public class BankAccount {
   private double balance;
   // Constructor for initializing a BankAccount object
   public BankAccount(double initialBalance) {
      if (initialBalance >= 0) {
         this.balance = initialBalance;
      } else {
         throw new IllegalArgumentException("Initial balance must be non-negative");
      }
   }
}
```

3. **Default Values:** Constructors can provide default values for class members. If a constructor is not explicitly defined, Java provides a default constructor with no arguments. However, if you want to provide specific default values or perform additional setup, you can define your own constructor.

```
public class Circle {
   private double radius;
   // Constructor with default radius
   public Circle() {
      this.radius = 1.0;
   }
   // Constructor with specified radius
   public Circle(double radius) {
```

```
        this.radius = radius;
    }
}
```

4. **Constructor Overloading:** Constructors support overloading, allowing a class to have multiple constructors with different parameter lists. This enables flexibility in creating objects with varying sets of initial values.

```
public class Book {
    private String title;
    private String author;
    // Constructor with only title
    public Book(String title) {
        this.title = title;
        this.author = "Unknown";
    }
    // Constructor with both title and author
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
}
```

5. **Initialization Blocks:** Constructors can include initialization blocks (using instance initializers) to execute additional code during object creation. This is helpful for complex initialization logic that goes beyond simple variable assignments.

```
public class ComplexObject {
        private int x;
        private int y;
        // Constructor with initialization block
        public ComplexObject(int x, int y) {
            this.x = x;
            this.y = y;
            // Additional initialization logic
            {
                // Complex initialization code
            }
        }
    }
```

In summary, constructors in Java are crucial for initializing objects, enforcing encapsulation, providing default values, supporting overloading, and executing additional setup logic during object creation. They are an integral part of Java classes, contributing to the overall robustness and usability of the object-oriented design.

Default Constructor:

If we do not provide any constructor in the class, JVM provides a default constructor to the class during compile time. In the default constructor, the name of the constructor MUST match the class name, and it should not have any parameters.

A constructor that has no parameters is known as default the constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But the Parameterized constructor can’t change the default constructor.

```
public class MyClass {
```

```
    // Default constructor
    public MyClass() {
        // This constructor is empty
    }
}
```

Parameterized Constructor:

A parameterized constructor is a constructor that accepts one or more parameters. It allows you to initialize the object with specific values during its creation. This is useful when you want to customize the initialization of an object based on the values provided at the time of instantiation.

```
public class Car {
    private String make;
    private String model;
    private int year;
    // Parameterized constructor
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
}
```

Copy Constructor:

A copy constructor in Java is a constructor that takes an object of the same class as a parameter and creates a new object by copying the values of the fields from the passed object. This allows you to create a new object that is a copy of an existing object.

The copy constructor is particularly useful when you want to create a new object with the same state as an existing object, providing a convenient way to duplicate objects.

```
public class Person {
    private String name;
    private int age;
    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Copy constructor
    public Person(Person otherPerson) {
        this.name = otherPerson.name;
        this.age = otherPerson.age;
    }
}
```

Shallow Copy:

- A shallow copy creates a new object but does not duplicate the content of the object deeply.
- If the object contains references to other objects, the shallow copy only copies the references, not the actual objects themselves.
- Changes made to the internal state of the copied object are reflected in the original object, and vice versa, if the internal state contains mutable objects.

```
public class ShallowCopyExample {
    public static void main(String[] args) {
        // Original object
```

```
        Person person1 = new Person("Alice", 25);
        person1.setAddress(new Address("123 Main St"));
        // Shallow copy using copy constructor
        Person person2 = new Person(person1);
        // Modifying the copied object's address
        person2.getAddress().setStreet("456 Oak St");
        // Changes are reflected in the original object
        System.out.println("Person 1 address: " + person1.getAddress().getStreet()); // Outputs: 456
Oak St
    }
}
```

Deep Copy:

- A deep copy creates a new object and recursively copies the content of the original object, including any objects referenced by the original object.
- Changes made to the internal state of the copied object do not affect the original object, and vice versa.

```
public class DeepCopyExample {
    public static void main(String[] args) throws CloneNotSupportedException {
        // Original object
        Person person1 = new Person("Bob", 30);
        person1.setAddress(new Address("789 Pine St"));
        // Deep copy using clone() method
        Person person2 = (Person) person1.clone();
        // Modifying the copied object's address
        person2.getAddress().setStreet("101 Maple St");
        // Changes do not affect the original object
        System.out.println("Person 1 address: " + person1.getAddress().getStreet()); // Outputs: 789
Pine St
    }
}
```

Shallow copy vs Deep Copy:

Shallow Copy	Deep Copy
Shallow Copy stores the references of objects to the original memory address.	Deep copy stores copies of the object's value.
Shallow Copy reflects changes made to the new/copied object in the original object.	Deep copy doesn't reflect changes made to the new/copied object in the original object.
Shallow Copy stores the copy of the original object and points the references to the objects.	Deep copy stores the copy of the original object and recursively copies the objects as well.
A shallow copy is faster.	Deep copy is comparatively slower.

Copy Constructor vs Assignment Operator:

Copy constructor	Assignment operator
It is called when a new object is created from an existing object, as a copy of the existing object	This operator is called when an already initialized object is assigned a new value from another existing object.
It creates a separate memory block for the new object.	It does not create a separate memory block or new memory space.
It is an overloaded constructor.	It is a bitwise operator.
C++ compiler implicitly provides a copy constructor, if no copy constructor is defined in the class.	A bitwise copy gets created, if the Assignment operator is not overloaded.

Syntax:
className(const className &obj) {
// body
}
	Syntax:
	className obj1, obj2;
	obj2 = obj1;

Virtual Constructor:

- In some languages, a "virtual constructor" refers to a mechanism where the actual class of an object is determined dynamically at runtime. This typically involves using virtual functions (methods) and polymorphism.
- In Java, polymorphism is achieved through method overriding and dynamic dispatch. The Object class has a method called getClass() that returns the runtime class of an object. This can be considered a form of dynamic type identification.

Virtual Copy Constructor:

- The term "virtual copy constructor" is not a standard term but can be interpreted as a mechanism for creating a copy of an object where the actual class of the object is determined dynamically at runtime.
- Achieving this kind of behavior may involve a combination of polymorphism and a factory method or some form of cloning, depending on the specific requirements.

Destructor:

Java Destructor - Javatpoint

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

- A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence the destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.

- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In a destructor, objects are destroyed in the reverse of an object creation.

Java does not have a built-in destructor concept like some other programming languages (e.g., C++). In Java, objects are managed by the garbage collector, and developers don't explicitly need to deallocate memory or perform cleanup tasks for objects. Instead, the Java Virtual Machine (JVM) automatically takes care of reclaiming memory when objects are no longer reachable.

In Java, there is a concept called finalization, which involves the finalize() method. The finalize() method is part of the Object class, and you can override it in your class to provide custom cleanup code. However, using finalize() is not recommended for general resource management, as its execution is not guaranteed, and it may introduce latency.

It's important to note that relying on finalize() for resource cleanup is discouraged because: Execution is not guaranteed: The finalize() method might not be executed promptly or at all, and it depends on the garbage collector's behavior.
Resource leaks: If your cleanup logic involves releasing external resources (e.g., closing files, sockets, or database connections), relying on finalize() might lead to resource leaks.

How does destructor work?

When the object is created it occupies the space in the heap. These objects are used by the threads. If the objects are no longer used by the thread it becomes eligible for the garbage collection. The memory occupied by that object is now available for new objects that are being created. It is noted that when the garbage collector destroys the object, the JRE calls the finalize() method to close the connections such as database and network connection.

Private Destructor in C++:

Private Destructor in C++ - GeeksforGeeks
Whenever we want to control the destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling.
Making a destructor private can be used as a technique to prevent the explicit destruction of an object from outside the class. This might be useful in scenarios where you want to control the lifetime of objects, possibly enforcing the use of a specific method to destroy an instance.

Virtual Destructor:

Virtual Destructor - GeeksforGeeks
A virtual destructor is used when you have a base class with virtual functions and you anticipate that objects of derived classes will be deleted through pointers to the base class. A virtual destructor ensures that the appropriate destructor for the actual object type is called during deletion.
When working with polymorphism in C++, it's often a good practice to provide a virtual destructor in base classes if they have virtual functions, to ensure correct destruction of objects of derived classes through base class pointers.

Only Destructors can be Virtual. Constructors cannot be declared as virtual, this is because if you try to override a constructor by declaring it in a base/super class and call it in the derived/subclass with same functionalities it will always give an error as overriding means a feature that lets us to use a method from the parent class in the child class which is not possible.

Pure Virtual Destructor in C++:

A pure virtual function is a function declared in a base class without providing an implementation. A class containing at least one pure virtual function becomes an abstract class,

and you cannot create an instance of an abstract class. Abstract classes are typically used as base classes for other classes that provide concrete implementations for the pure virtual functions.

While you can declare a pure virtual destructor in C++, it is rarely necessary and may lead to complications. The reason is that, by definition, a pure virtual function must be implemented by derived classes to make the class concrete and instantiable. However, the destructor is automatically called when an object is deleted, and requiring derived classes to provide a specific implementation of the destructor is not a common or recommended practice.

Can a destructor be pure virtual in C++?

Yes, it is possible to have a pure virtual destructor. Pure virtual destructors are legal in standard C++ and one of the most important things to remember is that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor.

Why does a pure virtual function require a function body?

The reason is that destructors (unlike other functions) are not actually 'overridden', rather they are always called in the reverse order of the class derivation. This means that a derived class destructor will be invoked first, then the base class destructor will be called. If the definition of the pure virtual destructor is not provided, then what function body will be called during object destruction? Therefore the compiler and linker enforce the existence of a function body for pure virtual destructors.

Object:

An object is an instance of a class, and a class is a blueprint or template for creating objects. Objects are the fundamental building blocks of an object-oriented system, and they encapsulate both data and the methods that operate on that data.

Characteristics of Object:

- **State:** The current values of each attribute continue to make up an object's state. It can be of two types: Static and Dynamic.
- **Behavior:** Behavior describes how an object behaves and responds regarding state transitions.
- **Identity:** An object's identity is a quality that makes it distinct from all other objects.
- **Responsibility:** It is the function of an object that it performs within the system.

Benefits of Using Objects in Programming:

1. **Encapsulation:** One of the key benefits of using objects is encapsulation. Encapsulation refers to the practice of hiding the implementation details of an object from the outside world. This means that the internal workings of an object are hidden from other parts of the program, which can help to reduce complexity and make the code easier to understand. Encapsulation also allows for better control over the object's behavior, as its internal state can only be modified through its methods.
2. **Reusability:** Another benefit of using objects is reusability. Objects can be created once and then reused in multiple parts of the program. This can help to reduce code duplication and make the program more efficient. Additionally, objects can be inherited from other objects, which can further increase reusability and reduce development time.
3. **Modularity:** Objects can also help to improve the modularity of a program. Modularity refers to breaking a program down into smaller, more manageable parts. Objects can represent these smaller parts, each representing a specific aspect of the program's functionality. This can make it easier to develop and maintain the program, as changes to one object will not affect other objects in the program.
4. **Polymorphism:** Polymorphism is another benefit of using objects. Polymorphism refers to the ability of objects to take on different forms. This means that objects can be used in

various contexts without modifying the object itself. For example, a "vehicle" object could represent a car, a truck, or a motorcycle, depending on the context in which it is used.

5. **Code organization:** Finally, objects can help to improve the organization of code. By grouping related data and behavior into objects, the code becomes easier to understand and maintain. Objects can also be organized into hierarchies, each inheriting from a parent object. This can help to organize the code further and make it easier to navigate.

Why do we need objects in OOPs?

Objects are a fundamental concept in object-oriented programming and allow for encapsulation, reusability, modularity, polymorphism, and code organization. By using objects, developers can create more efficient, maintainable, and scalable code that is easier to understand and adapt to changing requirements.

Is it always necessary to create objects from class?

No, it is not always necessary to create objects from a class. However, in object-oriented programming, objects are typically created from classes, as classes define the properties and behaviors of the objects, and only creating an object makes it possible to access the attributes and functions of the class.

What is used to create an object?

In object-oriented programming, objects are created from classes. To create an object, first, a class is defined with the desired properties and behaviors. Then, an instance of the class is created using the "new" keyword, which allocates memory for the object and initializes its properties. The resulting object can then perform various tasks within the program.

Object vs Class:

Object	Class
Object is an instance of a class.	Class is a blueprint or template from which objects are created.
Object is a real world entity such as a pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a group of similar objects.
Object is a physical entity.	Class is a logical entity.
Object is created through new keyword mainly e.g. Student s1=new Student();	Class is declared using class keyword e.g. class Student{}
Object is created many times as per requirement.	Class is declared once.
Object allocates memory when it is created.	Class doesn't allocate memory when it is created.
There are many ways to create objects in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only one way to define a class in java using the class keyword.

Real life example of OOP:

[Real Life Examples of Object Oriented Programming (c-sharpcorner.com)]

Keywords:

static Keyword:

The static keyword is used to define a class member (variable or method) that belongs to the class rather than to instances (objects) of the class. The static keyword indicates that a particular variable or method is associated with the class itself, rather than with instances of the class.

How the static keyword is commonly used:

Static Variables (Class Variables): A static variable is shared among all instances of a class. It is associated with the class rather than with individual objects. There is only one copy of a static variable, regardless of how many instances of the class are created.

```
public class MyClass {
   static int count = 0;        // Static variable (class variable)
   public MyClass() {
      Count++;          // Constructor increments the count for each instance
   }
}
```

Static Methods: A static method belongs to the class rather than to instances of the class. It can be called on the class itself, without creating an instance of the class. Static methods cannot access instance-specific variables directly, as they don't have access to a particular instance's state.

```
public class MathOperations {
   // Static method to add two numbers
   public static int add(int a, int b) {
      return a + b;
   }
}
```

Static Blocks: In Java, a static block is used to initialize static variables or perform other one-time actions when the class is loaded. It is executed only once when the class is loaded into memory.

```
public class MyClass {
   static int x;          // Static variable
   static {       // Static block to initialize the variable
      x = 10;
   }
}
```

Static Nested Classes: In Java, a static nested class is a nested class that is associated with its outer class rather than with an instance of the outer class. It can be instantiated without creating an instance of the outer class.

```
public class OuterClass {
  static class NestedClass {          // Static nested class
      void display() {
         System.out.println("NestedClass display");
      }
   } }
```

The use of the static keyword depends on the specific context and the desired behavior for the members of a class. It is often employed to represent shared resources, utility methods, constants, or one-time initialization logic associated with a class rather than individual instances.

Why is it used?

Static is a Non Access Modifier. It means that something (a field, method, block or nested class) is related to the type rather than any particular instance of the type. Static keyword is used for memory management. The Static keyword can be applied to: Static Method Static Variable

Initialization Block Nested class. Static methods can be called without creating an object of the class. A static variable is one that's associated with a class, not instance (object) of that class. They are initialized only once , at the start of the execution . A single copy to be shared by all instances of the class and it can be accessed directly by the class name and doesn't need any object. The static initializer is a static {} block of code inside a Java class, and runs only one time before the constructor or main method is called. The code block with the static modifier signifies a class initializer; without the static modifier the code block is an instance initializer. A Static Nested Class can't access an enclosing class instance and invoke methods on it, so should be used when the nested class doesn't require access to an instance of the enclosing class . A common use of static nested class is to implement components of the outer object.

virtual Keyword:

The virtual keyword is used to declare a virtual function or to indicate that a function in a base class is intended to be overridden by a function in a derived class. The virtual keyword is primarily associated with achieving polymorphism, which allows objects of different types to be treated as objects of a common base type.
The virtual keyword plays a crucial role in achieving runtime polymorphism in C++. It allows you to write code that can work with objects of different derived types through pointers or references to a common base type, and the appropriate function is dynamically selected at runtime based on the actual type of the object being manipulated.

abstract Keyword:

The abstract keyword is used to define abstract classes, abstract methods, or abstract properties. The concept of abstraction is a fundamental part of object-oriented programming, providing a way to define common characteristics and behaviors without specifying their implementation details.

Characteristics of Java abstract Keyword:

In Java, the abstract keyword is used to define abstract classes and methods. Here are some of its key characteristics:

- **Abstract classes cannot be instantiated:** An abstract class is a class that cannot be instantiated directly. Instead, it is meant to be extended by other classes, which can provide concrete implementations of its abstract methods.
- **Abstract methods do not have a body:** An abstract method is a method that does not have an implementation. It is declared using the abstract keyword and ends with a semicolon instead of a method body. Subclasses of an abstract class must provide a concrete implementation of all abstract methods defined in the parent class.
- **Abstract classes can have both abstract and concrete methods:** Abstract classes can contain both abstract and concrete methods. Concrete methods are implemented in the abstract class itself and can be used by both the abstract class and its subclasses.
- **Abstract classes can have constructors:** Abstract classes can have constructors, which are used to initialize instance variables and perform other initialization tasks. However, because abstract classes cannot be instantiated directly, their constructors are typically called constructors in concrete subclasses.
- **Abstract classes can contain instance variables:** Abstract classes can contain instance variables, which can be used by both the abstract class and its subclasses. Subclasses can access these variables directly, just like any other instance variables.
- **Abstract classes can implement interfaces:** Abstract classes can implement interfaces, which define a set of methods that must be implemented by any class that implements the interface. In this case, the abstract class must provide concrete implementations of all methods defined in the interface.

Abstract Classes:

- Abstract classes may or may not contain abstract methods, i.e., methods without a body (public void get();)
- But, if a class has at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, and provide implementations for the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Abstract Methods:

- The abstract keyword is used to declare the method as abstract.
- You have to place the abstract keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semicolon (;) at the end.
- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Advantages of Abstract Keywords:

1. **Provides a way to define a common interface:** Abstract classes can define a common interface that can be used by all subclasses. By defining common methods and properties, abstract classes provide a way to enforce consistency and maintainability across an application.
2. **Enables polymorphism:** By defining a superclass as abstract, you can create a collection of subclasses that can be treated as instances of the superclass. This allows for greater flexibility and extensibility in your code, as you can add new subclasses without changing the code that uses them.
3. **Encourages code reuse:** Abstract classes can define common methods and properties that can be reused by all subclasses. This saves time and reduces code duplication, which can make your code more efficient and easier to maintain.
4. **Provides a way to enforce implementation:** Abstract methods must be implemented by any concrete subclass of the abstract class. This ensures that certain functionality is implemented consistently across all subclasses, which can prevent errors and improve code quality.
5. **Enables late binding:** By defining a common interface in an abstract class, you can use late binding to determine which subclass to use at runtime. This allows for greater flexibility and adaptability in your code, as you can change the behavior of your program without changing the code itself.

final Keyword:

The final keyword is used to declare entities that cannot be modified. It can be applied to classes, methods, and variables, and its meaning varies depending on where it is used.

Characteristics of final keyword in java:

- **Final variables:**
  - When applied to a variable, the final keyword indicates that the variable's value cannot be changed (it becomes a constant).
  - The variable must be initialized at the time of declaration or in the constructor, and its value cannot be modified afterward.
- **Final methods:**
  - When applied to a method, the final keyword indicates that the method cannot be overridden by subclasses.

    - This is often used to prevent further modification of a method's behavior in a subclass.
- **Final classes:**
    - When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.
- **Initialization:** Final variables must be initialized either at the time of declaration or in the constructor of the class. This ensures that the value of the variable is set and cannot be changed.
- **Performance:** The use of final can sometimes improve performance, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
- **Security:** final can help improve security by preventing malicious code from modifying sensitive data or behavior.
- **Final Parameters:**
    - When applied to a method parameter, the final keyword indicates that the parameter's value cannot be changed within the method.
    - This is a way to ensure that the parameter is treated as a constant within the method.

explicit Keyword:

In C++, the explicit keyword is used as a specifier for constructors and conversion operators. Its primary purpose is to prevent implicit type conversions and improve code safety by making the compiler enforce explicit conversions.

Explicit Constructors: When applied to a constructor, the explicit keyword prevents implicit conversions during object construction. This means that the constructor cannot be called implicitly by the compiler for automatic type conversions.

Explicit Conversion Operators: The explicit keyword can also be applied to conversion operators, preventing them from being used implicitly for automatic type conversions.

this keyword:

It refers to the current object in a method or constructor.
this can also be used to:

- Invoke current class constructor
- Invoke current class method
- Return the current class object
- Pass an argument in the method call
- Pass an argument in the constructor call

Methods to use ‘this’ in Java:

- Using the ‘this’ keyword to refer to current class instance variables.
- Using this() to invoke the current class constructor
- Using ‘this’ keyword to return the current class instance
- Using ‘this’ keyword as the method parameter
- Using ‘this’ keyword to invoke the current class method
- Using ‘this’ keyword as an argument in the constructor call

Advantages of using ‘this’ reference:

1. It helps to distinguish between instance variables and local variables with the same name.
2. It can be used to pass the current object as an argument to another method.
3. It can be used to return the current object from a method.

4. It can be used to invoke a constructor from another overloaded constructor in the same class.

Disadvantages of using 'this' reference:

1. Overuse of this can make the code harder to read and understand.
2. Using this unnecessarily can add unnecessary overhead to the program.
3. Using this in a static context results in a compile-time error.
4. Overall, this keyword is a useful tool for working with objects in Java, but it should be used judiciously and only when necessary.

new Keyword:

In Java, the new keyword acts as a facilitator for the creation of new instances of classes, allowing the developer to initialize objects and allocate memory dynamically during runtime.

What Happens Under the Hood?

When you execute new MyClass(), a series of events unfold within the JVM:

- **Class Loading:** The JVM class loader jumps into action, searching for the MyClass bytecode. If the class is not already part of the runtime, first of all, the class loader reads it. Then it creates a Class object that represents the class definition in the memory.
- **Memory Allocation:** The new keyword triggers the dynamic allocation of memory within the Java Virtual Machine. It facilitates the creation of new objects with their own dedicated memory space. Allocating memory is a two-step process:
    - **Calculation:** The JVM calculates the required amount of memory. The JVM decides it based on the object's properties and the overhead of some housekeeping information.
    - **Allocation:** Then the JVM allocates the calculated space on the heap. Heap is the runtime data area from which memory for all class instances and arrays is allocated.
- **Constructor Execution:** With memory allocated, the JVM calls the constructor of MyClass. This step initializes the object with default or provided values and runs any startup code defined in the constructor.

What About Primitive Types?

It's essential to understand that you don't need the new keyword for creating primitive types like int, char, or boolean. These are not objects but basic data types and are allocated memory in the stack, not the heap.

Memory Management: The Role of Garbage Collection:

Once objects are created, they eventually need to be destroyed. In Java, the garbage collector (GC) handles it. The garbage collector is an automatic process that frees up memory by destroying objects that are no longer in use. When there are no more references to an object, the GC considers it eligible for garbage collection.

The new Keyword and Arrays:

Arrays in Java are also objects. To create an array, you also use the new keyword:

```
int[] numbers = new int[5];
```

This line tells the JVM to allocate memory for an array of int with five elements.

Behind the Scenes with Arrays:

Creating an array with new triggers a slightly different process:

- **Memory Calculation:** The JVM calculates the size needed based on the array type and length.

- **Memory Allocation:** The JVM allocates memory for the array, and initializes each element with the default value for the type (e.g., 0 for int, null for object references).

Const Keyword:

Const keyword in C++ - javatpoint

Const keyword is used to define the constant value that cannot change during program execution. It means once we declare a variable as the constant in a program, the variable's value will be fixed and never be changed. If we try to change the value of the const type variable, it shows an error message in the program.

There are a certain set of rules for the declaration and initialization of the constant variables:

- The const variable cannot be left un-initialized at the time of the assignment.
- It cannot be assigned value anywhere in the program.
- Explicit value needed to be provided to the constant variable at the time of declaration of the constant variable.

Java's 'final' vs. 'const' keywords:

- final Keyword in Java:
  - The final keyword is commonly used to declare constants in Java. It ensures that the value of the variable cannot be changed once it has been assigned. Constants are often declared in uppercase with underscores separating words (e.g., MAX_VALUE).
  - When applied to variables, the final keyword indicates that the variable cannot be reassigned after it has been initialized. This is commonly used for method parameters to make them effectively final.
  - When applied to methods, the final keyword prevents the method from being overridden in subclasses.
  - When applied to classes, the final keyword indicates that the class cannot be subclassed.
- No const Keyword in Java:
  - Unlike some other programming languages (e.g., C++), Java does not have a const keyword. The final keyword is used for similar purposes, including declaring constants and making variables or methods non-modifiable. The final keyword in Java serves a broader range of purposes than the const keyword in some other languages.

Why is 'const' not implemented in Java?

The absence of a const keyword in Java is a design choice based on the language's principles and goals. While other languages, such as C++ or C#, have a const keyword, Java has chosen a different approach to achieve similar goals. Here are some reasons why Java doesn't have a const keyword:

1. **Immutable Objects:** In Java, the concept of immutability is often used instead of const. An immutable object is an object whose state cannot be changed after it is created. String and wrapper classes (e.g., Integer, Double) in Java are examples of immutable objects.
   ```
   String str = "Hello";
   str = str.concat(" World");  // Creates a new string, doesn't modify the original
   ```
   By promoting immutability, Java achieves a similar effect to the const keyword, ensuring that certain objects or values cannot be modified once created.
2. **Final Keyword:** The final keyword in Java is versatile and serves the purpose of declaring constants, marking variables as non-modifiable, preventing method overrides, and making classes non-subclassable. The final keyword covers a wide range of use cases, eliminating the need for a separate const keyword.

3. **Simplicity and Consistency:** Java strives for simplicity and consistency in its design. By using final for various scenarios, the language maintains a unified and straightforward approach without introducing additional keywords that might add complexity.
4. **Backward Compatibility:** Introducing a new keyword like const could have implications for backward compatibility. Existing codebases might be using identifiers that happen to be named const, and introducing a new keyword could break such code.
5. **Philosophy of Java:** Java's design philosophy emphasizes readability, simplicity, and avoiding unnecessary complexity. Introducing a const keyword may have been considered unnecessary given that existing language features (especially final and immutability) cover the required functionality.

Super Keyword:

Super Keyword in Java - GeeksforGeeks

In Java, the super keyword is used to refer to the immediate parent class object. It is often used to invoke the superclass's methods, access superclass fields, and call the superclass constructor. The super keyword is particularly useful in scenarios involving method overriding and constructor chaining in inheritance.

Characteristics of Super Keyword in Java:

In Java, super keyword is used to refer to the parent class of a subclass. Here are some of its key characteristics:

1. **super is used to call a superclass constructor:** When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the constructor of the parent class.
2. **super is used to call a superclass method:** A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class's implementation of the method in addition to its own.
3. **super is used to access a superclass field:** A subclass can access a field defined in its parent class using the super keyword. This is useful when the subclass wants to reference the parent class's version of a field.
4. **super must be the first statement in a constructor:** When calling a superclass constructor, the super() statement must be the first statement in the constructor of the subclass.
5. **super cannot be used in a static context:** The super keyword cannot be used in a static context, such as in a static method or a static variable initializer.
6. **super is not required to call a superclass method:** While it is possible to use the super keyword to call a method in the parent class, it is not required. If a method is not overridden in the subclass, then calling it without the super keyword will invoke the parent class's implementation.

- **Use of super with Variables:** This scenario occurs when a derived class and base class have the same data members. In that case, there is a possibility of ambiguity in the JVM.

```
class Vehicle {          // Base class vehicle
   int maxSpeed = 120;
}
class Car extends Vehicle { // subclass Car extending vehicle
   int maxSpeed = 180;
   void display()
   {
      System.out.println("Maximum Speed: " + super.maxSpeed);
   }
```

```
}
class Test {
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

o/p: Maximum Speed: 120

In the above example, both the base class and subclass have a member maxSpeed. We could access the maxSpeed of the base class in subclass using super keyword.

- **Use of super with Methods:** This is used when we want to call the parent class method. So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword.

```
class Person {          // superclass Person
   void message()
   {
      System.out.println("This is person class\n");
   }
}
class Student extends Person {     // Subclass Student
   void message()
   {
      System.out.println("This is student class");
   }
   void display()
   {
      message();
      super.message();
   } }
class Test {
   public static void main(String args[])
   {
      Student s = new Student();
       s.display();
   }
}
```

o/p: This is student class This is person class

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of the super keyword, message() of the superclass could also be invoked.

- **Use of super with constructors:** The super keyword can also be used to access the parent class constructor. One more important thing is that 'super' can call both parametric as well as non-parametric constructors depending on the situation.

```
class Person {          // superclass Person
   Person()
   {
      System.out.println("Person class Constructor");
   }
}
class Student extends Person {     // subclass Student extending the Person class
```

```
    Student()
    {
        super();
        System.out.println("Student class Constructor");
    }
}
class Test {
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

o/p: Person class Constructor Student class Constructor

In the above example, we have called the superclass constructor using the keyword 'super' via subclass constructor.

Advantages of Using Java Super Keyword:

1. **Enables reuse of code:** Using the super keyword allows subclasses to inherit functionality from their parent classes, which promotes the reuse of code and reduces duplication.
2. **Supports polymorphism:** Because subclasses can override methods and access fields from their parent classes using super, polymorphism is possible. This allows for more flexible and extensible code.
3. **Provides access to parent class behavior:** Subclasses can access and use methods and fields defined in their parent classes through the super keyword, which allows them to take advantage of existing behavior without having to reimplement it.
4. **Allows for customization of behavior:** By overriding methods and using super to call the parent implementation, subclasses can customize and extend the behavior of their parent classes.
5. **Facilitates abstraction and encapsulation:** The use of super promotes encapsulation and abstraction by allowing subclasses to focus on their behavior while relying on the parent class to handle lower-level details.

Polymorphism:

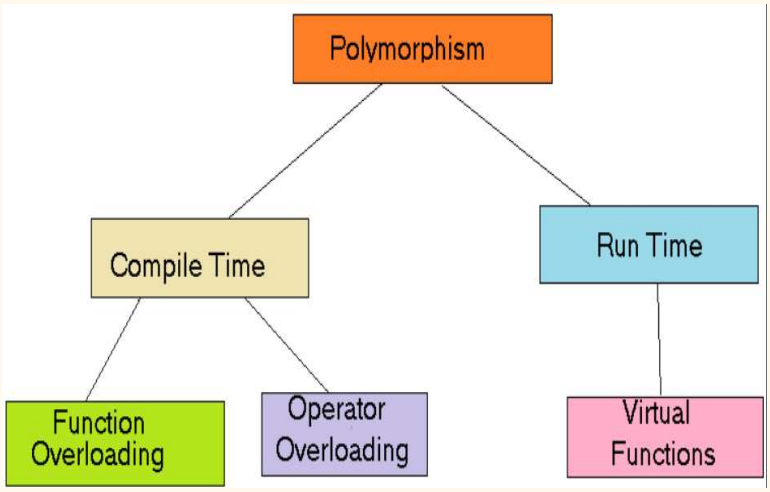
Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows us to perform a single action in different ways. In other words, polymorphism allows us to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms. In Java, polymorphism is primarily achieved through two mechanisms: method overriding (runtime polymorphism) and method overloading (compile-time polymorphism).

Needs of Polymorphism:

Here are some key reasons to use polymorphism:

1. **Code Reusability:** Polymorphism allows for the creation of code that can work with objects of different types through a common interface. By designing classes to adhere to common interfaces or base classes, you can write code that is more reusable across different parts of your application.
2. **Flexibility and Extensibility:** Polymorphism provides flexibility in handling different types of objects without the need to modify existing code. New classes can be added without affecting the existing codebase, as long as they adhere to the same interface or inherit from the same base class. It allows for easy extension and addition of new functionality.

3. **Reduced Code Redundancy:** Polymorphism helps eliminate redundant code by allowing you to write generic code that can work with various types of objects. Instead of writing separate code for each specific type, you can design code that works with a common interface, reducing duplication.
4. **Method Overriding:** Method overriding, a form of polymorphism, allows subclasses to provide specific implementations of methods defined in their superclass. This enables customization of behavior in subclasses while maintaining a consistent interface.
5. **Simplified Code Maintenance:** Polymorphism contributes to easier code maintenance. When changes are required, modifications can often be localized to specific classes without affecting the entire system. This makes it easier to understand, update, and debug code.
6. **Improved Readability:** Code that uses polymorphism is often more readable and intuitive. It follows the principles of abstraction and encapsulation, focusing on what objects can do rather than their internal details. This makes the code more understandable and expressive.
7. **Dynamic Binding (Runtime Polymorphism):** In languages like Java, polymorphism provides dynamic binding, where the appropriate method implementation is selected at runtime based on the actual type of the object. This allows for the creation of flexible and extensible systems where the behavior can be determined dynamically.
8. **Adherence to Design Principles:** Polymorphism supports principles of OOP, such as inheritance, encapsulation, and abstraction. It helps create code that is modular, and adheres to the principles of object-oriented design.



Compile-time polymorphism:
Compile-time polymorphism in Java is achieved through method overloading. Method overloading allows a class to have multiple methods with the same name but different parameter lists (different numbers or types of parameters). The appropriate method to be executed is determined by the compiler at compile time based on the method signature.

Key points about compile-time polymorphism:

- The selection of the method to be called is determined by the compiler at compile time.
- It is also known as static polymorphism or method overloading.
- Method overloading is achieved by having multiple methods with the same name but different parameter lists.
- The return type alone is not sufficient to differentiate overloaded methods; the parameter types or the number of parameters must be different.

Subtypes of Compile-time Polymorphism:

1. **Function Overloading:** It is a feature in C++ where multiple functions can have the same name but with different parameter lists. The compiler will decide which function to call based on the number and types of arguments passed to the function.
    a. The functions can be overloaded by using a different number of arguments and by using different types of arguments.
    b. If two same name and same argument functions just vary in their return type then such function isn't overloaded.

2. **Operator Overloading:** It is a feature in C++ where the operators such as +, -, *, etc. can be given additional meanings when applied to user-defined data types.
    a. Precedence and associativity remain intact in operators.
    b. List of operators that cannot be overloaded in C++ are ::, .*, ., ?:
    c. Operators = and & are already overloaded in C++, so we should avoid overloading them.
3. **Template:** It is a powerful feature in C++ that allows us to write generic functions and classes. A template is a blueprint for creating a family of functions or classes.

Runtime Polymorphism:

In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type. It is also known as Dynamic Polymorphism because the function calls are dynamically bonded at the runtime.

Run Time Polymorphism can be exhibited by: Method Overriding using Virtual Functions

- **Method Overriding:** Method overriding refers to the process of creating a new definition of a function in a derived class that is already defined inside its base class. Some rules that must be followed while overriding a method are:
    - Method names must be the same.
    - Method parameters must be the same.
- **Virtual Function:** Virtual Function is a member function that is declared as virtual in the base class and it can be overridden in the derived classes that inherit the base class.
- Virtual functions are generally declared in the base class and are typically defined in both the base and derived classes.

Compile-time polymorphism vs Run-time polymorphism:

Compile-Time Polymorphism	Run-Time Polymorphism
It is also called Static Polymorphism.	It is also known as Dynamic Polymorphism.
In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.	In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type.
Function calls are statically binded.	Function calls are dynamically binded.
Compile-time Polymorphism can be exhibited by: Function Overloading & Operator overloading.	Run-time Polymorphism can be exhibited by Function Overriding.
Faster execution rate.	Comparatively slower execution rate.
Inheritance is not involved.	Involves inheritance.

Inheritance:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherit the properties and behaviors of another class. In Java, inheritance is achieved through

the use of the extends keyword. The new class that is created is known as subclass (child or derived class) and the existing class from which the child class is derived is known as superclass (parent or base class).

- Constructor cannot be inherited in Java.
- Private members do not get inherited in Java.
- Cyclic inheritance is not permitted in Java.
- Assign parent reference to child objects.
- Constructors get executed because of super() present in the constructor.

Why Do We Need Java Inheritance?

Imagine, as a car manufacturer, you offer multiple car models to your customers. Even though different car models might offer different features like a sunroof or bulletproof windows, they would all include common components and features, like engine and wheels.

It makes sense to create a basic design and extend it to create their specialized versions, rather than designing each car model separately, from scratch.

In a similar manner, with inheritance, we can create a class with basic features and behavior and create its specialized versions, by creating classes that inherit this base class. In the same way, interfaces can extend existing interfaces.

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

Can Object-oriented programming exist without inheritance?

Yes, object-oriented programming (OOP) can exist and be practiced without using inheritance. While inheritance is a fundamental concept in OOP, it is just one of several features that define the paradigm. While inheritance is powerful and widely used in OOP, some programming languages or specific projects may choose not to emphasize or use it extensively. The other three principles—encapsulation, abstraction, and polymorphism—can still be applied to create object-oriented code.

In some cases, developers may favor composition over inheritance. Composition involves creating objects by combining simpler objects, rather than relying on the hierarchy of class inheritance. This approach can lead to more flexible and modular code.

Single Inheritance:

In single inheritance, only one class is derived from the parent class. In this type of inheritance, the properties are derived from a single parent class and not more than that.

Multilevel Inheritance:

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.

Hierarchical Inheritance:

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.

Multiple Inheritance:

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces.

Hybrid Inheritance:

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritance with classes, hybrid inheritance involving multiple inheritance is also not possible

with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.

Why is there no multiple inheritance in Java, but implementing multiple interfaces is allowed?

Java supports multiple inheritance through interfaces but not through classes. The decision to exclude multiple inheritance for classes in Java was made to address certain issues and complexities associated with it, while still allowing the benefits of multiple inheritance through interfaces.

Here are some reasons:

1. **Diamond Problem:**
    a. Multiple inheritance with classes can lead to the "diamond problem," where a class inherits from two classes that have a common ancestor. If there are conflicting methods or fields in the two parent classes, it becomes ambiguous which one should be inherited by the child class.
    b. Java aims to prevent such ambiguity and complexities that arise from the diamond problem.
2. **Simplicity and Readability:**
    a. Java places a strong emphasis on simplicity, readability, and ease of understanding code. Multiple inheritance with classes can lead to intricate relationships and potential conflicts between methods and fields from different parent classes, making the code more challenging to comprehend and maintain.
3. **Avoiding Ambiguity:**
    a. Java's designers made a deliberate choice to avoid the complications associated with multiple inheritance in class hierarchies. Instead, the language encourages developers to favor composition and interfaces to achieve flexibility and modularity.
4. **Interfaces for Contract-based Programming:**
    a. Java provides interfaces as a means of achieving multiple inheritance in a controlled and contract-based manner. Interfaces define contracts that classes can implement, allowing a single class to implement multiple interfaces without the risk of conflicts.
5. **Flexibility through Composition:**
    a. Java promotes the use of composition over inheritance, encouraging developers to create classes by composing simpler objects rather than relying on complex class hierarchies. This approach leads to more modular and maintainable code. Composition can lead to more modular and flexible designs, as objects are composed of other objects, making it easier to understand and maintain.
6. **Preventing Fragile Base Class Problem:**
    a. Multiple inheritance with classes can contribute to the "fragile base class" problem, where changes to a base class can have unintended consequences on derived classes. Java's design aims to reduce the impact of such problems by limiting class inheritance.
7. **Interface Segregation Principle (ISP):**
    a. Java adheres to the Interface Segregation Principle, one of the SOLID principles of object-oriented design. This principle suggests that a class should not be forced to implement interfaces it does not use. Interfaces provide a way to avoid unnecessary dependencies.

Interfaces for Multiple Inheritance: While Java does not allow multiple inheritance with classes, it provides support for multiple inheritance through interfaces. Interfaces in Java allow a class to implement multiple interfaces, enabling a form of multiple inheritance without the issues associated with class inheritance.

```
interface A {
  void methodA();
```

```
}
interface B {
   void methodB();
}
class MyClass implements A, B {
   public void methodA() {
      // Implementation for methodA
   }
   public void methodB() {
      // Implementation for methodB
   }
}
```

In this example, MyClass implements both interfaces A and B, effectively inheriting from both. The use of interfaces avoids the complexities and ambiguities associated with multiple inheritance in class hierarchies.

In summary, Java's decision to exclude multiple inheritance with classes helps avoid certain pitfalls and complexities, while still providing a form of multiple inheritance through interfaces. This design choice aligns with Java's goal of simplicity, readability, and ease of maintenance.

How c++ overcome these problems and support multiple inheritance?

C++ supports multiple inheritance, allowing a class to inherit from multiple classes. However, this feature introduces challenges and potential issues, including the "diamond problem." C++ provides mechanisms to address these challenges. Here are key features and approaches in C++ to support multiple inheritance:

1. **Virtual Inheritance:** In C++, virtual inheritance is a feature that helps address the diamond problem. By using the virtual keyword during inheritance, a class can inherit from a common base class only once, even if it appears in multiple paths through the inheritance hierarchy.

```
class Animal {
public:
   void eat() {
      std::cout << "Animal is eating" << std::endl;
   }
};
class Mammal : virtual public Animal {
public:
   void breathe() {
      std::cout << "Mammal is breathing" << std::endl;
   }
};
class Bird : virtual public Animal {
public:
   void fly() {
      std::cout << "Bird is flying" << std::endl;
   }
};
class Bat : public Mammal, public Bird {
Public:          // Bat inherits from both Mammal and Bird through virtual inheritance
};
int main() {
   Bat bat;
```

```
    bat.eat();   // Accessing eat() from Animal
    bat.breathe();// Accessing breathe() from Mammal
    bat.fly();   // Accessing fly() from Bird

    return 0;
}
```

By using virtual inheritance, the common base class (Animal in this case) is shared among the multiple paths, preventing the creation of duplicate instances of the base class.

2. **Access Control Modifiers:** C++ allows classes to specify access control for the base classes during multiple inheritance. Access specifiers (public, protected, and private) can be used to control how the members of the base classes are accessible in the derived class.

```
class BaseA {
public:
    void methodA() {}
};
class BaseB {
public:
    void methodB() {}
};
class Derived : public BaseA, private BaseB {
Public:          // Now methodA() is public in Derived, but methodB() is private
};
```

3. **Function Overriding:** C++ allows derived classes to override functions inherited from multiple base classes. This feature enables customization of behavior in the derived class.

```
class BaseA {
public:
    virtual void method() {
        std::cout << "BaseA method" << std::endl;
    }
};
class BaseB {
public:
    virtual void method() {
        std::cout << "BaseB method" << std::endl;
    }
};
class Derived : public BaseA, public BaseB {
public:
    void method() override {
        std::cout << "Derived method" << std::endl;
    }
};
```

In this example, Derived provides its own implementation of the method() function, overriding the versions in both BaseA and BaseB.

4. **Mixins and Composition:** C++ also supports mixins and composition as alternatives to traditional multiple inheritance. Mixins involve creating small, reusable components that can be combined to form a class, while composition involves creating objects by combining simpler objects.

```
class LoggerMixin {
public:
    void log(const std::string& message) {
```

```
        std::cout << "Log: " << message << std::endl;
    }
};
class DataSource {
public:
    virtual int getData() = 0;
};
class DataProcessor : public DataSource, public LoggerMixin {
public:
    int getData() override {
        int data = /* obtain data */;
        log("Data processed");
        return data;
    }
};
```

In this example, LoggerMixin is a mixin providing logging functionality, and DataProcessor combines this mixin with its base class (DataSource) through multiple inheritance.

Limitations of Inheritance:

1. **Increased Coupling:** Inheritance can lead to increased coupling between classes. Subclasses are dependent on the implementation details of their parent classes. Changes to the implementation of a superclass can affect its subclasses, creating tight dependencies.
2. **Fragile Base Class Problem:** The "fragile base class" problem occurs when changes to a base class (e.g., modifications or additions to methods) can unintentionally affect the behavior of derived classes. This makes the system more prone to errors and introduces maintenance challenges.
3. **Hierarchy Maintenance:** As a software system evolves, maintaining and updating the class hierarchy can become complex. Adding or modifying classes in the hierarchy may require changes throughout the system, leading to increased development effort.
4. **Rigidity:** Inheritance can make a system rigid and less adaptable to changes. Subclasses are tightly bound to their parent classes, and altering the structure of the inheritance hierarchy may be challenging without affecting existing code.
5. **Limited Code Reusability:** While inheritance promotes code reuse, it may not always lead to optimal reuse. Subclasses inherit the entire interface and implementation of their parent classes, including methods that may not be relevant to their context.
6. **Complexity:** Excessive use of inheritance can introduce unnecessary complexity. Deep and complex class hierarchies may be difficult to understand, making the codebase harder to maintain and debug.
7. **Difficulty in Testing:** Testing derived classes can be challenging because they inherit behavior from their parent classes. Understanding and testing all the inherited behavior, especially in the presence of multiple levels of inheritance, can be complex.
8. **Ambiguity and Diamond Problem:** Multiple inheritance with classes can lead to ambiguity and the "diamond problem." When a class inherits from.

How can we call the base method without creating an instance?

In Java, you cannot directly call a non-static base method without creating an instance of the class. Non-static methods in Java belong to instances of the class, and you need an object to invoke these methods.

new vs override:

new:

The new keyword in Java is used to create an instance of a class. It is used in the context of object creation. When you use new, you are allocating memory for an object, invoking its constructor, and obtaining a reference to the newly created instance.
@Override:
The @Override annotation in Java is used to indicate that a method in a subclass is intended to override a method with the same signature in its superclass. It is a helpful annotation for both developers and the compiler to catch errors related to method overriding.

Object slicing in Java:

Object slicing is a term that is commonly associated with C++ and refers to a situation where an object of a derived class is assigned to an object of its base class, resulting in the loss of information specific to the derived class. This phenomenon is inherent to C++ due to its value semantics and object representation in memory.

However, in Java, object slicing as described in C++ doesn't occur because Java uses reference semantics and objects are always accessed through references. When you assign an object of a subclass to a reference variable of its superclass, you are not dealing with the actual object but with a reference to that object.

Data Hiding:

Data Hiding is hiding internal data from outside users. The internal data should not go directly, that is outside person/classes are not able to access internal data directly. It is achieved by using an access specifier- a private modifier.

How to hide base class methods/ functions?

In Java, you cannot directly hide methods of a base class in the same way you might use the private access modifier in C++ to hide base class methods from derived classes. In Java, methods in the base class with private access are not visible to subclasses, but they are not overridden; they are simply not accessible.
However, if you want to override a method in the derived class and "hide" the base class method, you can use the @Override annotation to ensure that you are indeed overriding a method from the base class. This practice is not exactly hiding the base class method, but it's a common way to indicate the intent to override.

Inheritance vs Polymorphism:

Inheritance	Polymorphism
Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class).	Whereas polymorphism is that which can be defined in multiple forms.
It is basically applied to classes.	Whereas it is basically applied to functions or methods.
Inheritance supports the concept of reusability and reduces code length in object-oriented programming.	Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).

Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance.	Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding).
It is used in pattern designing.	While it is also used in pattern designing.
Example : The class bike can be inherited from the class of two-wheel vehicles, which in turn could be a subclass of vehicles.	Example : The class bike can have the method name set_color(), which changes the bike's color based on the name of color you have entered.

Association, Composition and Aggregation in Java:

Association, Composition and Aggregation in Java - GeeksforGeeks

- **Association:**
  Association represents a relationship between two or more classes. It can be a simple, bi-directional connection where one class knows about another, or it can involve more complex relationships. Associations can be one-to-one, one-to-many, or many-to-many.
- **Aggregation:**
  Aggregation is a more specific form of association representing a "whole-part" relationship. It implies a weaker relationship where one class (the whole) has a collection of objects (parts), but the parts can exist independently of the whole.
- **Composition:**
  Composition is a stronger form of aggregation, indicating a "whole-part" relationship where the parts are strongly dependent on the whole. In composition, the lifetime of the parts is managed by the whole, and when the whole is destroyed, the parts are also destroyed.

Encapsulation:

Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

How is encapsulation achieved in Java?

In Java, encapsulation is achieved through the use of access modifiers (public, private, protected, and default/package-private) to control the visibility of class members (fields and methods). Here are key aspects of encapsulation in Java:

1. **Private Access Modifier:** Declaring a field or method as private restricts its access to only within the same class. It is not visible to other classes.

```
public class MyClass {
   private int privateField;

   private void privateMethod() {
      // Implementation details
   }
}
```

2. **Public Access Modifier:** Declaring a field or method as public makes it accessible from any other class. It is part of the public interface of the class.

```
public class MyClass {
```

```
    public int publicField;

    public void publicMethod() {
        // Implementation details
    }
}
```

3. **Protected Access Modifier:** Declaring a field or method as protected allows access within the same package and by subclasses (even if they are in different packages).

```
public class MyBaseClass {
    protected int protectedField;

    protected void protectedMethod() {
        // Implementation details
    }
}
```

4. **Default (Package-Private) Access Modifier:** If no access modifier is specified (default), it is accessible only within the same package.

```
class PackagePrivateClass {
    int packagePrivateField;

    void packagePrivateMethod() {
        // Implementation details
    }
}
```

5. **Getter and Setter Methods:** Encapsulation often involves providing getter and setter methods to access and modify private fields, allowing controlled access.

```
public class MyClass {
    private int value;

    public int getValue() {
        return value;
    }

    public void setValue(int newValue) {
        this.value = newValue;
    }
}
```

Advantages of Encapsulation:

1. **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
2. **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
3. **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.

4. **Testing code is easy:** Encapsulated code is easy to test for unit testing.
5. **Freedom to programmers in implementing the details of the system:** This is one of the major advantages of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Disadvantages of Encapsulation in Java:

1. Can lead to increased complexity, especially if not used properly.
2. Can make it more difficult to understand how the system works.
3. May limit the flexibility of the implementation.

Data Hiding:

In Java, data hiding refers to the practice of encapsulating the internal details of a class and restricting direct access to its internal data (fields). This is achieved through the use of access modifiers to control the visibility of class members. The main idea is to hide the implementation details of a class and provide controlled access to the external world.

Encapsulation vs Data Hiding:

Encapsulation encompasses Data Hiding: Data hiding is a specific technique used within the broader concept of encapsulation. While encapsulation involves grouping data and methods together, data hiding specifically addresses controlling access to the data.
Encapsulation is a Design Principle: Encapsulation is a design principle that promotes organizing code in a way that enhances modularity, maintainability, and flexibility. It involves creating classes that encapsulate related functionalities.
Data Hiding is an Implementation Detail: Data hiding is one of the strategies employed to achieve encapsulation. It is concerned with the visibility of data members and the ways in which external code can interact with them.

In summary, encapsulation is a holistic design approach that involves bundling data and methods together, while data hiding is a specific strategy within encapsulation that focuses on controlling access to the internal data of a class.

How does data hiding relate to the broader principles of encapsulation and abstraction in Java?

Data hiding, encapsulation, and abstraction are interconnected principles in Java. Data hiding involves controlling access to class members, encapsulation involves bundling data and methods into a single unit, and abstraction involves presenting only essential information while hiding implementation details. Together, these principles facilitate the creation of modular, secure, and easily maintainable Java code.

Abstraction:

Abstraction refers to the practice of hiding implementation details and providing a simplified view of a system to its users. It is used to simplify complex systems by exposing only the necessary features and behaviors, while hiding the underlying complexity.

Abstraction is important because it helps to simplify code, making it easier to use, maintain and extend. By providing a simplified view of a system, abstraction helps to encapsulate complexity, making it easier to manage and work with. Abstraction also promotes code reuse, as abstract classes and interfaces can be used as templates for creating new classes with similar behavior. In Java, abstraction is implemented through two primary mechanisms: abstract classes and interfaces.

Types of abstraction:

1. **Data Abstraction:** Data abstraction is the way to create complex data types and expose only meaningful operations to interact with the data type while hiding all the implementation details from outside works.

The benefit of this approach involves the capability of improving the implementation over time e.g. solving performance issues if any. The idea is that such changes are not supposed to impact client code since they involve no difference in abstract behavior.

2. **Control Abstraction:** Any software consists of numerous statements written in any programming language. Most of the time, statements are similar and repeated over places multiple times.

   Control abstraction is the process of identifying all such statements and exposing them as a unit of work. We normally use this feature when we create a function to perform any work.

How to Achieve Abstraction in Java?

As abstraction is one of the core principles of Object-oriented programming practices and Java follows all OOPs principles, abstraction is one of the major building blocks of the Java language. In Java, abstraction is achieved by interfaces and abstract classes. Interfaces allow you to abstract the implementation completely, while abstract classes allow partial abstraction as well.

- Data abstraction spans from creating simple data objects to complex collection implementations such as HashMap or HashSet.
- Similarly, control abstraction can be seen from defining simple function calls to complete open-source frameworks. Control abstraction is the main force behind structured programming.

Abstract Classes vs. Interfaces:

Abstract classes:

- Can have both abstract and non-abstract methods.
- Can have fields (including non-static and non-final fields).
- Support constructors.
- Can extend only one class.

Interfaces:

- Can only have abstract methods (prior to Java 8).
- Can have constant fields (implicitly public, static, and final).
- Do not support constructors (prior to Java 8).
- Can extend multiple interfaces.

Java Abstract classes and Java Abstract methods:

1. An abstract class is a class that is declared with an abstract keyword.
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

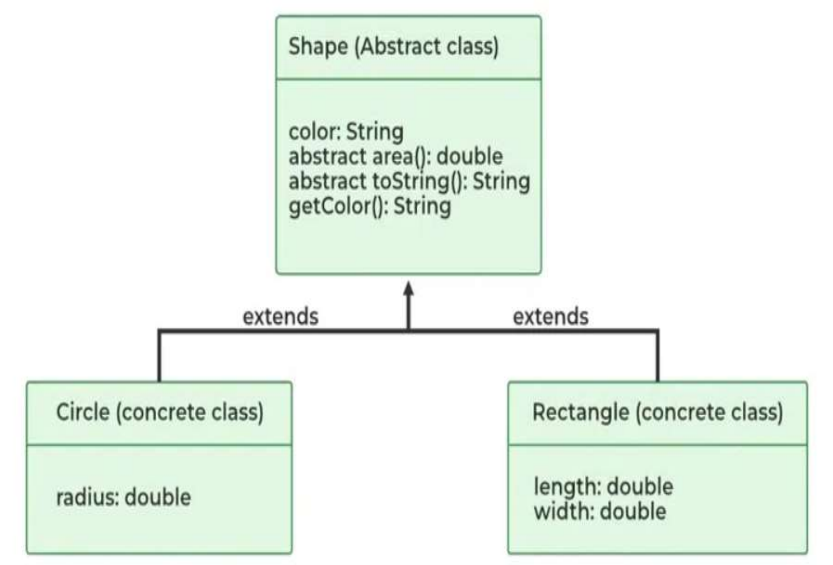
Algorithm to implement abstraction in Java:

1. Determine the classes or interfaces that will be part of the abstraction.
2. Create an abstract class or interface that defines the common behaviors and properties of these classes.

3. Define abstract methods within the abstract class or interface that do not have any implementation details.
4. Implement concrete classes that extend the abstract class or implement the interface.
5. Override the abstract methods in the concrete classes to provide their specific implementations.
6. Use the concrete classes to implement the program logic.

When to use abstract classes and abstract methods?

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.



Consider a classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape" and each shape has a color, size, and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle, and so on — each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.

Benefits of Abstraction in Java:

1. **Code Reusability:** Abstraction allows for the creation of reusable components (abstract classes or interfaces) that can be used in various contexts.
2. **Flexibility:** Abstraction provides a way to define a common interface without specifying the details, allowing for flexibility in implementation.
3. **Encapsulation:** Abstraction often goes hand-in-hand with encapsulation, as it involves hiding the internal details and exposing only what is necessary.
4. **Polymorphism:** Abstraction supports polymorphism, allowing objects to be treated uniformly through a common interface.
5. **Maintenance:** Changes to the internal implementation of a class (abstract or concrete) do not affect the external code that uses the abstraction.

Advantages of Abstraction:

1. It reduces the complexity of viewing things.
2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only essential details are provided to the user.
4. It improves the maintainability of the application.
5. It improves the modularity of the application.
6. The enhancement will become very easy because without affecting end-users we can able to perform any type of changes in our internal system.
7. Improves code reusability and maintainability.
8. Hides implementation details and exposes only relevant information.

9. Provides a clear and simple interface to the user.
10. Increases security by preventing access to internal class details.
11. Supports modularity, as complex systems can be divided into smaller and more manageable parts.
12. Abstraction provides a way to hide the complexity of implementation details from the user, making it easier to understand and use.
13. Abstraction allows for flexibility in the implementation of a program, as changes to the underlying implementation details can be made without affecting the user-facing interface.
14. Abstraction enables modularity and separation of concerns, making code more maintainable and easier to debug.

Disadvantages of Abstraction in Java:

1. Here are the main disadvantages of abstraction in Java:
2. Abstraction can make it more difficult to understand how the system works.
3. It can lead to increased complexity, especially if not used properly.
4. This may limit the flexibility of the implementation.
5. Abstraction can add unnecessary complexity to code if not used appropriately, leading to increased development time and effort.
6. Abstraction can make it harder to debug and understand code, particularly for those unfamiliar with the abstraction layers and implementation details.
7. Overuse of abstraction can result in decreased performance due to the additional layers of code and indirection.

Abstraction vs Interface:

Abstract class	Interface
1) Abstract classes can have abstract and non-abstract methods.	Interfaces can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interfaces can't provide the implementation of abstract classes.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare the interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using the keyword "extends".	An interface can be implemented using the keyword "implements".
8) A Java abstract class can have class members like private, protected etc.	Members of a Java interface are public by default.

9)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Abstraction vs Encapsulation:

Abstraction	Encapsulation
Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
Abstraction is the method of hiding unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using an access modifier i.e. private, protected and public.
In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.
Abstraction provides access to specific parts of data.	Encapsulation hides data and the user can not access the same directly (data hiding.
Abstraction focuses on “what” should be done.	Encapsulation focus is on “How” it should be done.

Why can't we create an object (instance) of an abstract class in Java?

1. Abstract class is not a complete class, it means abstract class contains only method declaration and variable declaration.
2. While creating objects we don't know what the behavior of class and Compilers prevent you from instantiating such incomplete objects.
3. If we create an instance of class and call the method (which doesn't have a method body) the compiler throws an error.

4. An abstract class has a protected constructor (by default) allowing derived types to initialize it.
5. The abstract class is designed to create one abstract idea which can be implemented in sub classes.

If we try to create an instance for the above class compiler will be confused as to how much memory should be created for class as there is no any method implementation available hence it will throw the compile time error.

Static Binding and Dynamic Binding:

Static Binding (Early Binding): Static binding, also known as early binding, occurs during compile-time. In statically bound languages like Java, the binding of a method call to its definition happens at compile-time. The compiler determines the appropriate method or function to be called based on the type of the reference used to invoke the method. This type of binding is also referred to as compile-time binding or method overloading.

Dynamic Binding (Late Binding): Dynamic binding, also known as late binding or runtime polymorphism, occurs during runtime. In dynamically bound languages like Java, the binding of a method call to its definition happens at runtime. The determination of the method or function to be called is based on the actual type of the object. This type of binding is also referred to as runtime binding or method overriding.

Static Binding	Dynamic Binding
It takes place at compile time for which is referred to as early binding	It takes place at runtime so it is referred to as late binding.
It uses overloading more precisely operator overloading method	It uses overriding methods.
It takes place using normal functions	It takes place using virtual functions
Static or const or private functions use real objects in static binding	Real objects use dynamic binding.

Message Passing:

Message passing is a concept used in communication between objects in object-oriented programming (OOP). In an object-oriented system, objects communicate and interact with each other by sending messages. This communication allows objects to invoke methods, request services, and exchange information. Message passing is a fundamental mechanism that enables the implementation of encapsulation, polymorphism, and abstraction.
Here are key aspects of message passing:

- **Object Communication:**
  - In OOP, objects are instances of classes, and communication between objects is achieved through message passing.
  - Objects interact by sending messages to each other, invoking methods or requesting services.
- **Methods and Messages:**
  - When one object wants another object to perform an action or provide information, it sends a message to the target object.

    - The message consists of the name of the method to be invoked and any required parameters.
- **Encapsulation:**
    - Message passing supports encapsulation by allowing objects to hide their internal details and expose a well-defined interface.
    - Objects communicate through the public methods provided by the class.
- **Polymorphism:**
    - Message passing enables polymorphism, where objects of different classes can respond to the same message (method call) in a way that is appropriate for their type.
    - Polymorphism allows for flexibility and extensibility in the design of object-oriented systems.

Procedural vs Object Oriented Programming:

Procedural Oriented Programming	Object-Oriented Programming
In procedural programming, the program is divided into small parts called functions.	In object-oriented programming, the program is divided into small parts called objects.
Procedural programming follows a top-down approach.	Object-oriented programming follows a bottom-up approach.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and functions is easy.
Procedural programming does not have any proper way of hiding data so it is less secure.	Object-oriented programming provides data hiding so it is more secure.
In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.
In procedural programming, there is no concept of data hiding and inheritance.	In object-oriented programming, the concept of data hiding and inheritance is used.
In procedural programming, the function is more important than the data.	In object-oriented programming, data is more important than function.
Procedural programming is based on the unreal world.	Object-oriented programming is based on the real world.

Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
Code reusability absent in procedural programming,	Code reusability present in object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	Examples: C++, Java, Python, C#, etc.

Garbage Collection:

What is Java Garbage Collection? Best Practices, Tutorials & More (stackify.com)

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine(JVM). When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

How Java Garbage Collection Works:

Java garbage collection is an automatic process. The programmer does not need to explicitly mark objects to be deleted. The garbage collection implementation lives in the JVM. Every JVM can implement garbage collection however it pleases. The only requirement is that it should meet the JVM specification. Although there are many JVMs, Oracle's HotSpot is by far the most common. It offers a robust and mature set of garbage collection options.

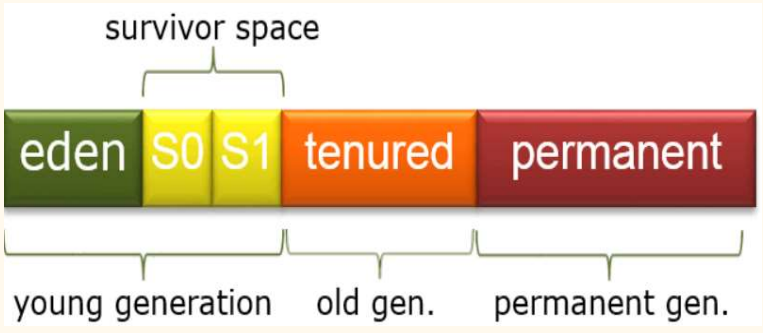
What are the Various Steps During the Garbage Collection?

While HotSpot has multiple garbage collectors that are optimized for various use cases, all its garbage collectors follow the same basic process.

- Unreferenced objects are identified and marked as ready for garbage collection.
- Marked objects are deleted. Optionally, memory can be compacted after the garbage collector deletes objects, so remaining objects are in a contiguous block at the start of the heap.
- The compaction process makes it easier to allocate memory to new objects sequentially after the JVM allocates the memory blocks to existing objects.

How Generational Garbage Collection Strategy Works?

All of HotSpot's garbage collectors implement a generational garbage collection strategy that categorizes objects by age. The rationale behind generational garbage collection is that most objects are short-lived and will be ready for garbage collection soon after creation.



What are Different Classifications of Objects by Garbage Collector?

- **Young Generation:** Newly created objects start in the Young Generation. The garbage collector further subdivides Young Generation into an Eden space, where all new objects start, and two Survivor spaces, where it moves objects from Eden after surviving one garbage collection cycle. When

objects are garbage collected from the Young Generation, it is a minor garbage collection event.

- **Old Generation:** Eventually, the garbage collector moves the long-lived objects from the Young Generation to the Old Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.
- **Permanent Generation:** The JVM stores the metadata, such as classes and methods, in the Permanent Generation. JVM garbage collects the classes from the Permanent Generation that are no longer in use.

During a full garbage collection event, unused objects from all generations are garbage collected.

What are Different Types of Garbage Collector?

HotSpot has four garbage collectors:

- **Serial:** All garbage collection events are conducted serially in one thread. JVM executes the compaction after each garbage collection.
- **Parallel:** JVM uses multiple threads for minor garbage collection. It uses a single thread for major garbage collection and Old Generation compaction. Alternatively, the Parallel Old variant uses multiple threads for major garbage collection and Old Generation compaction.
- **CMS (Concurrent Mark Sweep):** Multiple threads are used for minor garbage collection using the same algorithm as Parallel. Major garbage collection is multi-threaded, like Parallel Old. Still CMS runs concurrently alongside application processes to minimize "stop the world" events (i.e., when the garbage collector stops the application). Here, the JVM does not perform compaction of memory.
- **G1 (Garbage First):** The newest garbage collector is intended as a replacement for CMS. It is parallel and concurrent, like CMS. However, it works quite differently under the hood than older garbage collectors.

Benefits of Java Garbage Collection:

Automatically handles the deletion of unused objects or objects that are out of reach to free up vital memory resources. Programmers working in languages without garbage collection (like C and C++) must implement manual memory management in their code.

What Triggers Garbage Collection?

The Garbage Collection process is triggered by a variety of events that signal to the Garbage Collector that memory needs to be reclaimed.
Here are some common events that trigger Java Garbage Collection:

1. **Allocation Failure:** When an object cannot be allocated in the heap because there is not enough contiguous free space available, the JVM triggers the Garbage Collection to free up memory.
2. **Heap Size:** When the heap reaches a certain capacity threshold, the JVM triggers Garbage Collection to reclaim memory and prevent an OutOfMemoryError.
3. **System.gc():** Calling the System.gc() method can trigger Garbage Collection, although it does not guarantee that Garbage Collection will occur.
4. **Time-Based:** Some Garbage Collection algorithms, such as G1 Garbage Collection, use time-based triggers to initiate Garbage Collection.

Ways for requesting JVM to run Garbage Collector:

There are several ways to request the JVM to run Garbage Collector in a Java application:

1. **System.gc() method:** Calling this method is the most common way to request Garbage Collection in a Java application. However, it does not guarantee that Garbage Collection will occur as it is only a suggestion to the JVM.
2. **Runtime.getRuntime().gc() method:** This method provides another way to request Garbage Collection in a Java application. This method is similar to the System.gc() method,

and it also suggests that the JVM should run Garbage Collector, but again it does not guarantee that Garbage Collection will occur.

3. **JConsole or VisualVM:** JConsole or VisualVM is a profiling tool that is included with the Java Development Kit. These tools provide a graphical user interface that allows developers to monitor the memory usage of their Java application in real-time. They also provide a way to request Garbage Collection on-demand by clicking a button.
4. **Command-Line Options:** The JVM can be configured with various command-line options to control Garbage Collection. For example, the -Xmx option can be used to specify the maximum heap size, which can affect the frequency and duration of Garbage Collection events. The -XX:+DisableExplicitGC option can be used to disable explicit calls to System.gc() or Runtime.getRuntime().gc().

Heap Dumps:

Heap dumps are snapshots of the Java heap that can be taken at any time during the application's execution. They can be analyzed to identify memory leaks or other memory-related issues. Heap dumps can be requested using command-line options or profiling tools.

Java finally block:

Java finally block is a block used to execute important code such as closing the connection, etc. Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

Why use Java finally block?

finally block in Java can be used to put "cleanup" code such as closing a file, closing connection, etc. The important statements to be printed can be placed in the finally block.
In Java, the try, catch, and finally blocks are used together to handle exceptions.

- **try Block:** The code that might throw an exception is placed inside the try block.
- **catch Block:** If an exception occurs in the try block, it's caught by an appropriate catch block. Multiple catch blocks can be used to catch different types of exceptions.
- **finally Block:** The finally block contains code that will be executed regardless of whether an exception is thrown or not. It is often used for cleanup tasks (e.g., closing resources like files or network connections).

Error vs exception/What is the difference between error and an exception?

The term exception is shorthand for the phrase "exceptional event." An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program.
An error in a program code can be classified into two categories -

- **Semantic Errors:** Semantic Errors are also known as logical errors. And although the program will compile successfully you will not get the desired output upon running it because of the flaw in your logic. Suppose you wanted to get the summation of two numbers 'a' & 'b' as output. Your logic would have looked Something like c=a+b. But instead, you wrote c=a-b. Compiler will not identify such errors for you as it is syntactically correct (I'll explain what that means later). Such errors are difficult to find and correct and have to be identified by the programmer himself.
- **Syntactic Errors:** If you break the rules and use the wrong syntax the compiler does not understand your code. This is known as Syntactic error. The compiler will tell you exactly what it was not able to understand and even provide suggestions on how to correct it.

Exceptions are also known as run-time errors. So yes, essentially an Exception is also an error. But it's not detected during compilation phase unlike Syntactic errors and also they don't execute

normally during run time, unlike semantic Errors. These errors are detected only during runtime and prevent the normal execution of a program.

Some of the differences between exception and error classes are:

- Exception and Error both are subclasses of java.lang.Throwable class.
- We can handle Exceptions at runtime but Errors we can not handle.
- Exceptions are the objects representing the logical errors that occur at run time and makes JVM enter into the state of "ambiguity".
- The objects which are automatically created by the JVM for representing these run time errors are known as Exceptions.
- An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.
- If an exception occurs we can handle it by using a try and catch block. If Error occurs we can not handle it , program execution will be terminated.
- In Exception we have two types
    - Checked Exception
    - Unchecked Exceptions
- Errors are by default unchecked exceptions.
- Exceptions are related to applications where ad Error are related to the environment in which application is running.
- Exception are of type java.lang.Exception
- Errors are of type java.lang.Error
- Error will run at run time.
- In Exceptions Checked Exceptions will be known to the compiler so we need to handle these exceptions at compile time itself otherwise compile time Error will come.
- Unchecked Exceptions will come at run time needed to handle by using try and catch blocks.

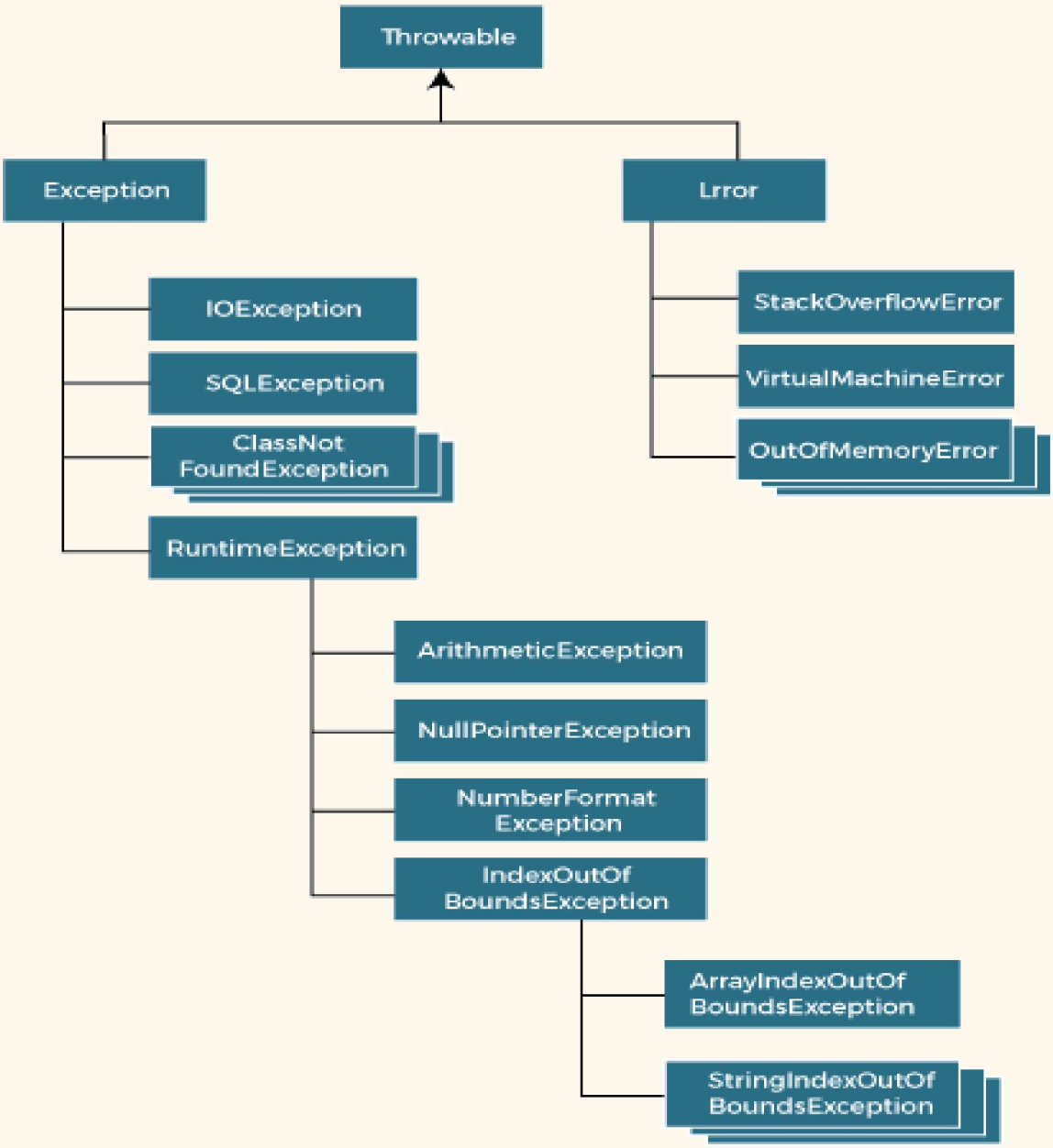
Exception Handling:

The Exception Handling in Java is a mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

Difference between Checked and Unchecked Exceptions:

1. **Checked Exceptions:** The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.
2. **Unchecked Exception:** The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.
3. **Error:** Error is irrecoverable. Some examples of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Hierarchy of Java Exception classes:



Java Exception Keywords:

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signatures.

Enums:

The Enum in Java is a data type which contains a fixed set of constants.

Properties of Enum in Java:

- Every enum is internally implemented by using Class.
- Every enum constant represents an object of type enum.
- Enum type can be passed as an argument to switch statements.
- Every enum constant is always implicitly a public static final. Since it is static, we can access it by using the enum Name. Since it is final, we can't create child enums.
- We can declare the main() method inside the enum. Hence we can invoke the enum directly from the Command Prompt.

Singleton design pattern in Java:

A class that has only one instance and provides a global point of access to it". In other words, a class must ensure that only a single instance should be created and a single object can be used by all other classes.

- Early Instantiation: creation of instances at load time.
- Lazy Instantiation: creation of instances when required.

Advantage of Singleton design pattern:

Saves memory because the object is not created at each request. Only a single instance is reused again and again.

Usage of Singleton design pattern

Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

How to create a Singleton design pattern?

To create the singleton class, we need to have a static member of class, private constructor and static factory method.

- **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
- **Private constructor:** It will prevent instantiating the Singleton class from outside the class.
- **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

Cohesion vs Coupling:

Cohesion: Cohesion is the indication of the relationship within the module. It is the concept of intro-module. Cohesion has many types but usually, high cohesion is good for software.

Coupling: Coupling is also the indication of the relationships between modules. It is the concept of the Inter-module. The coupling has also many types but typically, the low coupling is good for software.

Cohesion	Coupling
Cohesion is the concept of intro-module.	Coupling is the concept of inter-module.
Cohesion represents the relationship within a module.	Coupling represents the relationships between modules.
Increasing cohesion is good for software.	Increasing coupling is avoided for software.

Cohesion represents the functional strength of modules.	Coupling represents the independence among modules.
Highly cohesive gives the best software.	Whereas loosely coupling gives the best software.
In cohesion, the module focuses on a single thing.	In coupling, modules are connected to the other modules.
Cohesion is created between the same module.	Coupling is created between two different modules.
There are Six types of Cohesion 1. Functional Cohesion. 2. Procedural Cohesion. 3. Temporal Cohesion. 4. Sequential Cohesion. 5. Layer Cohesion. 6. Communication Cohesion.	There are Six types of Coupling 1. Common Coupling. 2. External Coupling. 3. Control Coupling. 4. Stamp Coupling. 5. Data Coupling 6. Content Coupling.