Abstract Data Types (ADTs) are a fundamental concept in computer science that refer to a high-level description of a set of operations that can be performed on a data structure, without specifying the details of how these operations are implemented. ADTs provide a way to encapsulate data and operations on that data, allowing for modularity and abstraction in software design. Here are some commonly used abstract data types and their specifications:

**1. \*\*Stack:\*\***

- \*\*Operations:\*\*

- `push(item)`: Adds an item to the top of the stack.

- `pop()`: Removes and returns the item from the top of the stack.

- `peek()`: Returns the item at the top of the stack without removing it.

- `is\_empty()`: Returns true if the stack is empty, false otherwise.

**2. \*\*Queue:\*\***

- \*\*Operations:\*\*

- `enqueue(item)`: Adds an item to the back of the queue.

- `dequeue()`: Removes and returns the item from the front of the queue.

- `peek()`: Returns the item at the front of the queue without removing it.

- `is\_empty()`: Returns true if the queue is empty, false otherwise.

**3. \*\*Linked List:\*\***

- \*\*Operations:\*\*

- `insert\_front(item)`: Adds an item to the front of the linked list.

- `insert\_back(item)`: Adds an item to the end of the linked list.

- `delete(item)`: Removes the specified item from the linked list.

- `search(item)`: Returns true if the item is present in the linked list, false otherwise.

**4. \*\*Tree:\*\***

- \*\*Operations:\*\*

- `insert(item)`: Adds an item to the tree.

- `delete(item)`: Removes the specified item from the tree.

- `search(item)`: Returns true if the item is present in the tree, false otherwise.

- `traverse()`: Visits all nodes of the tree in a specified order (in-order, pre-order, post-order).

**5. \*\*Map (Dictionary):\*\***

- \*\*Operations:\*\*

- `insert(key, value)`: Adds a key-value pair to the map.

- `delete(key)`: Removes the key-value pair with the specified key from the map.

- `lookup(key)`: Returns the value associated with the specified key.

- `is\_empty()`: Returns true if the map is empty, false otherwise.

**6. \*\*Set:\*\***

- \*\*Operations:\*\*

- `add(item)`: Adds an item to the set.

- `remove(item)`: Removes the specified item from the set.

- `contains(item)`: Returns true if the item is present in the set, false otherwise.

- `union(set2)`: Returns a new set containing all unique items from the current set and another set.

- `intersection(set2)`: Returns a new set containing common items between the current set and another set.

**The terms "concrete state space," "concrete invariant," and "abstraction function" are concepts commonly associated with formal methods, particularly in the context of software verification and program correctness. These concepts are often used in the specification and verification of abstract data types or other components of a system. Here's a brief explanation of each term:**

1. \*\*Concrete State Space:\*\*

- The concrete state space refers to the set of all possible states that a system or a data structure can be in during its execution.

- In software engineering, a state space is a representation of all possible states that a system or a program can enter.

2. \*\*Concrete Invariant:\*\*

- A concrete invariant is a condition or property that holds true for all states in the concrete state space.

- Invariants are used to describe constraints or properties that must be maintained by a program or a data structure throughout its execution.

3. \*\*Abstraction Function:\*\*

- An abstraction function is a mapping between the concrete state space and an abstract state space.

- The abstraction function defines how the concrete states are mapped to abstract states, allowing developers to reason about the system or data structure at a higher level of abstraction.

- It is a crucial part of abstraction, which is the process of simplifying complex systems to make them more manageable.

Now, let's provide a simple and abstract example in Java to illustrate these concepts. Suppose we are dealing with a stack as an abstract data type:

**import java.util.ArrayList;**

**import java.util.List;**

**public class Stack {**

**private List<Integer> items;**

**public Stack() {**

**this.items = new ArrayList<>();**

**}**

**public void push(int item) {**

**items.add(item);**

**}**

**public int pop() {**

**if (isEmpty()) {**

**throw new IllegalStateException("Stack is empty");**

**}**

**return items.remove(items.size() - 1);**

**}**

**public boolean isEmpty() {**

**return items.isEmpty();**

**}**

**// Abstraction Function: The abstraction function is implicit in this simple example.**

**// We are using a List<Integer> to represent the stack.**

**public static void main(String[] args) {**

**Stack stack = new Stack();**

**stack.push(5);**

**stack.push(10);**

**int poppedItem = stack.pop();**

**System.out.println("Popped item: " + poppedItem);**

**System.out.println("Is the stack empty? " + stack.isEmpty());**

**}}**

In this example:

- \*\*Concrete State Space:\*\* The concrete state space includes all possible states of the `Stack` object during its execution. This includes different combinations of elements in the list.

- \*\*Concrete Invariant:\*\* A concrete invariant could be that the list (representing the stack) is never allowed to have a negative size, and popping an item is only allowed when the stack is not empty.

- \*\*Abstraction Function:\*\* The abstraction function is implicit in this example. The list of integers represents the stack, and the operations (`push` and `pop`) are defined on this list. The abstraction function is the mapping between the concrete states (list states) and the abstract states (stack states).

It's important to note that in more complex systems, especially those involving concurrent programming or distributed systems, these concepts become more intricate, and formal methods may be employed for rigorous verification.

-----------------------------------------------------------------------------------------------------------------------------------

**Java is a widely used programming language that follows the principles of object-oriented programming (OOP). Here are some key features of object-oriented programming in Java:**

1. \*\*Classes and Objects:\*\*

- Java supports the concept of classes and objects. A class is a blueprint for creating objects, and objects are instances of classes.

- Classes encapsulate data and behavior, providing a way to model real-world entities in code.

2. \*\*Encapsulation:\*\*

- Encapsulation involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit known as a class.

- Access modifiers (e.g., public, private, protected) control the visibility of class members, allowing you to restrict access to certain components.

3. \*\*Inheritance:\*\*

- Inheritance allows a class (subclass/child) to inherit properties and behaviors from another class (superclass/parent).

- Java supports single inheritance for classes and multiple inheritance through interfaces.

4. \*\*Polymorphism:\*\*

- Polymorphism allows objects to be treated as instances of their parent class rather than their actual class.

- It includes method overloading (multiple methods with the same name but different parameters) and method overriding (providing a specific implementation for a method in a subclass).

5. \*\*Abstraction:\*\*

- Abstraction involves hiding the complex implementation details and showing only the essential features of an object.

- Abstract classes and interfaces in Java are mechanisms for achieving abstraction.

6. \*\*Modularity:\*\*

- Java programs are organized into modular units known as classes and packages.

- This modular structure enhances code organization, maintainability, and reusability.

7. \*\*Dynamic Binding:\*\*

- Java supports dynamic method dispatch, which allows the binding of a method call to its implementation at runtime.

- This is achieved through the use of overridden methods and polymorphism.

8. \*\*Message Passing:\*\*

- Objects communicate with each other by sending and receiving messages.

- Java supports message passing through method calls, enabling objects to interact and collaborate.

9. \*\*Encapsulation:\*\*

- Encapsulation helps in bundling the data (attributes) and methods (functions) together in a single unit.

- It restricts direct access to some of an object's components and can prevent the accidental modification of data.

10. \*\*Association:\*\*

- Objects can be associated with each other, representing relationships between them.

- Associations can be one-to-one, one-to-many, or many-to-many, allowing you to model complex relationships between objects.

**These features collectively contribute to the development of modular, scalable, and maintainable software in Java. They promote code reuse, flexibility, and ease of understanding, making it easier to manage and extend large codebases.**

**Certainly, let's discuss encapsulation, object identity, and polymorphism in the context of Java without including inheritance.**

1. \*\*Encapsulation:\*\*

- \*Definition:\* Encapsulation is the concept of bundling the data (attributes) and methods (functions) that operate on the data into a single unit known as a class.

public class Student {

private String name;

private int age;

// Getter and setter methods for encapsulation

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

if (age > 0) {

this.age = age;

} else {

System.out.println("Age must be a positive number.");

}

}

}

- \*Explanation:\* In this example, the `Student` class encapsulates the `name` and `age` attributes, providing controlled access to them through getter and setter methods. The `private` access modifier ensures that these attributes are not directly accessible from outside the class.

**2. \*\*Object Identity:\*\***

- \*Definition:\* Object identity refers to the unique identity of an object during its lifetime. It allows us to distinguish one object from another, even if they have the same data.

- \*Java Example:\*

```java

public class ObjectIdentityExample {

public static void main(String[] args) {

String str1 = new String("Hello");

String str2 = new String("Hello");

// Checking object identity using '=='

System.out.println("Are str1 and str2 the same object? " + (str1 == str2));

}}

- **\*Explanation:\* In Java, the `==` operator checks for object identity. In the example, even though `str1` and `str2` contain the same content, they are different objects, and `==` returns `false`.**

3. \*\*Polymorphism:\*\*

- \*Definition:\* Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. It includes method overloading and method overriding.

- \*Java Example:\*

```java

public class PolymorphismExample {

public static void main(String[] args) {

Animal dog = new Dog();

Animal cat = new Cat();

dog.makeSound(); // Output: Bark

cat.makeSound(); // Output: Meow

}}

class Animal {

public void makeSound() {

System.out.println("Generic Animal Sound");

} }

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Bark");

}}

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Meow");

} }

- \*Explanation:\* In this example, `Dog` and `Cat` are subclasses of `Animal`. The `makeSound` method is overridden in both subclasses, demonstrating polymorphism. The actual method called is determined at runtime based on the type of the object.

These concepts—encapsulation, object identity, and polymorphism—form essential components of object-oriented programming in Java, contributing to code organization, flexibility, and code reuse.

--------------------------------------------------------------------------------------------------------------------------------------**Inheritance is a fundamental concept in object-oriented design (OOD) that allows one class to inherit properties and behaviors from another class. It promotes code reuse, extensibility, and the creation of a hierarchy of related classes. Here are some key aspects of inheritance in object-oriented design:**

1. \*\*Class Hierarchy:\*\*

- Inheritance establishes a relationship between a superclass (base class) and one or more subclasses (derived classes).

- The superclass contains common attributes and behaviors shared by its subclasses.

2. \*\*Base Class (Superclass):\*\*

- The base class provides a blueprint for common attributes and behaviors.

- It may contain concrete methods, abstract methods, or a combination of both.

- An object of the base class can be instantiated, but it is often used as a generic representation.

3. \*\*Derived Class (Subclass):\*\*

- A derived class inherits attributes and behaviors from its superclass.

- It can add new attributes and behaviors or override existing ones.

- The derived class can be more specialized or specific than its superclass.

4. \*\*Code Reusability:\*\*

- Inheritance promotes code reuse by allowing subclasses to inherit the properties and behaviors of a superclass.

- Common functionality is defined in the base class, reducing redundancy in the code.

5. \*\*Method Overriding:\*\*

- Subclasses can provide their own implementation for methods defined in the superclass. This is known as method overriding.

- Overriding allows a subclass to customize or extend the behavior inherited from the superclass.

6. \*\*Access Modifiers:\*\*

- Access modifiers (e.g., `public`, `private`, `protected`) control the visibility of members (fields and methods) in both the superclass and subclasses.

- Access to superclass members depends on their access modifiers.

7. \*\*"is-a" Relationship:\*\*

- Inheritance establishes an "is-a" relationship between a subclass and its superclass. For example, if `Dog` is a subclass of `Animal`, you can say, "A `Dog` is an `Animal`."

8. \*\*Polymorphism:\*\*

- Inheritance contributes to polymorphism, allowing objects of a derived class to be treated as objects of the base class.

- This enables more flexible and extensible code, as objects can be manipulated at a higher level of abstraction.

9. \*\*Extensibility:\*\*

- Inheritance supports the extensibility of a system. New functionality can be added by creating new subclasses without modifying existing code in the superclass.

10. \*\*Drawbacks:\*\*

- Inappropriate use of inheritance can lead to a complex and rigid class hierarchy.

- Care should be taken to avoid the "fragile base class" problem, where changes to the base class may inadvertently affect all derived classes.

Here's a simple example in Java to illustrate inheritance:

class Animal {

String name;

Animal(String name) {

this.name = name;

}

void makeSound() {

System.out.println("Generic animal sound");

}}

class Dog extends Animal {

Dog(String name) {

super(name);

}

void makeSound() {

System.out.println("Bark");

}

void fetch() {

System.out.println(name + " is fetching.");

}

}

class Cat extends Animal {

Cat(String name) {

super(name);

}

@Override

void makeSound() {

System.out.println("Meow");

}

void scratch() {

System.out.println(name + " is scratching.");

}

}

In this example, `Dog` and `Cat` are subclasses of `Animal`, demonstrating the use of inheritance in object-oriented design.

**Design Patterns:**

Design patterns are reusable solutions to common problems encountered in software design. They provide a way to capture best practices and proven solutions to recurring design challenges. Design patterns are categorized into three main types: creational, structural, and behavioral.

**Classification:**

1. **Creational Patterns:** Concerned with the process of object creation.
   * Examples: Singleton, Factory Method, Abstract Factory.
2. **Structural Patterns:** Focus on the composition of classes and objects.
   * Examples: Adapter, Decorator, Composite.
3. **Behavioral Patterns:** Address communication between objects and the responsibility of objects.
   * Examples: Observer, Strategy, Command.

**Iterator Pattern in Java:**

The Iterator Pattern is a behavioral design pattern that provides a standardized way to traverse a collection of objects without exposing its underlying representation. It involves defining an interface for iterating over a collection and implementing it in the collection class. Java provides built-in support for the Iterator Pattern through the **Iterator** interface and related classes in the **java.util** package.

* **Java does not support destructor explain it Top of Form**

Java does not have explicit destructors like some other programming languages. Instead, it relies on automatic garbage collection to reclaim memory. Objects are automatically finalized before being removed, but developers have limited control over this process. The lack of destructors simplifies memory management and reduces the risk of resource leaks, promoting a more robust and secure programming environment.

1. Java lacks explicit destructors, unlike languages such as C++.

2. Memory management in Java is primarily handled by automatic garbage collection.

3. Objects are automatically finalized before being garbage-collected.

4. Finalization allows for cleanup operations, but it's less deterministic than explicit destructors.

5. Developers have limited control over the timing and execution of finalization.

6. The absence of destructors simplifies the language, making it less error-prone.

7. It reduces the risk of resource leaks and promotes more secure code.

8. Java's focus on garbage collection enhances memory management efficiency.

9. The lack of explicit destructors aligns with Java's design philosophy of simplicity and safety.

10. While some may miss fine-grained control, Java's approach minimizes manual memory management complexities.

**EXPLAIN PSVM IN BRIEF------------------------🡪**

1. Public: Indicates that the `main` method can be accessed from outside the class.

2. Static: Suggests that the `main` method belongs to the class itself, not to instances of the class.

3. Void: Denotes that the `main` method does not return any value.

4. Main:The entry point for the Java program, where execution begins.

5. String[] args : Represents an array of strings that can be passed as command-line arguments to the program.

6. Public static void main(String[] args): The standard signature for the main method in Java.

7. Execution starts here: When a Java program is run, the Java Virtual Machine (JVM) looks for this method to start the program.

8. Only one main method: Each Java application has only one `main` method, acting as the starting point.

9. Args parameter: Allows the program to receive inputs from the command line.

10. Key method for Java execution: Understanding the `main` method is crucial for comprehending how Java applications are initiated and run.

**EXPLAIN JVM --------------------------------------------🡪**

1. **\*\*Definition:\*\*** JVM, or Java Virtual Machine, is a crucial component of the Java Runtime Environment (JRE).

2. **\*\*Execution:\*\*** It interprets and executes Java bytecode, serving as an intermediary between compiled Java programs and the underlying hardware.

3. **\*\*Platform Independence:\*\*** JVM enables "write once, run anywhere" by providing a platform-independent execution environment for Java applications.

4**. \*\*Memory Management:\*\*** Responsible for memory allocation, garbage collection, and overall management of Java objects in the heap.

5. **\*\*Security:\*\*** JVM enhances Java's security with a sandbox environment, restricting potentially harmful actions during program execution.

6. **\*\*API Access:\*\*** Provides access to the extensive Java API, a collection of classes and methods for various programming tasks.

7. **\*\*Class Loader:\*\*** Dynamically loads Java classes into memory, locating and loading class files from the classpath as needed.

8. **\*\*JIT Compilation:\*\*** Some JVMs use Just-In-Time compilation to translate bytecode into native machine code at runtime, optimizing performance.

9. **\*\*Standard Specification:\*\*** Governed by the Java Virtual Machine Specification, ensuring consistency and compatibility across different implementations.

10. **\*\*Execution Control:\*\*** Manages the execution lifecycle of Java applications, initializing them, invoking the main method, and handling program termination.

**WHAT IS BYTE CODE? WHAT DOES THE JVM DO? WHY JAVA IS CALLED COMPILER INTERPRETER LANGUAGE?**

**1. \*\*Bytecode:\*\***

- Bytecode is an intermediate representation of a Java program that is generated by the Java compiler.

- It is not machine code but rather a set of instructions designed to be executed by the Java Virtual Machine (JVM).

**2. \*\*JVM (Java Virtual Machine):\*\***

- The JVM is responsible for executing Java bytecode.

- It interprets the bytecode or, in some cases, uses Just-In-Time (JIT) compilation to translate it into native machine code for better performance.

- JVM manages memory, handles security, and provides a runtime environment for Java applications.

**3. \*\*Java as a Compiler-Interpreter Language:\*\***

- Java is often referred to as a "compiler-interpreter" language due to its two-step process.

- Compilation: Java source code is initially compiled into bytecode by the Java compiler.

- Interpretation: The JVM then interprets this bytecode or may use JIT compilation to convert it into machine code at runtime.

- This two-step process combines the benefits of both compilation (portability, early error checking) and interpretation (dynamic adaptability, platform independence).

- The compilation step ensures that the code is syntactically correct and catches certain types of errors early. The interpretation step allows Java to run on any system with a compatible JVM, making it a platform-independent language.

In summary, bytecode is an intermediary form of code in Java, the JVM executes this bytecode, and Java is called a "compiler-interpreter" language because of its combination of compilation and interpretation steps for achieving portability and platform independence.

**EXPLAIN IMPLICIT CASTING VS EXPLICIT CASTING**

In Java, casting refers to the process of converting a value of one data type into another. There are two types of casting: implicit casting (also known as widening or automatic casting) and explicit casting (also known as narrowing or manual casting).

1. \*\*Implicit Casting:\*\*

- Implicit casting is performed automatically by the compiler when there is no loss of data.

- It occurs when a smaller data type is assigned to a larger data type.

- For example, assigning an `int` to a `long` or a `float` to a `double`.

- No explicit casting operator is needed.

**int intValue = 10;**

**long longValue = intValue; // Implicit casting from int to long**

2. \*\*Explicit Casting:\*\*

- Explicit casting is necessary when there may be a loss of data, and the programmer needs to explicitly specify the conversion.

- It is performed by the programmer using a casting operator.

- Explicit casting is required when converting from a larger data type to a smaller one.

- For example, assigning a `double` to an `int` or a `float` to an `int`.

**double doubleValue = 10.5;**

**int intValue = (int) doubleValue; // Explicit casting from double to int**

- It's important to note that explicit casting may result in loss of precision or information, and care should be taken to avoid unintended consequences.

In summary, implicit casting is done automatically by the compiler when there is no loss of data, while explicit casting is manually performed by the programmer when there may be a loss of data, and it requires the use of a casting operator.

**Explain various access specifiers in java--------------------🡪**

In Java, access specifiers (also known as access modifiers) are keywords used to define the visibility and accessibility of classes, methods, and fields within a program. There are four access specifiers in Java:

1. \*\*Public:\*\*

- The most permissive access level.

- Classes, methods, and fields declared as public are accessible from any other class.

**public class MyClass {**

**public int publicField;**

**public void publicMethod() {**

**// Code here**

**}**

**}**

2. \*\*Protected:\*\*

- Accessible within the same package and by subclasses even if they are in a different package.

- Used to provide a limited level of access.

**class MyBaseClass {**

**protected int protectedField;**

**protected void protectedMethod() {**

**// Code here**

**}**

**}**

3. \*\*Default (Package-Private):\*\*

- If no access specifier is specified, it defaults to package-private.

- Accessible within the same package, but not accessible from outside the package.

**class PackagePrivateClass {**

**int packagePrivateField;**

**void packagePrivateMethod() {**

**// Code here**

**}**

**}**

4. \*\*Private:\*\*

- The most restrictive access level.

- Accessible only within the same class.

- Used to hide the implementation details of a class.

**public class Example {**

**private int privateField;**

**private void privateMethod() {**

**// Code here**

**}**

**}**

These access specifiers help control the visibility and accessibility of classes, methods, and fields, contributing to the encapsulation and security of Java programs. They allow developers to define the level of access to different parts of their code, balancing the need for encapsulation with the need for collaboration and extensibility.

**Explain various access specifiers in java---🡪**

In Java, a wrapper class is a class that encapsulates, or wraps, the primitive data types, turning them into objects. This is necessary in situations where an object is required, but a primitive data type is available. Wrapper classes provide a way to treat primitive data types as objects.

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |

**DIFF BETWEEN OBJ AND OBJ REFERENCE**

In Java, the terms "object" and "object reference" refer to different concepts, and understanding their distinction is crucial for grasping how Java handles objects in memory.

1. \*\*Object:\*\*

- An object is an instance of a class in Java.

- It represents a real-world entity and is created from a class blueprint.

- Objects have state (attributes or fields) and behavior (methods).

- When an object is created using the `new` keyword, memory is allocated to store its data.

**// Creating an object of the class MyClass**

**MyClass myObject = new MyClass();**

2. \*\*Object Reference:\*\*

- An object reference is a variable that points to the memory location of an object.

- It holds the memory address where the actual object is stored.

- The reference allows you to access and manipulate the object's data and invoke its methods.

- Multiple references can point to the same object, enabling shared access

**// Creating object references**

**MyClass objRef1 = new MyClass();**

**MyClass objRef2 = objRef1; // objRef2 now refers to the same object as objRef1**

- Changing the state of the object using one reference affects all references pointing to that object.

Understanding the difference is crucial because in Java, objects are manipulated through references. When you pass an object to a method or assign it to another variable, you're dealing with the reference to the object, not the actual object itself. This distinction is important for managing memory, understanding object behavior, and designing robust and efficient Java programs.

**Difference between Static and Final Variable in Java**

| **S.No.** | **Static Variable** | **Final Variable** |
| --- | --- | --- |
| 1 | The static keyword is connected to occupied static classes, variables, methods and blocks. | The final keyword is connected to class, variables and methods. |
| 2 | Here, it is not mandatory to initialise the static variable while declaring it.. | Here, it is mandatory to initialise the final variable while declaring it. |
| 3 | They can be reinitialized. | We can not reinitialize the final keyword. |
| 4 | They can only access the static members of the class, and only static methods can call them. | They cannot be inherited. |
| 5 | The object of the static class cannot be created, and holds only static members. | It is not possible to inherit the final class from any class. |
| 6 | Static block is used to initialise the static variables. | Final keyword does not support such a block. |

**DIFF BETWEEN PARAMETARIZED VS NON PARAMETARIZED CONSTRUCTOR IN JAVA**

In Java, constructors are special methods used for initializing objects. Constructors can be categorized into two types: parametric (or parameterized) constructors and non-parametric (or default or no-argument) constructors.

1. \*\*Non-Parametric Constructor (Default Constructor):\*\*

- A non-parametric constructor is a constructor that takes no parameters.

- It is often created automatically by the compiler if no constructor is explicitly defined in a class.

- The purpose of the default constructor is to initialize the object with default values or perform any necessary setup.

- If a class does not have any constructor explicitly defined, Java provides a default constructor by default

**public class MyClass {**

**// Non-parametric constructor (default constructor)**

**public MyClass() {**

**// Initialization code or default setup**

**}**

**}**

2. \*\*Parametric Constructor:\*\*

- A parametric constructor is a constructor that takes one or more parameters.

- It allows the initialization of an object with specific values provided during the object creation.

- Parametric constructors are useful when you want to customize the initialization process based on external input.

**public class MyClass {**

**// Parametric constructor**

**public MyClass(int param1, String param2) {**

**// Initialization code using parameters**

**}**

**}**

- Parametric constructors can have any number and types of parameters based on the requirements of the class.

Usage of constructors depends on the needs of the class and how objects should be initialized. Classes can have both non-parametric and parametric constructors, and Java allows overloading constructors, meaning a class can have multiple constructors with different parameter lists.

In summary, non-parametric constructors are used for default initialization, while parametric constructors allow more customized object creation by accepting parameters during instantiation.

**Characteristics of final keyword in java:**

In Java, the final keyword is used to indicate that a variable, method, or class cannot be modified or extended. Here are some of its characteristics:

Final variables: When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.

Final methods: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class’s public API and should not be modified by subclasses.

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.

Overall, the final keyword is a useful tool for improving code quality and ensuring that certain aspects of a program cannot be modified or extended. By declaring variables, methods, and classes as final, developers can write more secure, robust, and maintainable code.

**\*\*Constructor:\*\***

In Java, a constructor is a special method used to initialize objects of a class. It has the same name as the class and does not have a return type. The primary purpose of a constructor is to set up the initial state of an object by initializing its attributes or performing any necessary setup. Constructors are automatically called when an object is created using the `new` keyword.

\*\*Example of a Constructor:\*\*

public class MyClass {

// Constructor

public MyClass() {

// Initialization code goes here

}

}

\*\*Why Do We Overload a Constructor:\*\*

Constructor overloading refers to the practice of defining multiple constructors within a class, each with a different parameter list. Overloading allows flexibility in how objects are instantiated, catering to various scenarios and input combinations. Here are some reasons why constructor overloading is beneficial:

1. \*\*Initialization with Different Values:\*\*

- Overloading allows objects to be initialized with different sets of values, accommodating different use cases.

2. \*\*Flexibility in Object Creation:\*\*

- Provides flexibility in creating objects with various combinations of parameters, avoiding the need for multiple constructors for each case.

3. \*\*Default Values:\*\*

- Allows the definition of a default constructor with no parameters and additional constructors with specific parameters, providing default values when needed.

4. \*\*Avoids Redundant Code:\*\*

- Helps avoid redundant code by consolidating common initialization logic into a single constructor that can be reused by other overloaded constructors.

\*\*Example of Constructor Overloading:\*\*

public class Rectangle {

private int length;

private int width;

// Default constructor

public Rectangle() {

length = 0,width = 0;

}

// Parameterized constructor

public Rectangle(int length, int width) {

this.length = length;

this.width = width;

}}

In the example above, the class `Rectangle` has a default constructor that initializes length and width to 0, and a parameterized constructor that allows the user to specify custom values during object creation. Constructor overloading provides a convenient way to create objects with different initial states based on the needs of the program.

# **this keyword in Java**

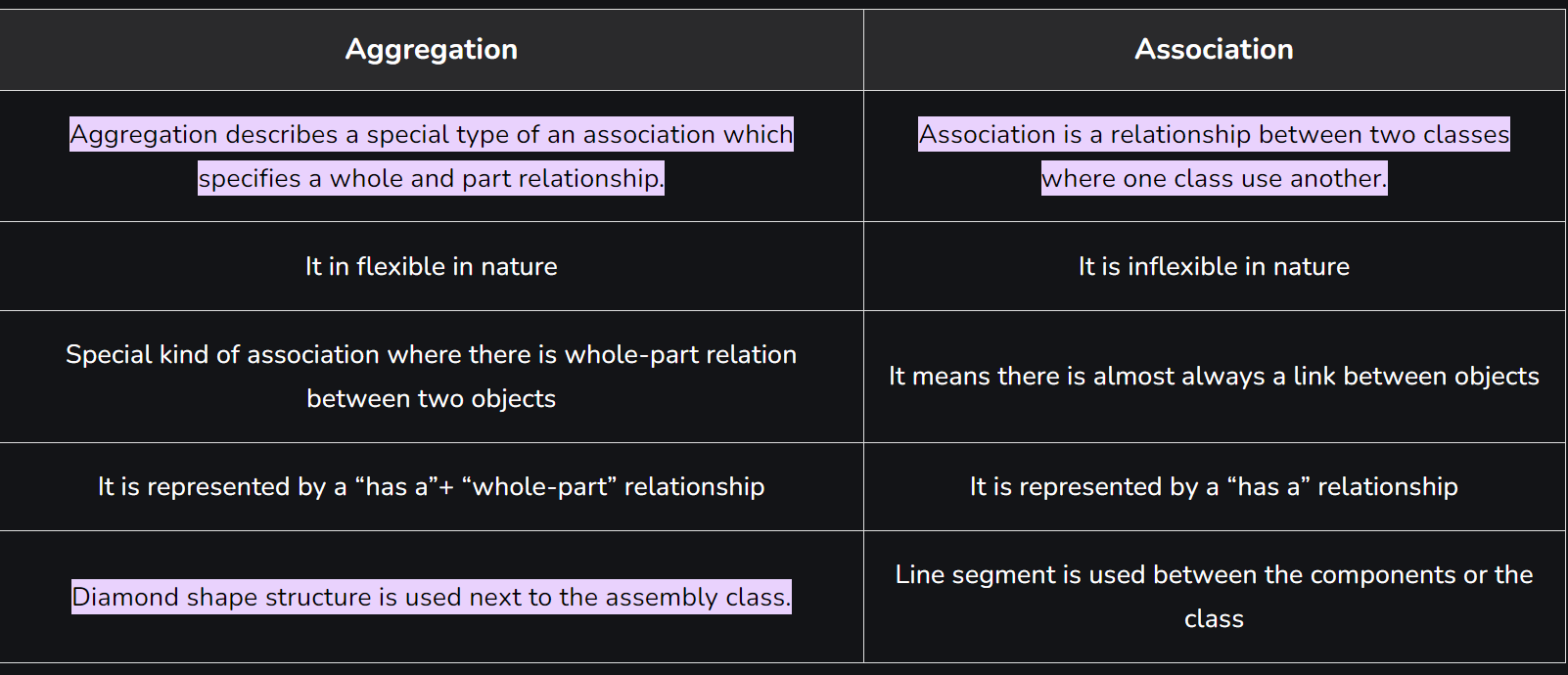
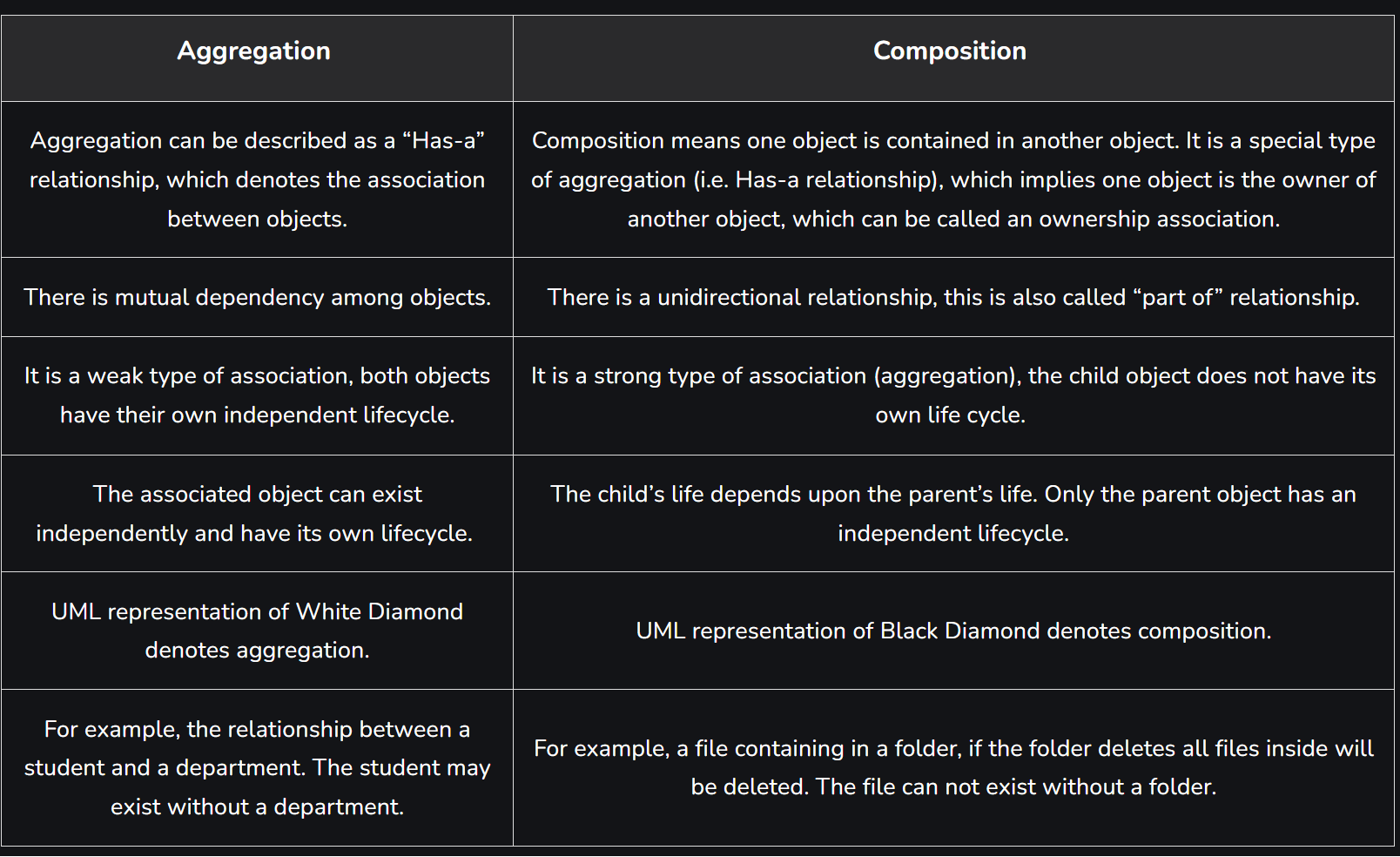
There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.

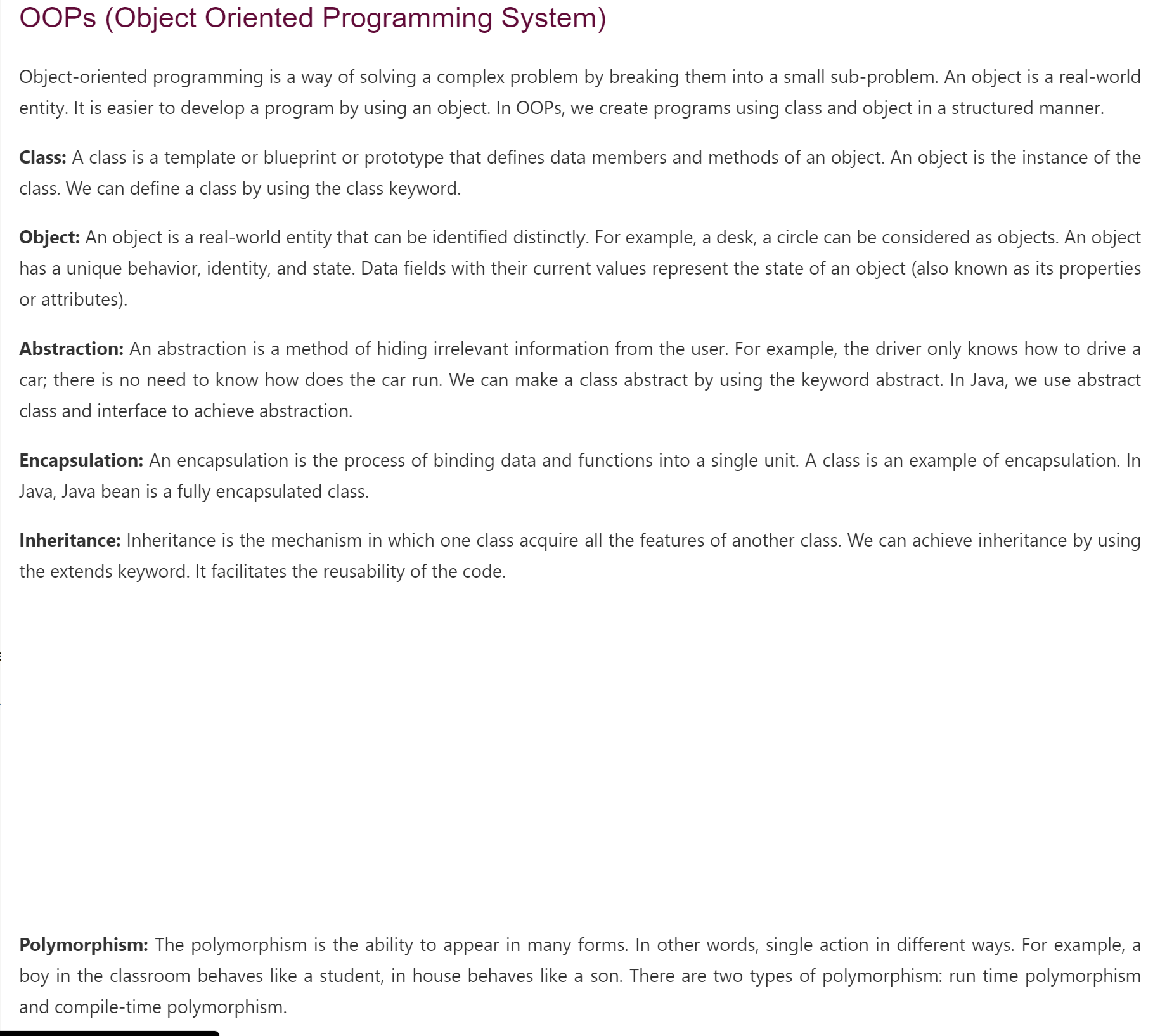
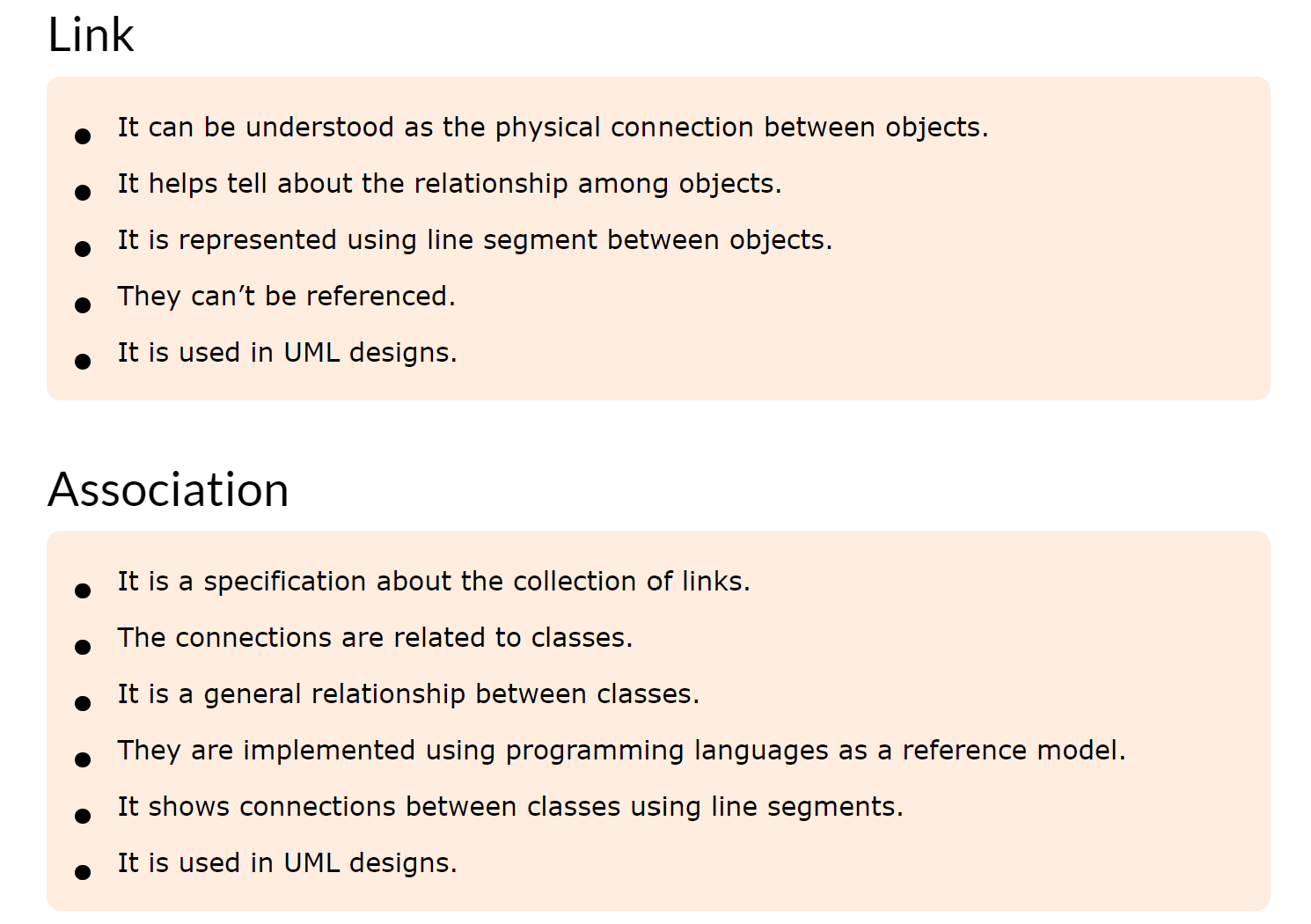
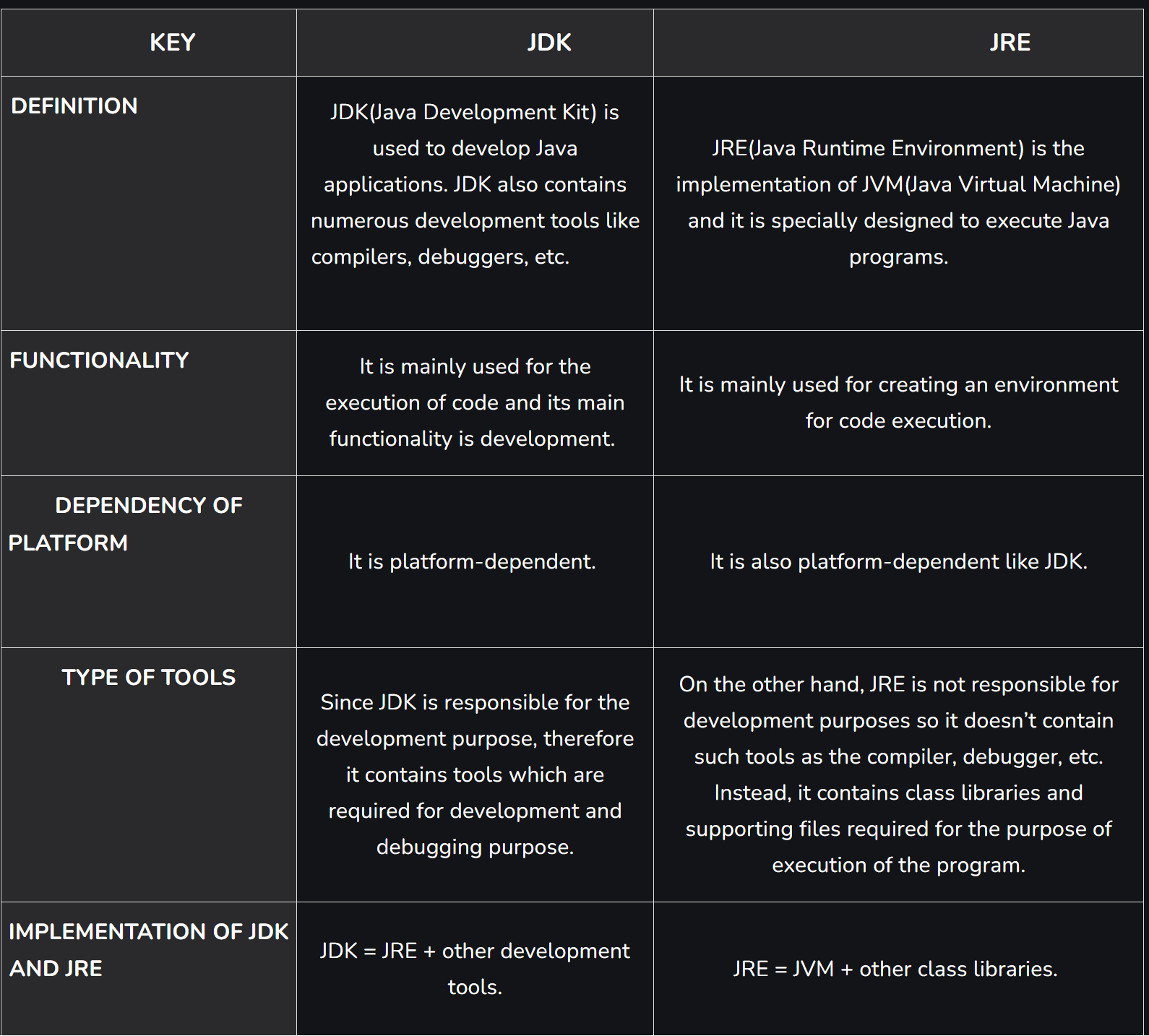
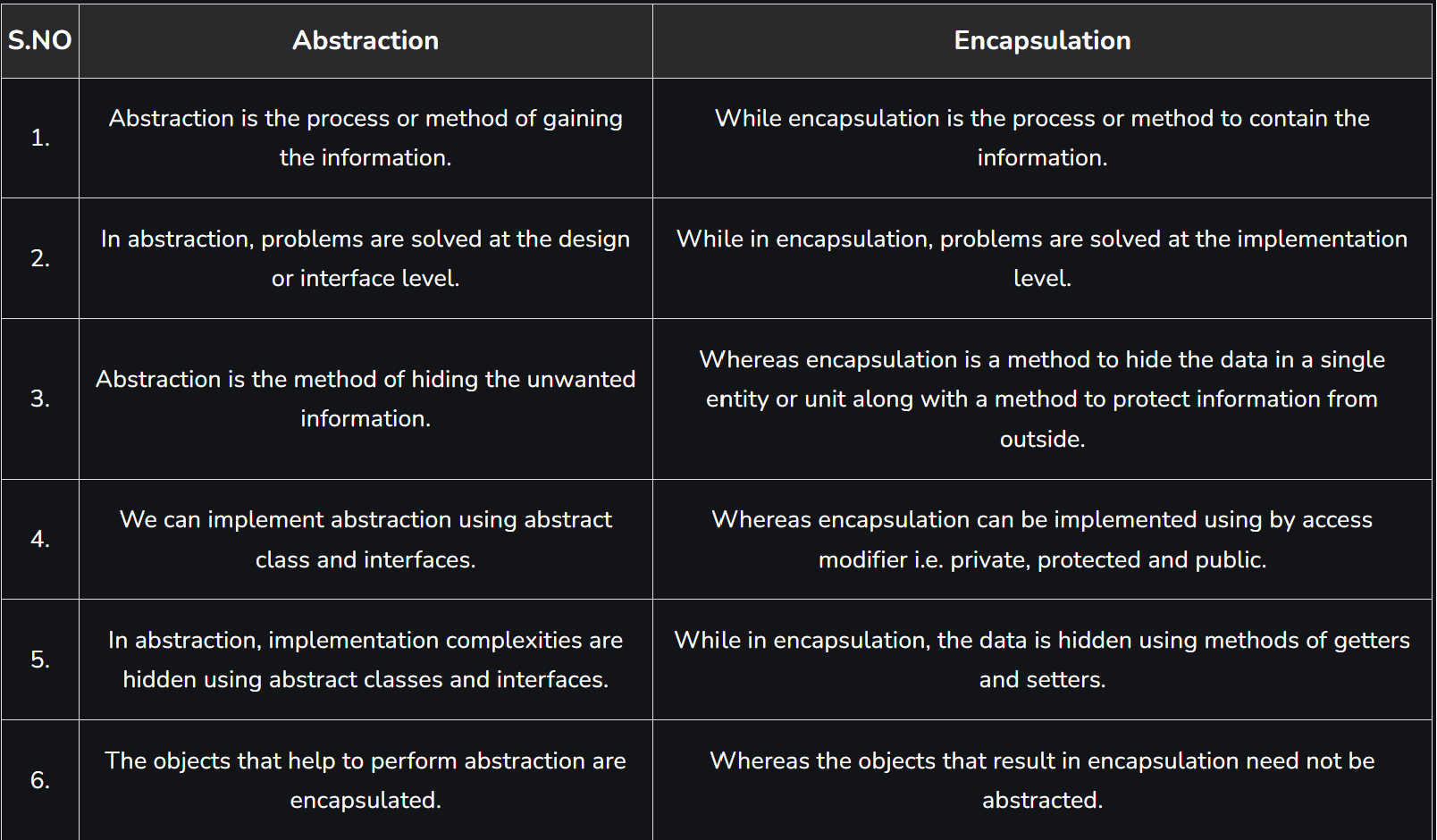
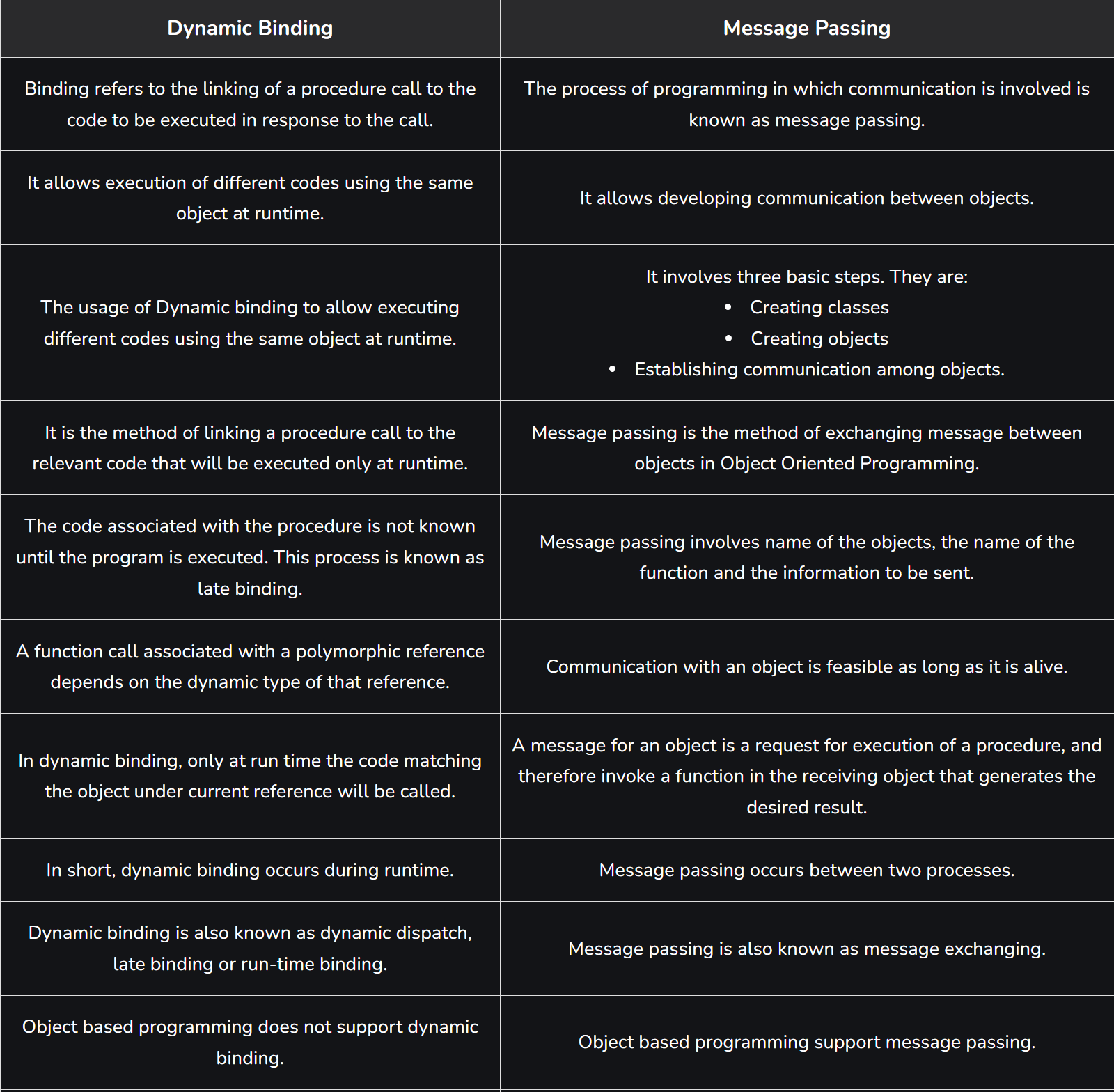


## **Usage of Java this keyword**

Here is given the 6 usage of java this keyword.

1. [this can be used to refer current class instance variable.](https://www.javatpoint.com/this1)
2. [this can be used to invoke current class method (implicitly)](https://www.javatpoint.com/this2)
3. [this() can be used to invoke current class constructor.](https://www.javatpoint.com/this3)
4. [this can be passed as an argument in the method call.](https://www.javatpoint.com/this4)
5. [this can be passed as argument in the constructor call.](https://www.javatpoint.com/this5)
6. [this can be used to return the current class instance from the method.](https://www.javatpoint.com/this6)







In Java, Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essential units are not displayed to the user.

**What is Abstraction in Java?**

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

**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.

**In Java, the following some important observations about abstract classes are as follows:**

1. An instance of an abstract class can not be created.

2. Constructors are allowed.

3. We can have an abstract class without any abstract method.

4. There can be a final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”

5. We can define static methods in an abstract class

6. We can use the abstract keyword for declaring top-level classes (Outer class) as well as inner classes as abstract

7. If a class contains at least one abstract method then compulsory should declare a class as abstract

8. If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method

**2. Base class and derived class.**

**Base Class:** A base class is a class in Object-Oriented Programming language, from which other classes are derived. The class which inherits the base class has all members of a base class as well as can also have some additional properties. The Base class members and member functions are inherited to Object of the derived class. A base class is also called parent class or superclass.

**Derived Class:** A class that is created from an existing class. The derived class inherits all members and member functions of a base class. The derived class can have more functionality with respect to the Base class and can easily access the Base class. A Derived class is also called a child class or subclass.

| **Early Binding** | **Late Binding** |
| --- | --- |
| It is a compile-time process | It is a run-time process |
| The method definition and method call are linked during the compile time. | The method definition and method call are linked during the run time. |
| Actual object is not used for binding. | Actual object is used for binding. |
| For example: [Method overloading](https://www.geeksforgeeks.org/method-overloading-in-java/) | For example: [Method overriding](https://www.geeksforgeeks.org/overriding-in-java/) |
| Program execution is faster | Program execution is slower |

**Early Binding:**

The binding which can be resolved at compile time by the compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time.

public class NewClass {

public static class superclass {

static void print()

{

System.out.println("print in superclass.");

}

}

public static class subclass extends superclass {

static void print()

{

System.out.println("print in subclass.");

}

}

public static void main(String[] args)

{

superclass A = new superclass();

superclass B = new subclass();

A.print();

B.print();

}

}

**Late binding:**

In the late binding or dynamic binding, the compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have the same method.

Example:

public class NewClass {

public static class superclass {

void print()

{

System.out.println("print in superclass.");

}

}

public static class subclass extends superclass {

@Override

void print()

{

System.out.println("print in subclass.");

}

}

public static void main(String[] args)

{

superclass A = new superclass();

superclass B = new subclass();

A.print();

B.print();

}

}

Output:

print in superclass.

print in subclass.

**4. main characteristics of oops language are:**

4. Pillars of OOPs

• Abstraction

• Encapsulation

• Inheritance

• Polymorphism

• Compile-time polymorphism

• Runtime polymorphism

**Pillar 1: Abstraction**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or non-essential units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the object.

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

The abstract method contains only method declaration but not implementation.

//abstract class

abstract class GFG{

//abstract methods declaration

abstract void add();

abstract void mul();

abstract void div();

}

**Pillar 2: Encapsulation**

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

• Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.

• In encapsulation, the data in a class is hidden from other classes, which is similar to what data-hiding does. So, the terms “encapsulation” and “data-hiding” are used interchangeably.

• Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

//Encapsulation using private modifier

//Employee class contains private data called employee id and employee name

class Employee {

private int empid;

private String ename;

}

**Pillar 3: Inheritance**

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using extends keyword. Inheritance is also known as “is-a” relationship.

Let us discuss some frequently used important terminologies:

• Superclass: The class whose features are inherited is known as superclass (also known as base or parent class).

• Subclass: The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

• Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

//base class or parent class or super class

class A{

//parent class methods

void method1(){}

void method2(){}

}

//derived class or child class or base class

class B extends A{ //Inherits parent class methods

//child class methods

void method3(){}

void method4(){}

}

**Pillar 4: Polymorphism**

It refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called polymorphism.

sleep(1000) //millis

sleep(1000,2000) //millis,nanos

Note: Polymorphism in Java is mainly of 2 types:

**1. Overloading**

**2. Overriding**

**5. The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritances in Java using Interface.**

Uses of Interfaces in Java

Uses of Interfaces in Java are mentioned below:

• It is used to achieve total abstraction.

• Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.

• Any class can extend only 1 class but can any class implement an infinite number of interface.

• It is also used to achieve loose coupling.

• Interfaces are used to implement abstraction.

Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

1. Without bothering about the implementation part, we can achieve the security of the implementation.

2. In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

6. Dynamic Method

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

• When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

• At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed

• A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is a compile-time polymorphism. | Method overriding is a run-time polymorphism. |
| Method overloading helps to increase the readability of the program. | Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass. |
| It occurs within the class. | It is performed in two classes with inheritance relationships. |
| Method overloading may or may not require inheritance. | Method overriding always needs inheritance. |
| In method overloading, methods must have the same name and different signatures. | In method overriding, methods must have the same name and same signature. |
| In method overloading, the return type can or can not be the same, but we just have to change the parameter. | In method overriding, the return type must be the same or co-variant. |
| Static binding is being used for overloaded methods. | Dynamic binding is being used for overriding methods. |
| Poor Performance due to compile time polymorphism. | It gives better performance. The reason behind this is that the binding of overridden methods is being done at runtime. |
| Private and final methods can be overloaded. | Private and final methods can’t be overridden. |
| The argument list should be different while doing method overloading. | The argument list should be the same in method overriding. |

**Ways to prevent method overriding:**

there are ways to prevent method overriding in child classes which are as follows:

Methods:

1. Using a static method

2. Using private access modifier

3. Using default access modifier

4. Using the final keyword method

**Method 1: Using a static method**

This is the first way of preventing method overriding in the child class. If you make any method static then it becomes a class method and not an object method and hence it is not allowed to be overridden as they are resolved at compilation time and overridden methods are resolved at runtime.

**Method 2: Using private access modifier**

Making any method private reduces the scope of that method to class only which means absolutely no one outside the class can reference that method.

**Method 3: Using default access modifier**

This can only be used when the method overriding is allowed within the same package but not outside the package. Default modifier allows the method to be visible only within the package so any child class outside the same package can never override it.

**Method 4: Using the final keyword method**

The final way of preventing overriding is by using the final keyword in your method. The final keyword puts a stop to being an inheritance. Hence, if a method is made final it will be considered final implementation and no other class can override the behavior.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface 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**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare 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 keyword "extends". | An **interface** can be implemented using 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(); } |

1. **The use of super keyword.**

The super keyword in Java is a reference variable that is used to refer to parent class objects

• 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.

• 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.

• 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.

• 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.

• 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.

• 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 keyword in Java**

**• Use of super with variables**

**• Use of super with methods**

**• Use of super with constructors**

**1. Use of super with Variables**

This scenario occurs when a derived class and base class has the same data members. In that case, there is a possibility of ambiguity for the JVM.

// Java code to show use of super

// keyword with variables

// Base class vehicle

class Vehicle {

int maxSpeed = 120;

}

// sub class Car extending vehicle

class Car extends Vehicle {

int maxSpeed = 180;

void display()

{

// print maxSpeed of base class (vehicle)

System.out.println("Maximum Speed: "

+ super.maxSpeed);

}

}

// Driver Program

class Test {

public static void main(String[] args)

{

Car small = new Car();

small.display();

}

}

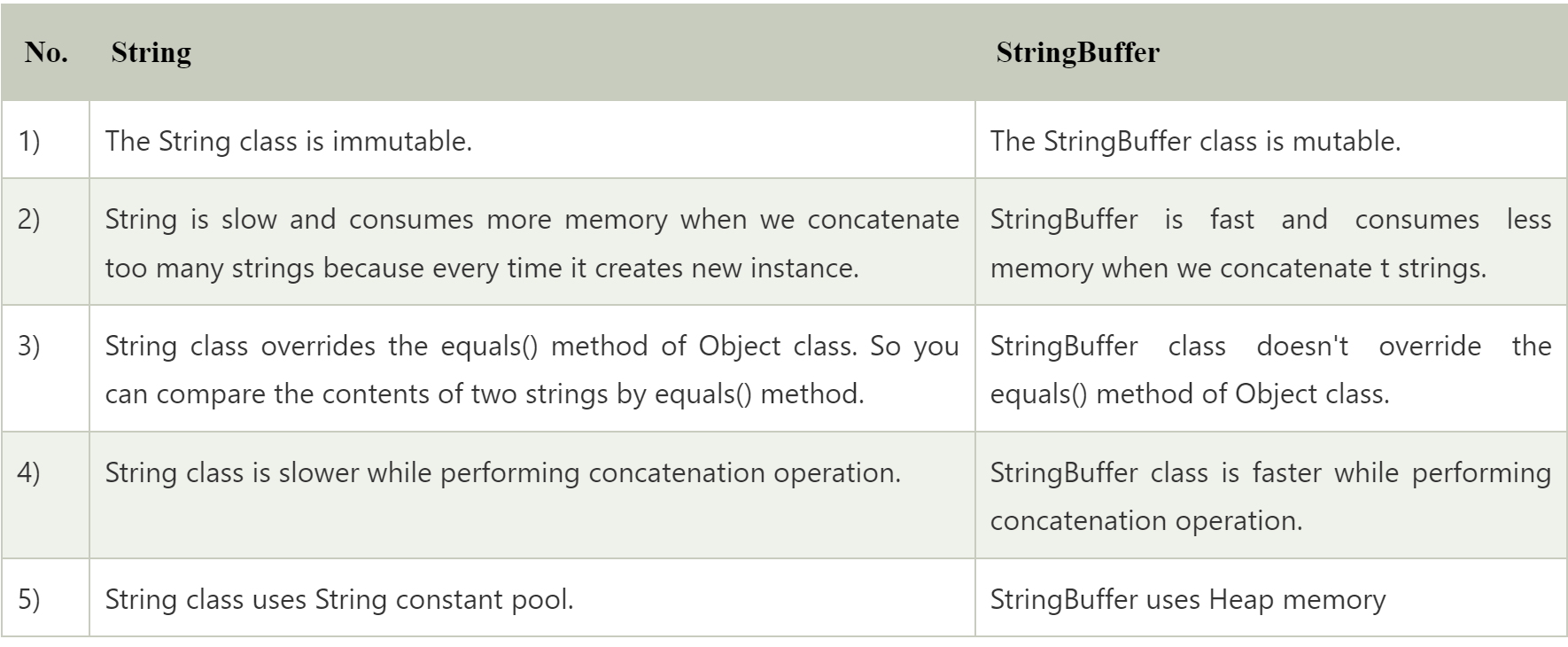
In the above example, both the base class and subclass have a member maxSpeed. We could access maxSpeed of base class in subclass using super keyword.

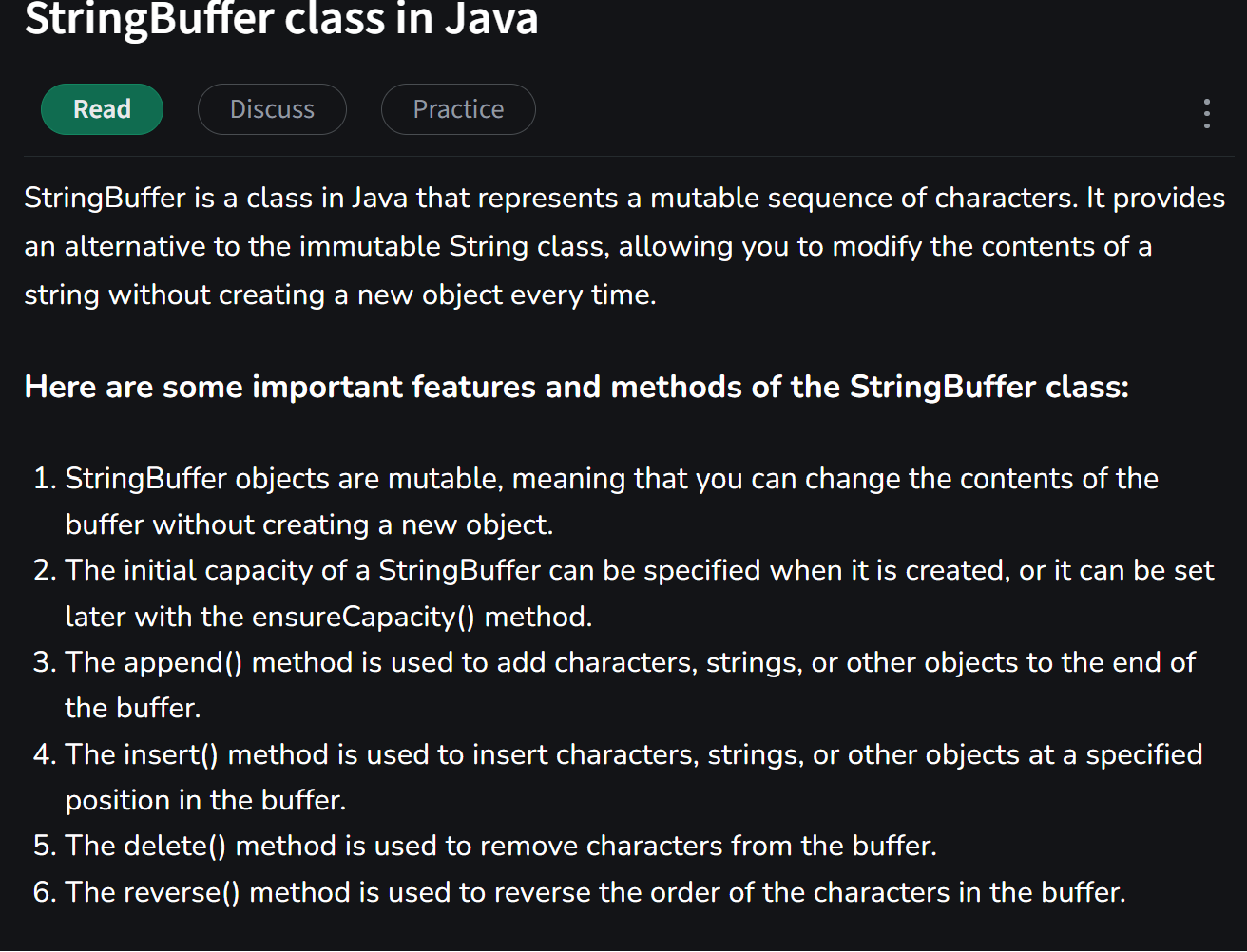
**2.Use of super with Methods**

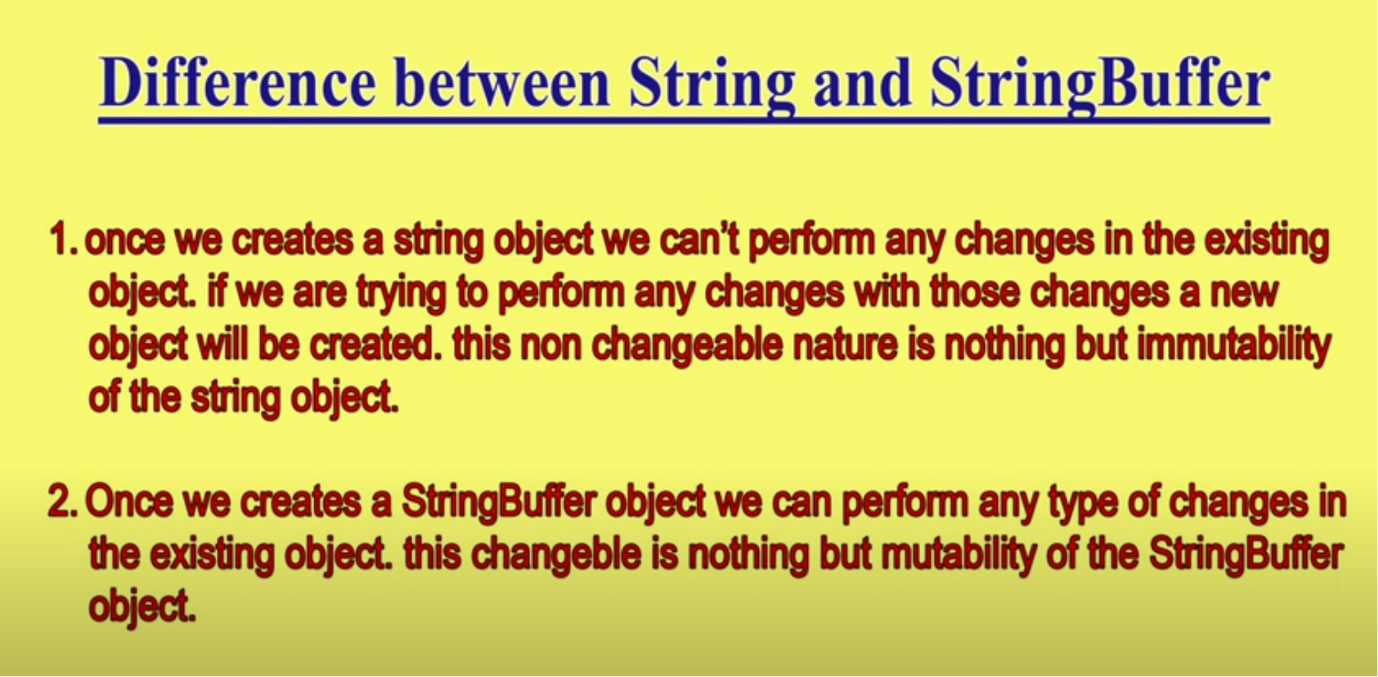
This is used when we want to call the parent class method. So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword.

**3.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.







**public class StringBufferExample {**

**public static void main(String[] args)**

**{**

**StringBuffer sb = new StringBuffer();**

**sb.append("Hello");**

**sb.append(" ");**

**sb.append("world");**

**String message = sb.toString();**

**System.out.println(message);**

**}**

**}**

There are several advantages of using StringBuffer over regular String objects in Java:

Mutable: StringBuffer objects are mutable, which means that you can modify the contents of the object after it has been created. In contrast, String objects are immutable, which means that you cannot change the contents of a String once it has been created.

Efficient: Because StringBuffer objects are mutable, they are more efficient than creating new String objectseach time you need to modify a string. This is especially true if you need to modify a string multiple times, as each modification to a String object creates a new object and discards the old one.

Thread-safe: StringBuffer objects are thread-safe, which means multiple threads can access it simultaneously( they can be safely accessed and modified by multiple threads simultaneously). In contrast, String objects are not thread-safe, which means that you need to use synchronization if you want to access a String object from multiple threads.

Overall, if you need to perform multiple modifications to a string, or if you need to access a string from multiple threads, using StringBuffer can be more efficient and safer than using regular String objects.

StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. StringBuffer may have characters and substrings inserted inthe middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

**New State**

**Runnable State**

**Blocked State**

**Waiting State**

**Timed Waiting State**

**Terminated State**

Lifecycle and States of a Thread in Java

States of Thread in its Lifecycle

**Life Cycle of a Thread**

**There are multiple states of the thread in a lifecycle as mentioned below:**

**New Thread**: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.

**Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.

A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

**Blocked:** The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.

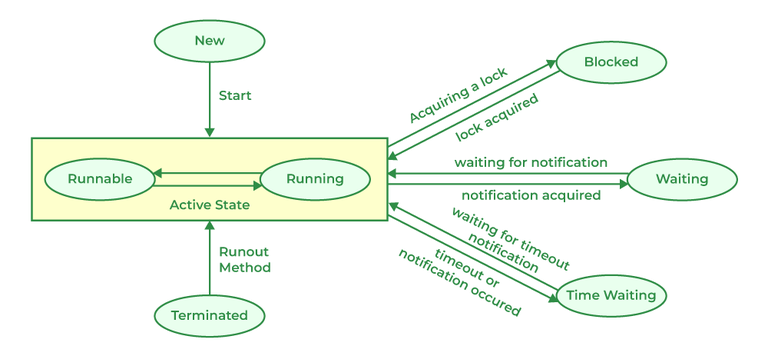
**Waiting state:** The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.

**Timed Waiting**: A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

**Terminated State:** A thread terminates because of either of the following reasons:

Because it exits normally. This happens when the code of the thread has been entirely executed by the program.

Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

****

## What Is Process?

When a program is under execution, then it is known as a process. Basically, a process allows you to execute all the relevant tasks in a program. In this, the operating system helps you to design, plan, schedule, and terminate the processes used by the CPU.

## What Is thread?

In the world of computer science, a thread is the smallest segment of instructions that can be handled independently by a scheduler.

|  |  |  |
| --- | --- | --- |
| **S.No** | **Process** | **Thread** |
| 1. | When a program is under execution, then it is known as a process. | A segment of a process is known as thread. |
| 2. | It consumes maximum time to stop. | It consumes minimum time to stop. |
| 3. | It needs more time for work and conception. | It needs less time for work and conception. |
| 4. | Context switching takes maximum time here. | Here, context switching takes minimum time. |
| 5. | It is not that effective in terms of communication. | It is effective in terms of communication. |
| 6. | It takes more resources. | It takes less resources. |
| 7. | It is a heavy weight process. | It is a light weight process. |
| 8. | If one process is obstructed then it will not affect the operation of another process. | If one thread is obstructed then it will affect the execution of another process. |

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**  
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

**Steps to create a new thread using Runnable**

1. Create a Runnable implementer and implement the run() method.
2. Instantiate the Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instances.
3. Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start() creates a new Thread that executes the code written in run(). Calling run() directly doesn’t create and start a new Thread, it will run in the same thread. To start a new line of execution, call start() on the thread.

**Example 1**

public class RunnableDemo {

public static void main(String[] args)

{

System.out.println("Main thread is- "

+ Thread.currentThread().getName());

Thread t1 = new Thread(new RunnableDemo().new RunnableImpl());

t1.start();

}

private class RunnableImpl implements Runnable {

public void run()

{

System.out.println(Thread.currentThread().getName()

+ ", executing run() method!");

}

}

}