**Access Modifiers:** The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.

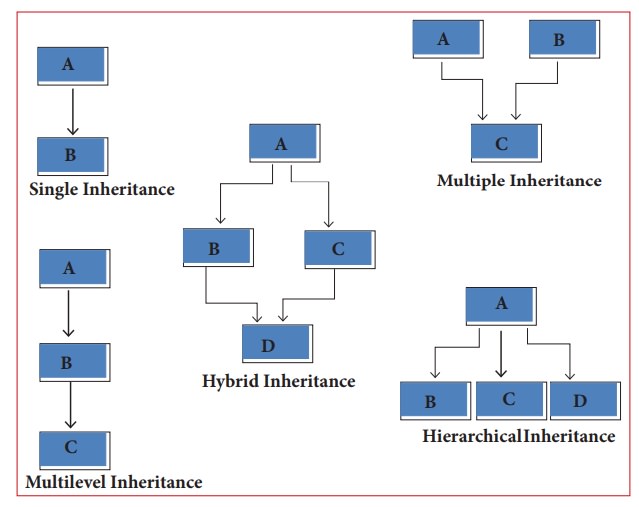
There are four types of Java access modifiers:

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.

**Inheritance :** Inheritance is a mechanism in object-oriented programming that allows a subclass(derived) to inherit properties and behaviours from super(base) class .

* This promotes code reusability and establishes a relationship between classes.

Types of Inheritance:

(Java does not support multiple inheritance for classes, but it supports it for interfaces)

Lets take example of hierarchical inheritance:



**Polymorphism:**

Polymorphism, literally means “different forms,” is one of the core concepts of OOP. Polymorphism explores how to create and use multiple methods with the same name to execute multiple different functionalities — like adding two functions with the same name but that accept different parameters and different data Types.

In Java Polymorphism is mainly divided into two types:

* Compile-time Polymorphism
* Runtime Polymorphism

**Compile Time Polymorphism:**

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

types:

* Function overloading
* Operator overloading

**Note:** But Java doesn’t support the Operator Overloading.



**Runtime Polymorphism in Java:**

**Runtime Polymorphism** concept in object-oriented programming where a method call is resolved at runtime rather than at compile time. This is achieved through method overriding, where a subclass provides a specific implementation for a method that is already defined in its superclass.

Types:

* Virtual Functions: In Java, all non-static methods are implicitly virtual, which means they exhibit polymorphic behavior by default. Unlike C++, you don't explicitly use the **virtual** keyword in Java to declare a method as virtual. Instead, method overriding is achieved using the **@Override** annotation.

Example of runtime polymorphism:



**Encapsulation**: It is the bundling of data (attributes or fields) and methods that operate on the data into a single unit, often known as a class. Encapsulation restricts direct access to some of an object's components and can prevent the accidental modification of data.

**Purpose**: Data Security.

* Encapsulation hides the internal state of the object and restricts direct access to its data.
* Private members are accessible only within the class, providing a way to enforce encapsulation.



**Abstraction:** It involves simplifying complex systems by modelling classes based on the essential properties and behaviours they possess, while ignoring or hiding the irrelevant details. Abstraction provides a way to focus on the essential characteristics of an object while ignoring the non-essential details.

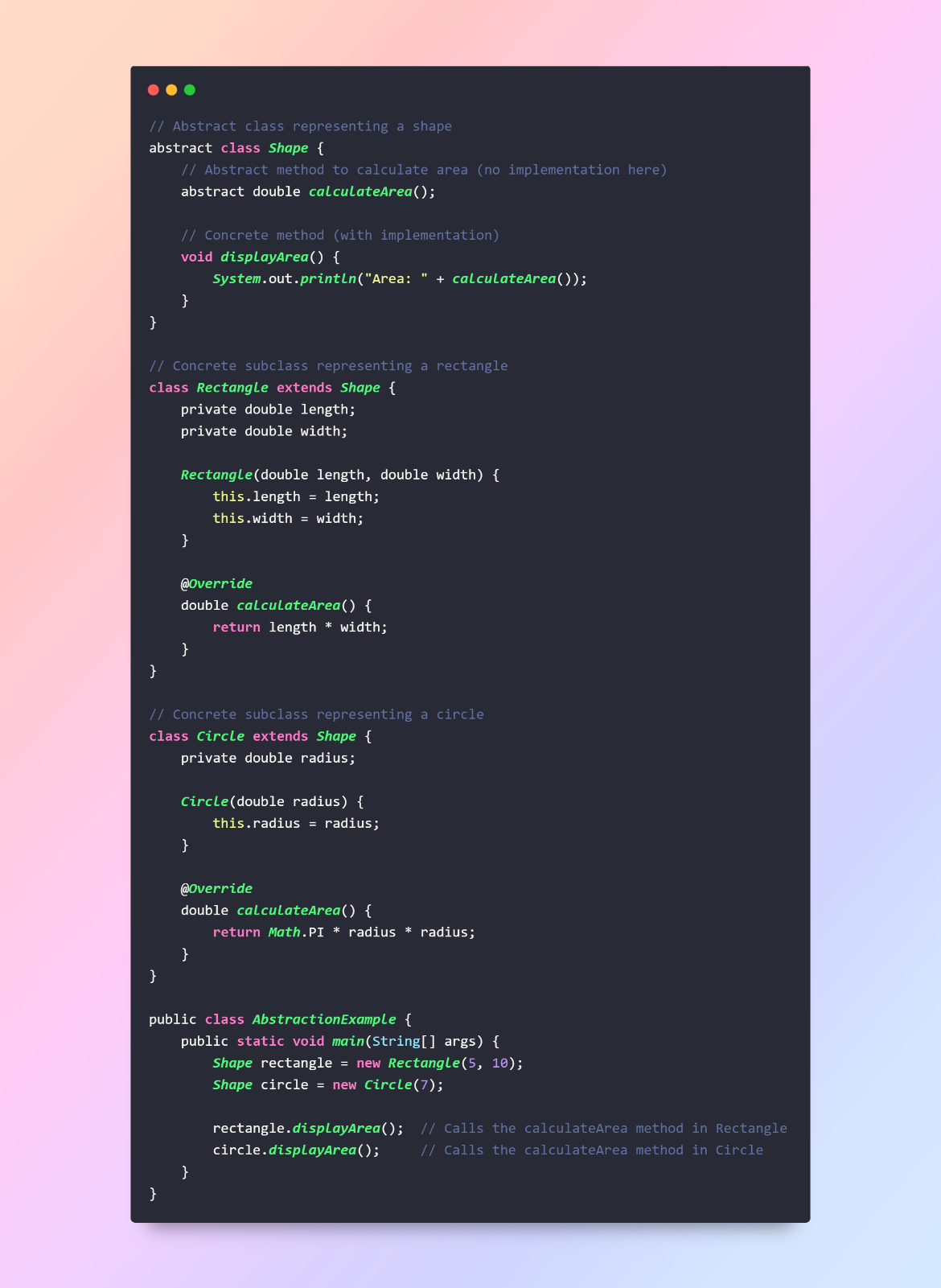
•Abstract classes and interfaces in OOP provide a way to define abstract types.

•Abstract types can have abstract methods, which are declared but not implemented in the abstract class or interface.

•Abstract methods are methods without a body (implementation).

•They are declared in an abstract class or interface and must be implemented by concrete subclasses or classes that implement the interface.

Example Below : Using class using class and interface.



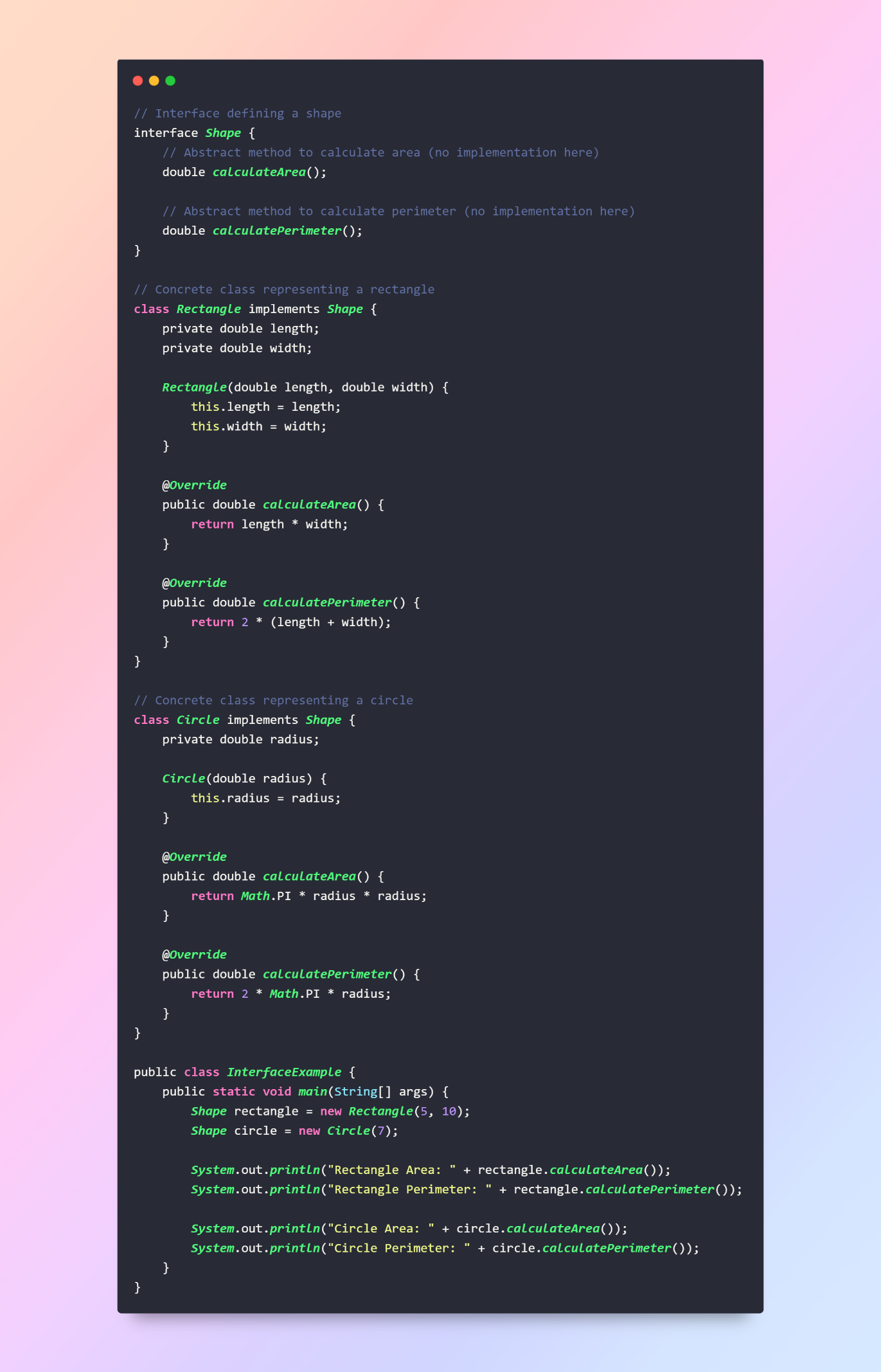
A screen shot of a computer

Description automatically generated

**Interfaces:** in Java provide a way to achieve abstraction by defining a contract that classes must adhere to. An interface is a collection of abstract methods (methods without a body) that can be implemented by any class. It allows multiple classes to share a common set of method signatures, promoting code flexibility and maintainability.

* Interfaces contain abstract methods, which are declared without providing an implementation.
* Any class that implements an interface must provide concrete implementations for all its abstract methods.
* A class in Java can implement multiple interfaces, allowing it to inherit the abstract methods from each interface.
* This is a form of multiple inheritance, which is not allowed for classes.

**Example:**



**Constructor:**   
A **constructor** in Java is a special type of method that is invoked when an object is created. It is used to initialize the object and set its initial state. Constructors have the same name as the class and do not have a return type, not even **void**.

**Types:**

* Default Constructor
* Parameterized Constructor
* Copy Constructor

**Default Constructor:** If a class does not have any constructor, Java provides a default constructor automatically.

The default constructor takes no parameters and initializes the object to default values.

**Parameterized Constructor:** Constructors can have parameters, allowing you to initialize the object with specific values at the time of creation.

**Constructor Overloading:** Just like method overloading Like regular methods, constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists.

**Examples:**



Constructor Overloading:

A screen shot of a computer

Description automatically generated

Copy Constructor: A Copy Constructor is a constructor that creates a new object by copying the properties of an existing object of the same class. Although Java doesn't have a built-in copy constructor, you can manually implement similar functionality.



**Final Keyword:** The Finalkeyword in Java is a modifier that can be applied to variables, methods, and classes. It indicates that the marked entity is not allowed to be changed or overridden, depending on where it is used.

**Final Variable:**

* If a variable is declared as final, its value cannot be changed once it has been assigned.
* It is a constant, and it is common to declare constants in all uppercase letters.

**Final Method:**

* If a method is declared as **final** in a class, it cannot be overridden by any subclass.

**Final Class:**

* If a class is declared as **final**, it cannot be subclassed (no other class can extend it).

**Static Keyword:** The static keyword in Java is used to define a member (variable or method) that belongs to the class rather than to any specific instance of the class. This means that the member is shared among all instances of the class and can be accessed using the class name rather than an instance.

**Static Variables (Class Variables):** A static variable (or class variable) is a variable that belongs to the class rather than to instances of the class. It is shared among all instances of the class.

A screen shot of a computer program

Description automatically generated

**Static Methods:** A static method is a method that belongs to the class rather than to instances of the class. It can be called using the class name and does not have access to instance-specific data.

A computer screen shot of a program code

Description automatically generated

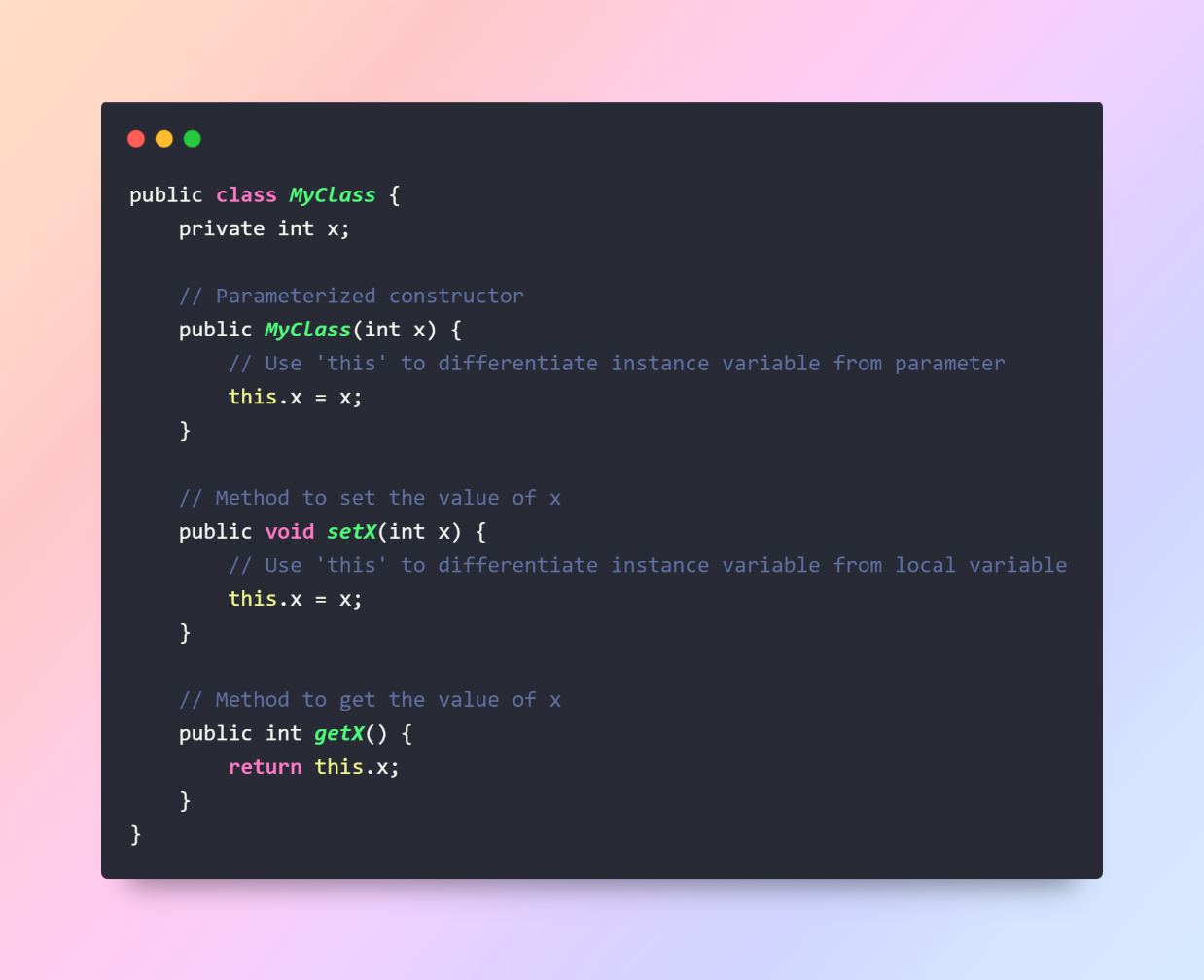
**Static Block:** A static block is a block of code that is executed when the class is loaded into the memory. It is used to initialize static variables or perform other static operations.

A computer screen shot of a program

Description automatically generated

* In Java, the static keyword is not used to make a class itself static. In Java, classes are inherently non-static. However, if you are referring to a class nested within another class, the static keyword can be applied to the nested class.

**`this` Keyword:** The this keyword is a reference variable in Java that refers to the current object. It is primarily used to differentiate instance variables from local variables when they have the same names. Additionally, it can be used to invoke the current object's method or constructor.



**`super` Keyword:** The super keyword in Java is used to refer to the immediate parent class object. It is often used to call the parent class methods, access parent class fields, or invoke the parent class constructor.



**Using `this` and `super` in Constructors:** Both this and super can be used in constructors to invoke another constructor within the same class (this) or the immediate parent class (super).

