Java OOPs (Object-Oriented Programming) Concepts

* [Object](https://www.tutorialspoint.com/java/java_object_classes.htm)
* Class
* [Inheritance](https://www.tutorialspoint.com/java/java_inheritance.htm)
* [Polymorphism](https://www.tutorialspoint.com/java/java_polymorphism.htm)
* [Abstraction](https://www.tutorialspoint.com/java/java_abstraction.htm)
* [Encapsulation](https://www.tutorialspoint.com/java/java_encapsulation.htm)

1. Object

In object-oriented programming, an object is an entity that has two characteristics (states and behavior). Some of the real-world objects are book, mobile, table, computer, etc. An object is a [variable](https://www.tutorialspoint.com/java/java_variable_types.htm) of the type class, it is a basic component of an object-oriented programming system. A class has the [methods](https://www.tutorialspoint.com/java/java_class_methods.htm) and [data members](https://www.tutorialspoint.com/java/java_class_attributes.htm) (attributes), these methods and data members are accessed through an object. Thus, an object is an instance of a class.

2. Class

In object-oriented programming, a class is a blueprint from which individual objects are created (or, we can say a class is a [data type](https://www.tutorialspoint.com/java/java_basic_datatypes.htm) of an object type). In Java, everything is related to classes and objects. Each class has its methods and attributes that can be accessed and manipulated through the objects.

3. Inheritance

In object-oriented programming, inheritance is a process by which we can reuse the functionalities of existing classes to new classes. In the concept of inheritance, there are two terms base (parent) class and derived (child) class. When a class is inherited from another class (base class), it (derived class) obtains all the properties and behaviors of the base class.

4. Polymorphism

The term "polymorphism" means "many forms". In object-oriented programming, polymorphism is useful when you want to create multiple forms with the same name of a single entity. To implement polymorphism in Java, we use two concepts [method overloading](https://www.tutorialspoint.com/java/java_method_overloading.htm) and [method overriding](https://www.tutorialspoint.com/java/java_overriding.htm).

The method overloading is performed in the same class where we have multiple methods with the same name but different parameters, whereas the method overriding is performed by using the inheritance where we can have multiple methods with the same name in parent and child classes.

5. Abstraction

In object-oriented programming, an abstraction is a technique of hiding internal details and showing functionalities. The abstract classes and interfaces are used to achieve abstraction in Java.

The real-world example of an abstraction is a Car, the internal details such as the engine, process of starting a car, process of shifting gears, etc. are hidden from the user, and features such as the start button, gears, display, break, etc. are given to the user. When we perform any action on these features, the internal process works.

6. Encapsulation

In an object-oriented approach, encapsulation is a process of binding the data members (attributes) and methods together. The encapsulation restricts direct access to important data. The best example of the encapsulation concept is making a class where the data members are private, and methods are public to access through an object. In this case, only methods can access those private data.

Advantages of Java OOPs

The following are the advantages of using the OOPs in Java:

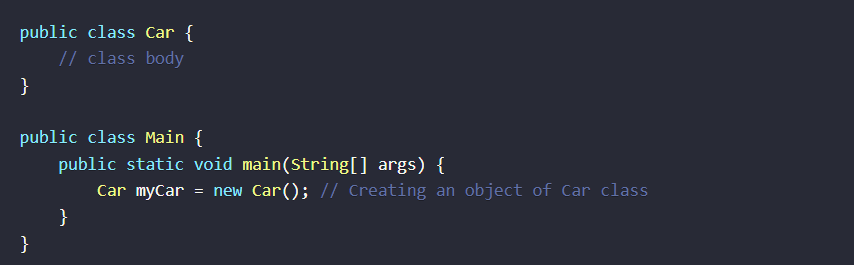
* The implementations of OOPs concepts are easier.
* The execution of the OOPs is faster than procedural-oriented programming.
* OOPs provide code reusability so that a programmer can reuse an existing code.
* OOPs help us to keep the important data hidden.

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and programs. These objects are created using classes, which are essentially blueprints for creating objects. In Java, OOP is a core concept, and understanding it is crucial for any Java developer. Here's a beginner-friendly tutorial to get you started with OOP in Java, including examples.

1. Understanding Objects and Classes

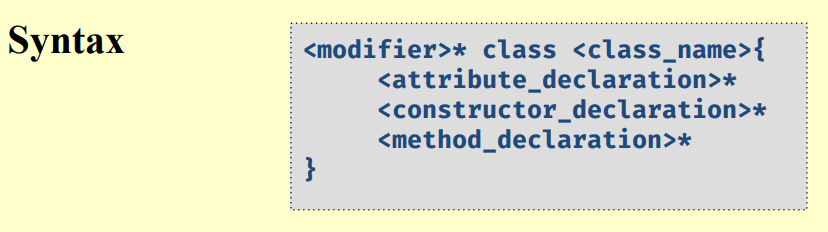
* Objects: In Java, everything is an object. An object is an instance of a class.
* Classes: A class is a blueprint or prototype from which objects are created.

Example:



2. Defining Classes

A class in Java is defined using the class keyword. It can have fields (variables) and methods (functions).



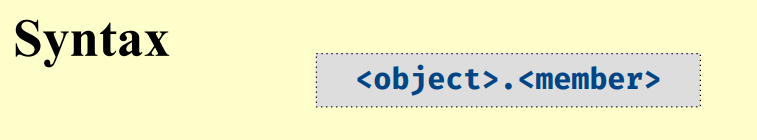
Example:

A computer screen shot of a computer code

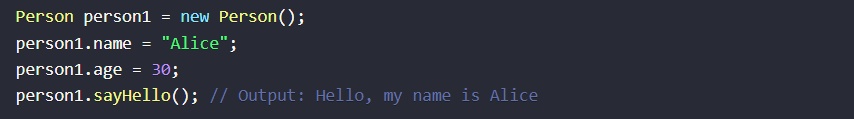
Description automatically generated

3. Creating Objects

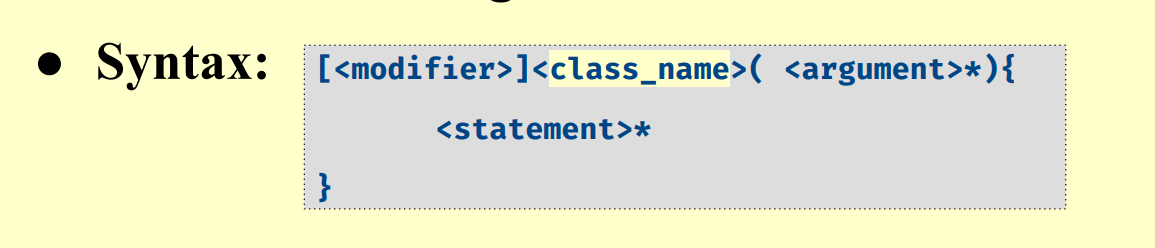
To create an object, use the new keyword followed by the class name and parentheses.



Example:



4. Constructors



A constructor is a special method used to initialize objects. It is called when an object of a class is created.

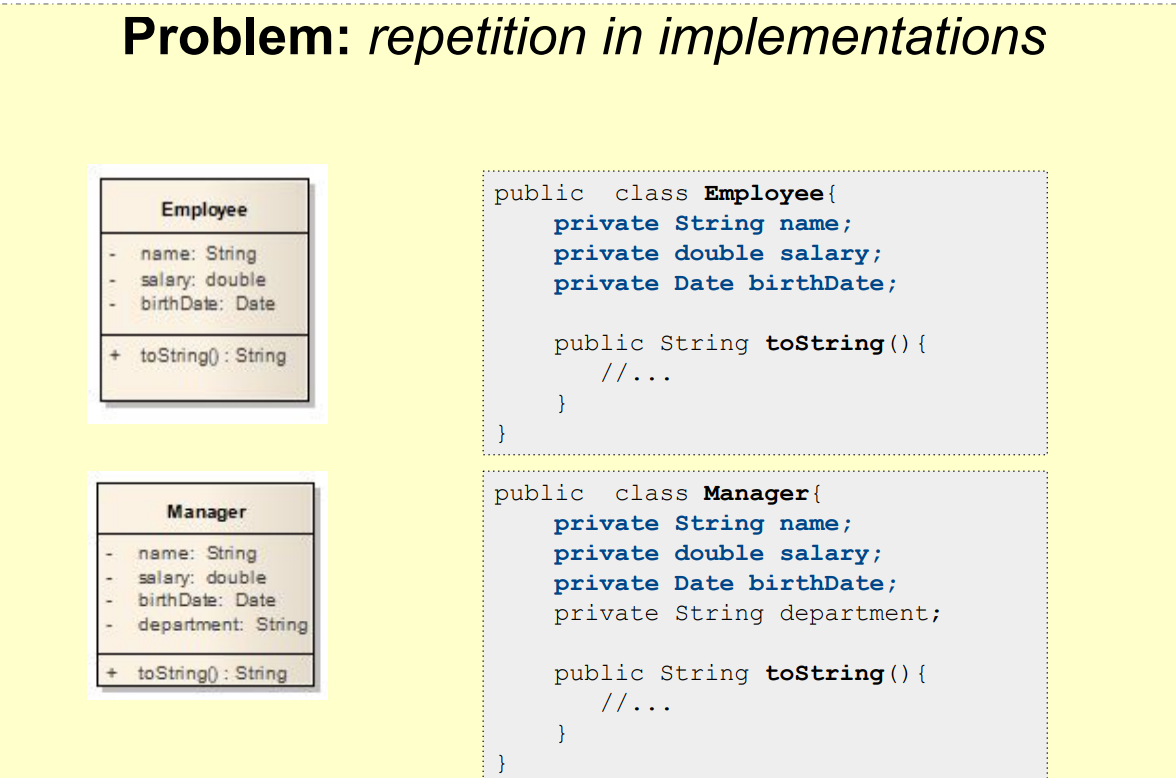
Example:

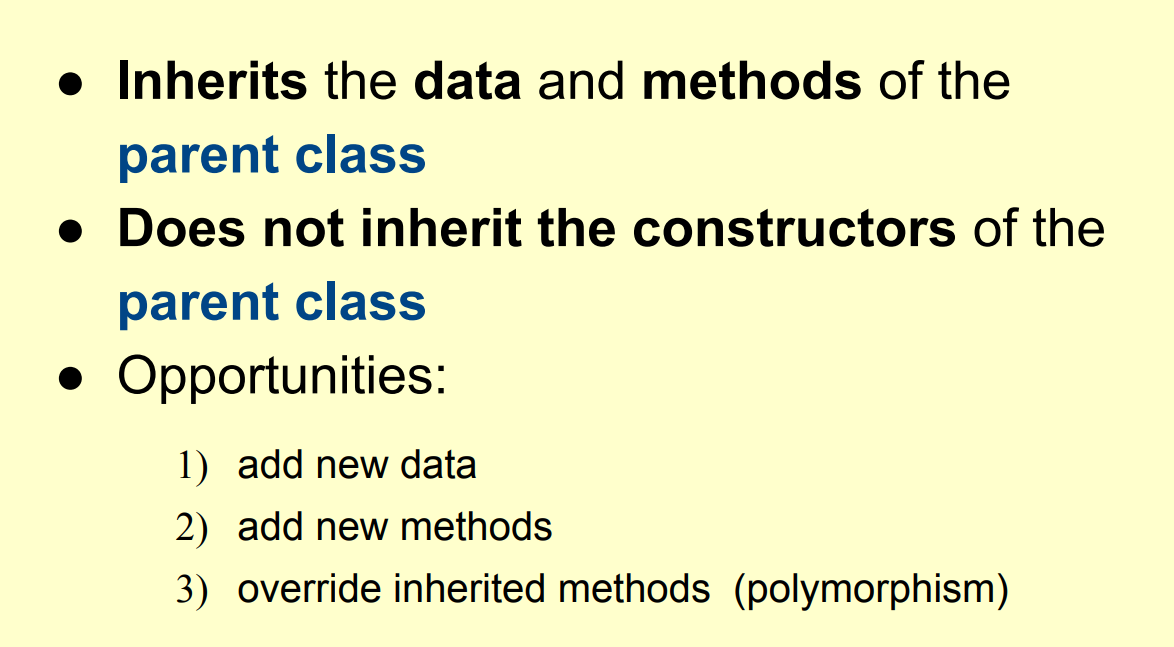
A computer screen shot of code

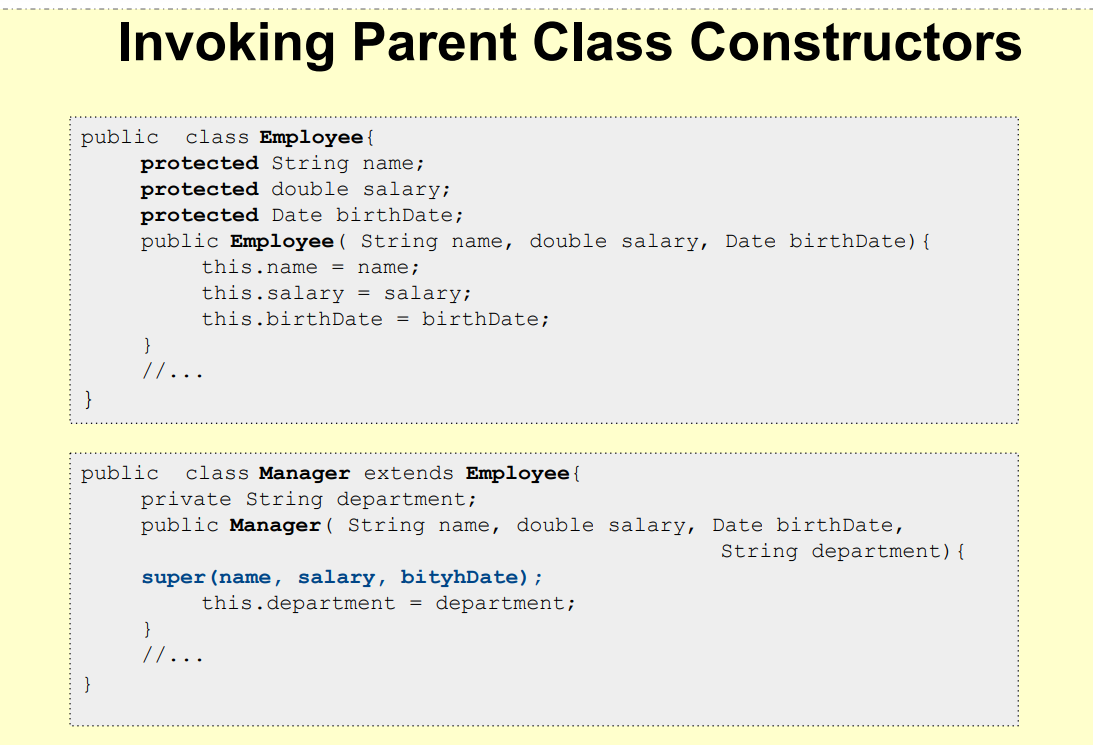
Description automatically generated

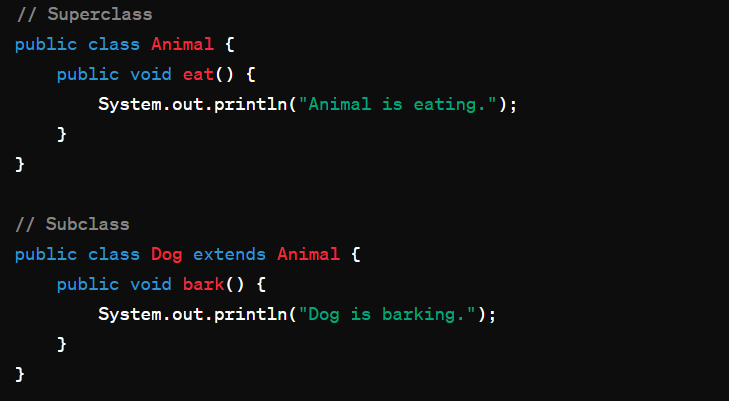
5. Inheritance

Inheritance is a mechanism where one class acquires the properties (fields) and behaviors (methods) of another class.







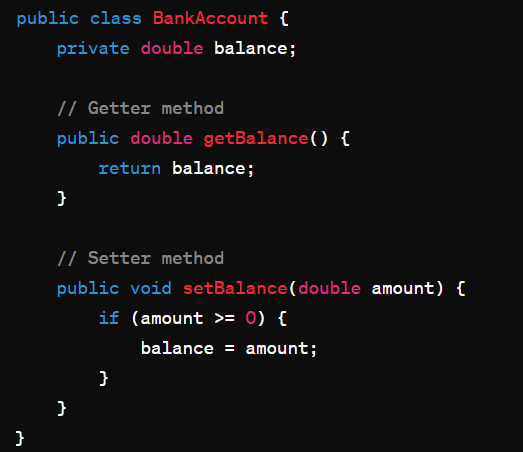
Example: 

6. Encapsulation

Encapsulation is the mechanism of hiding the data (variables) and methods within an object from the outside access.

Example: A white text and a yellow background

Description automatically generated with medium confidence



A screen shot of a computer program

Description automatically generated

7. Polymorphism

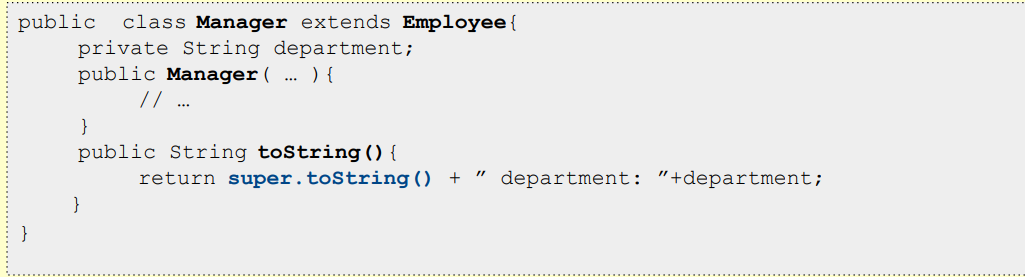
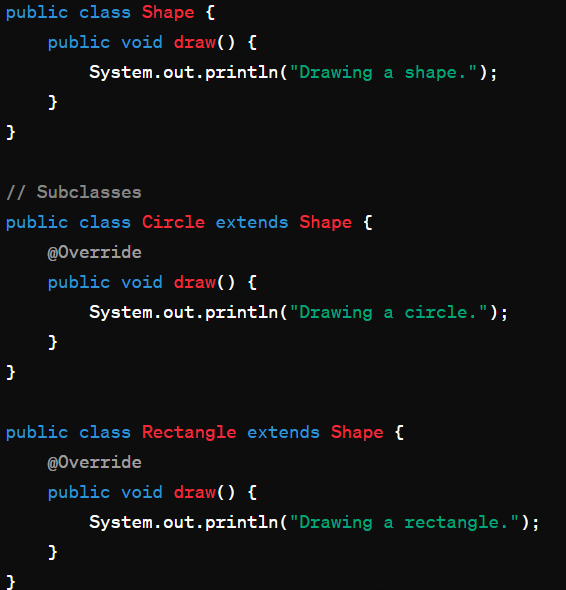
Polymorphism allows us to perform a single action in different ways.

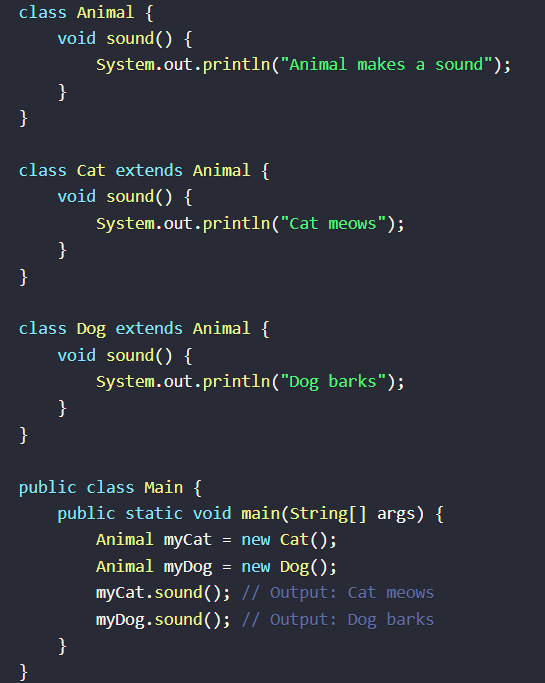
A white paper with black text

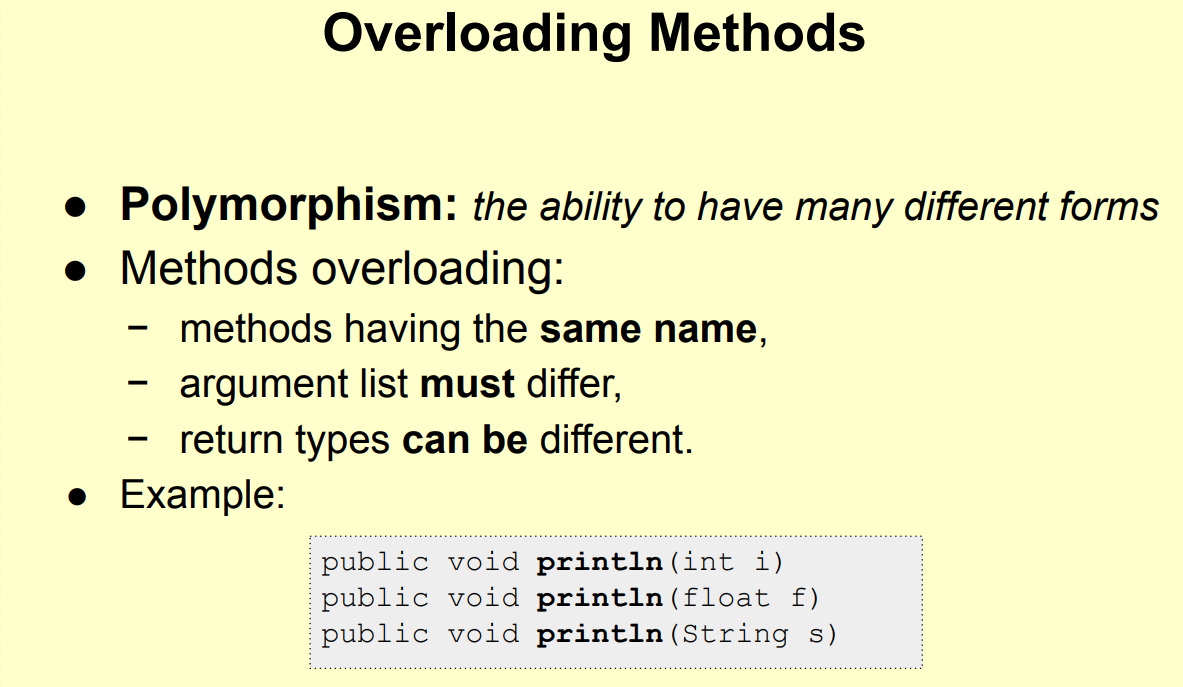
Description automatically generated

Example:



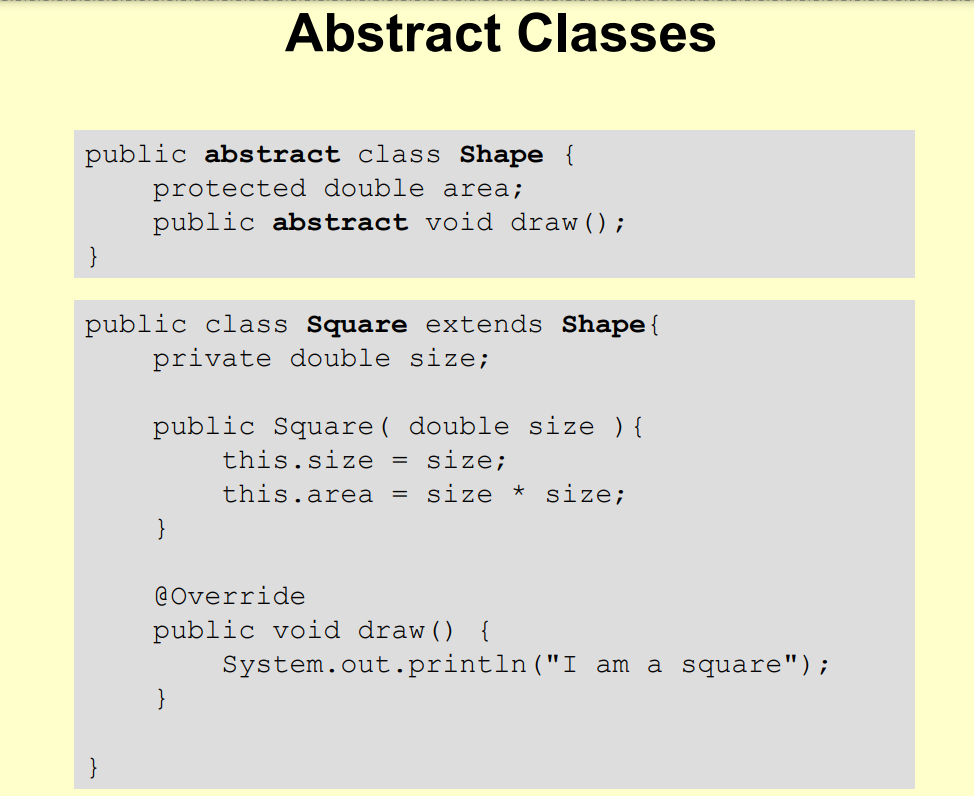
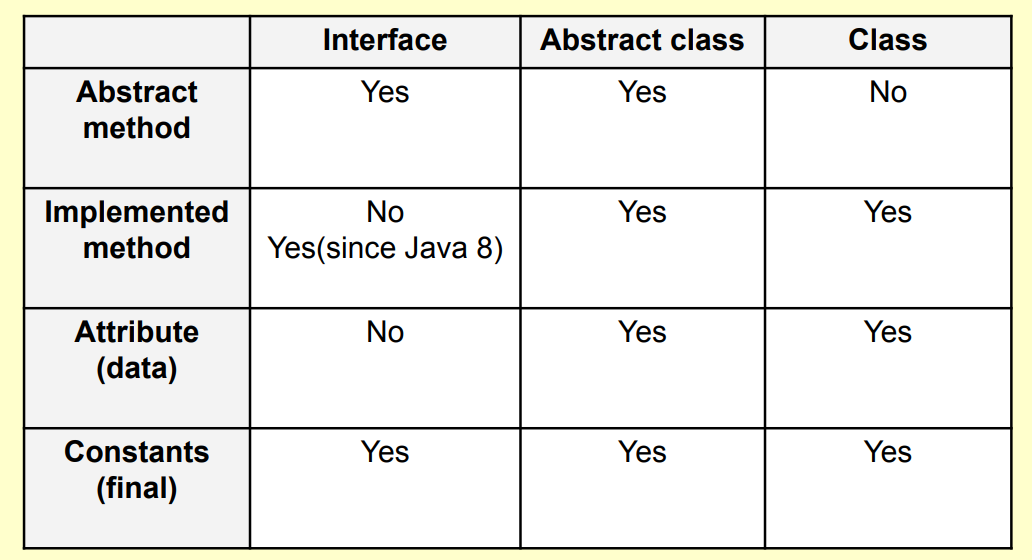
 





8. Abstraction

Abstraction is a process of hiding the implementation details and showing only the functionality to the user.

Example:  

A computer screen shot of text

Description automatically generated

Data Types in Java

Java has several primitive data types and one reference type.

* Primitive Data Types: These are the most basic types. They include:
* boolean: true or false
* byte: -128 to 127
* short: -32,768 to 32,767
* int: -2,147,483,648 to 2,147,483,647
* long: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
* float: 1.4E-45 to 3.4028235E38
* double: 4.9E-324 to 1.7976931348623157E308
* char: 0 to 65,535
* Reference Type: Strings are reference types. They are objects that represent sequences of characters.

Variables in Java

Variables are used to store data values. They are defined with a data type.

* Local Variables: Defined within a block or method.
* Instance Variables: Defined in a class outside of any method, constructor, or block.
* Static Variables: Also known as class variables, defined using the static keyword within a class.

Access Modifiers

Access modifiers define the visibility of a class, method, or variable.

* public: Accessible from any class.
* private: Accessible only within the class.
* protected: Accessible within the same package and subclasses.
* default (no keyword): Accessible within the same package.

A close-up of a sign

Description automatically generated

Operations and Logical Responses

Java supports arithmetic operations like addition, subtraction, multiplication, and division. Logical operations include AND (&&), OR (||), and NOT (!).

Arrays and Lists

* Arrays: A collection of elements of the same type.
* Lists: An interface in Java, part of the Java Collections Framework, used for storing groups of objects.

Static Members

Static members belong to the class itself rather than to any object. They are accessed using the class name.

Final Keyword

The final keyword is used to restrict the user. It can be applied to variables, methods, and classes.

* Final Variables: Once assigned, cannot be changed.
* Final Methods: Cannot be overridden by subclasses.
* Final Classes: Cannot be subclassed.

Enumerations

Enumerations are a feature in Java that allows you to define a type by a set of constants.

Constant Instance Data

Constant instance data is data that does not change once it is initialized. It is defined using the final keyword.

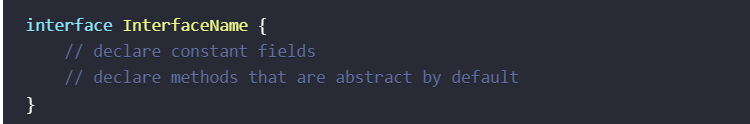
Relationships Between Classes

* Inheritance: One class inherits the properties and methods of another class.
* Composition: A class is composed of other classes.
* Aggregation: A class is made up of other classes, but they can exist independently.

Interfaces

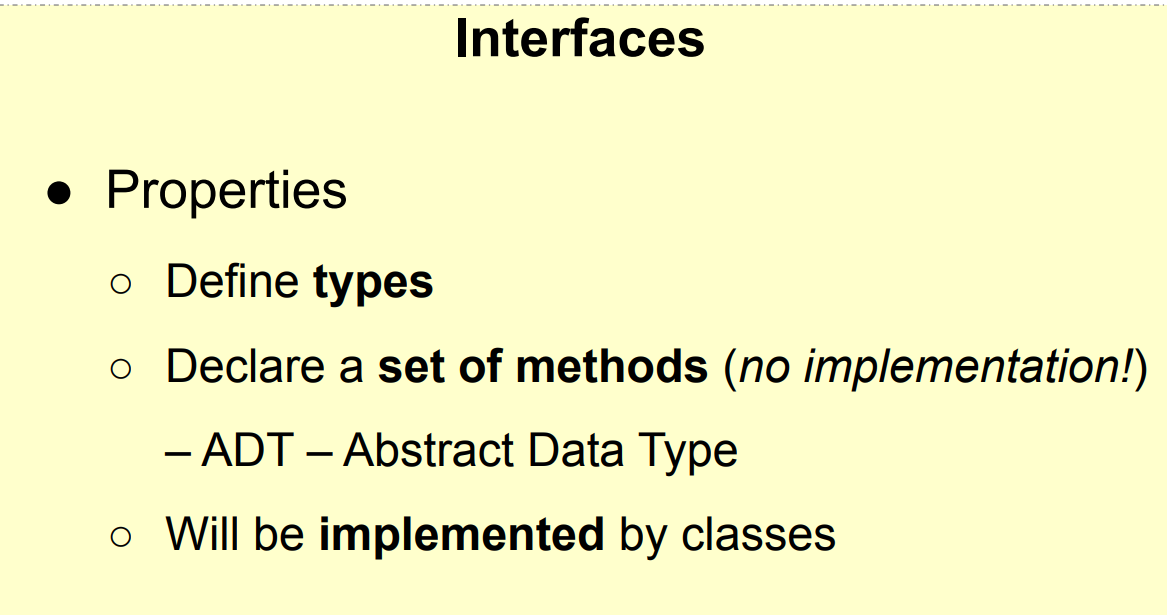
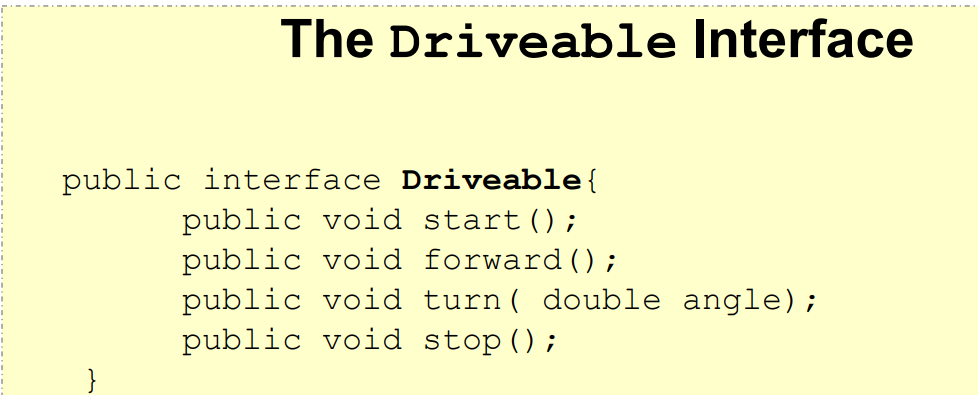
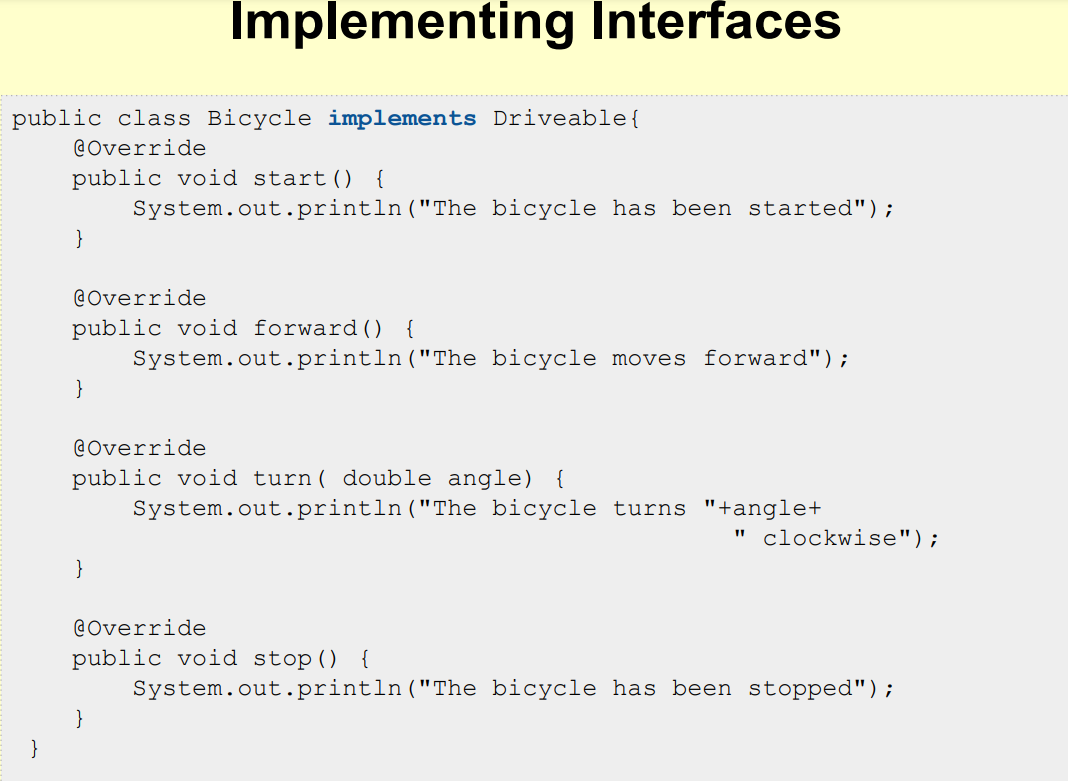
An interface in Java is a blueprint of a class or a contract for what a class can do, without saying anything about how the class will do it. Interfaces are used to achieve abstraction, polymorphism, and multiple inheritances in Java. They contain static constants and abstract methods, and they cannot have concrete methods (methods with an implementation).

To define an interface, use the interface keyword. All methods in an interface are implicitly abstract and public, and all variables are public, static, and final. A class that implements an interface must provide implementations for all the methods declared in the interface.



Rules for Creating Interfaces

* An interface can be empty, with no methods or variables in it.
* You can't instantiate interfaces directly.
* An interface method can't be protected or final.
* Up until Java 9, interface methods could not be private; however, Java 9 introduced the possibility to define private methods in interfaces.
* Interface variables are public, static, and final by definition; you're not allowed to change their visibility.

SOLID Principle

1. Single Responsibility Principle (SRP)

A class should have only one reason to change. It means that a class should only have one job or responsibility.

Example:

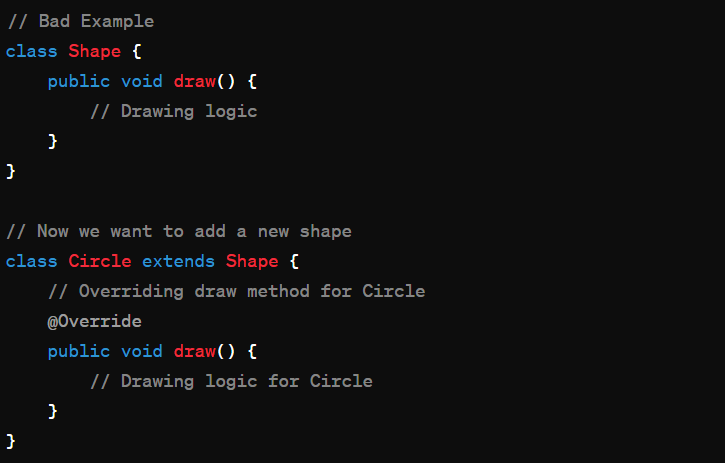


In this example, the **User** class violates SRP because it has two responsibilities: saving user and generating a report. It would be better to split these responsibilities into separate classes.

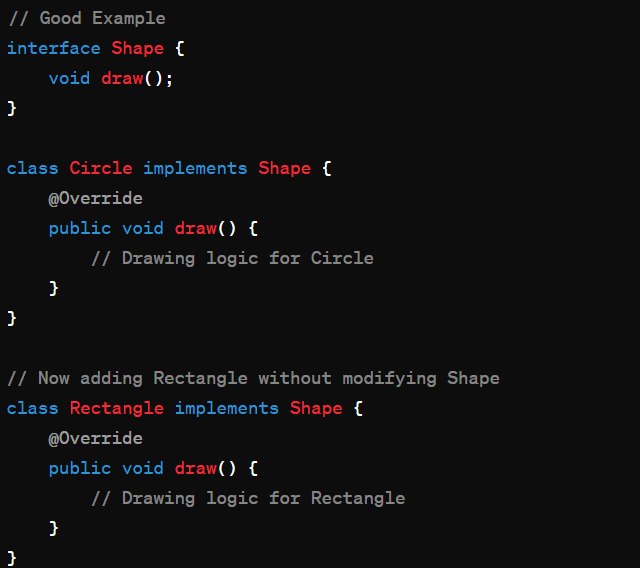
2. Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Example:



In this example, if we want to add a new shape, say **Rectangle**, we will need to modify the **Shape** class, which violates OCP. It's better to use abstraction and inheritance to make the system open for extension.



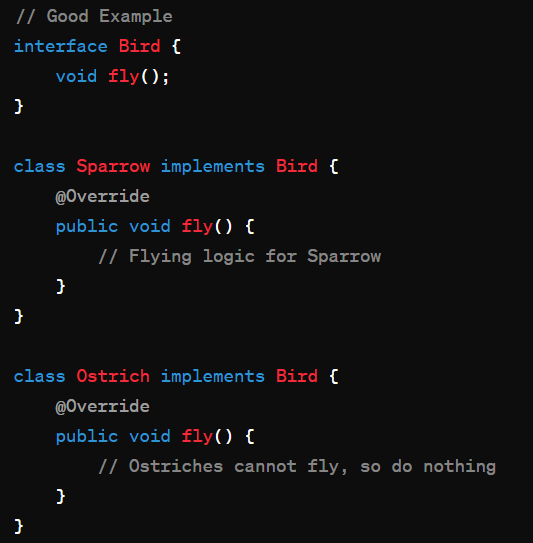
3. Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types without altering the correctness of the program.

Example:



In this example, the **Ostrich** class violates LSP because it cannot perform the behavior expected from its superclass **Bird**. It's better to follow the principle by making sure subclasses can substitute their superclasses without any unexpected behavior.



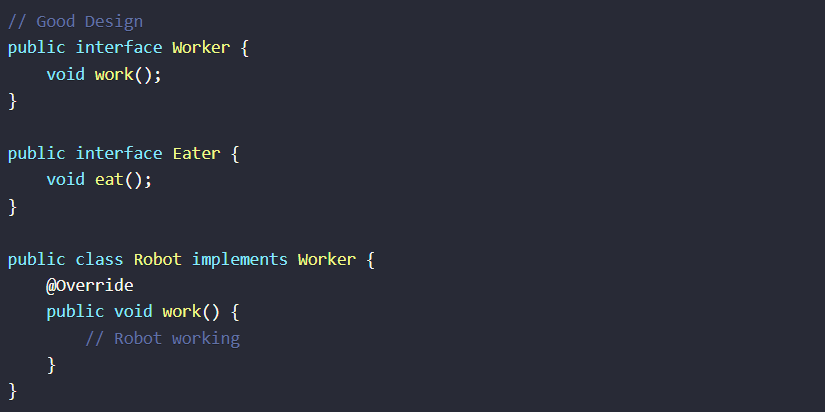
4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. This means that a class should not have to implement methods it doesn't use.

Example:



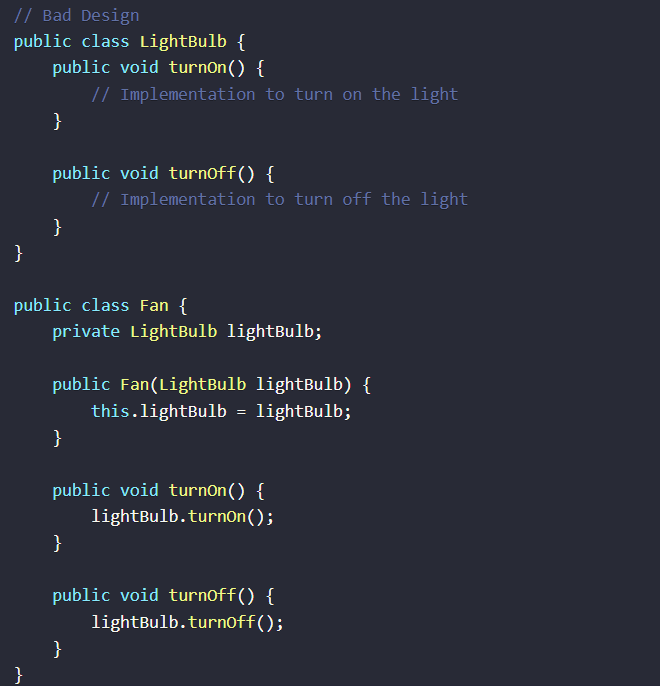
In this example, the **Robot** class is forced to implement both **work()** and **eat()** methods, even though it might not need the **eat()** method. It's better to segregate the interface into smaller, more specific interfaces.



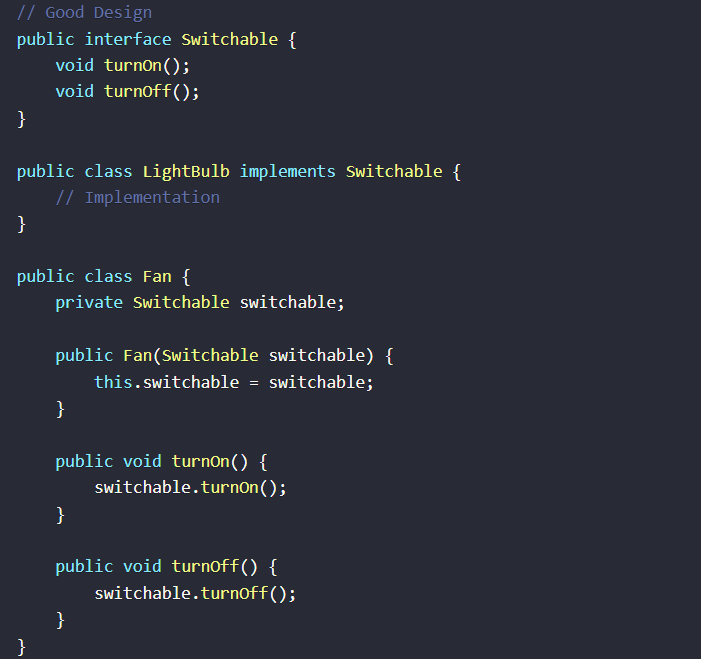
5. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Example:



In this example, the **Fan** class directly depends on the **LightBulb** class, violating DIP. It's better to introduce an abstraction so that both **Fan** and **LightBulb** depend on the abstraction rather than each other.



Requirements:

1. Classes: Define classes for Book, Author, and User.
2. Inheritance: Use inheritance to create a LibraryMember class that extends the User class.
3. Interfaces: Create an interface LibraryItem for items that can be borrowed.
4. Encapsulation: Use encapsulation to protect class fields.
5. Polymorphism: Implement polymorphism to allow different types of users to borrow items.
6. SOLID Principles: Apply SOLID principles to ensure the code is maintainable, understandable, and flexible.

Step 1: Define Basic Classes

Book.java

public class Book {

private String title;

private String isbn;

private Author author;

private boolean isAvailable;

public Book(String title, String isbn, Author author) {

this.title = title;

this.isbn = isbn;

this.author = author;

this.isAvailable = true;

}

public String getTitle() {

return title;

}

public String getIsbn() {

return isbn;

}

public Author getAuthor() {

return author;

}

public boolean isAvailable() {

return isAvailable;

}

public void borrow() {

if (isAvailable) {

isAvailable = false;

System.out.println("Book borrowed: " + title);

} else {

System.out.println("Book is not available.");

}

}

public void returnBook() {

isAvailable = true;

System.out.println("Book returned: " + title);

}

}

**Author.java**

public class Author {

private String name;

public Author(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**User.java**

public class User {

private String name;

private String libraryCardNumber;

public User(String name, String libraryCardNumber) {

this.name = name;

this.libraryCardNumber = libraryCardNumber;

}

public String getName() {

return name;

}

public String getLibraryCardNumber() {

return libraryCardNumber;

}

}

Step 2: Inheritance

LibraryMember.java

public class LibraryMember extends User {

private List<Book> borrowedBooks;

public LibraryMember(String name, String libraryCardNumber) {

super(name, libraryCardNumber);

this.borrowedBooks = new ArrayList<>();

}

public void borrowBook(Book book) {

if (book.isAvailable()) {

book.borrow();

borrowedBooks.add(book);

System.out.println(getName() + " borrowed " + book.getTitle());

} else {

System.out.println("Book is not available.");

}

}

public void returnBook(Book book) {

if (borrowedBooks.contains(book)) {

book.returnBook();

borrowedBooks.remove(book);

System.out.println(getName() + " returned " + book.getTitle());

} else {

System.out.println("You didn't borrow this book.");

}

}

}

Step 3: Interface

LibraryItem.java

public interface LibraryItem {

void borrow();

void returnItem();

}

**Book.java** (updated to implement LibraryItem)

public class Book implements LibraryItem {

// Existing fields and methods

@Override

public void borrow() {

// Implementation

}

@Override

public void returnItem() {

returnBook();

}

}