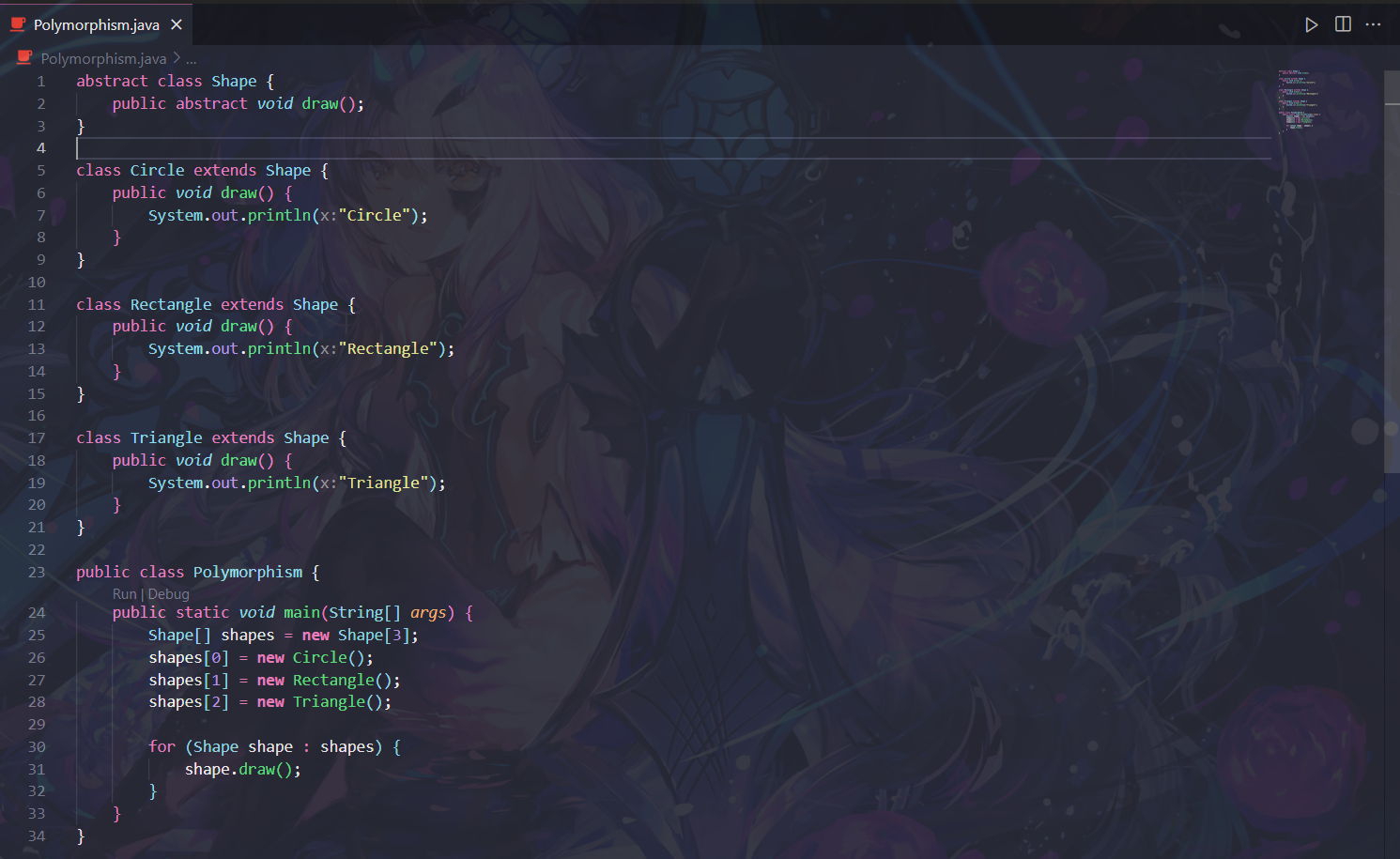
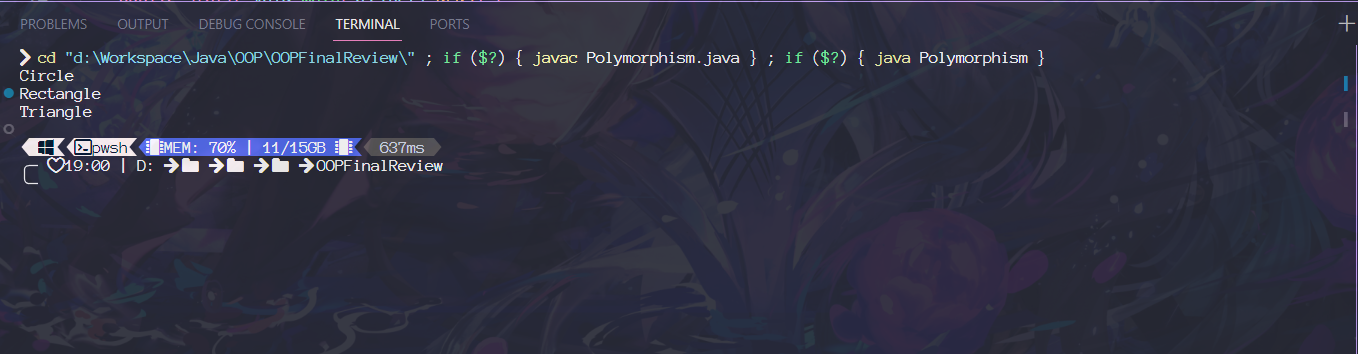
1. Polymorphisms

A concept that describes situations in which something occurs in several different forms. It’s a concept that you can access objects of different types through the same superclass. Polymorphism is achieved through method **overloading** and method **overriding**:

* Method **overloading**: When multiple methods have the same name but different parameters.
* Method **overriding**: When a subclass provides its own implementation of a method that is already present in its parent class.

For example, a program that has a *Shape* class and several subclasses such as *Circle*, *Rectangle,* and *Triangle*. Each of them has its own implementation of the *draw()* method. By using polymorphism, we can write a method that takes an object of the Shape class as a parameter and call its *draw()* method. This method will work correctly regardless of whether the object is a *Circle*, *Rectangle*, or *Triangle*.





1. Interfaces & Abstract classes

Abstraction in OOP is the process of hiding the complex implementation details of a program, exposing only the essential features or behaviors to the user. Abstraction can be achieved by using interfaces and abstract classes.

Interface is a blueprint that defines a set of methods that a class must implement. Interfaces are useful because they provide a standard way to define the behavior of a group of related classes.

Interfaces help you standardize all the methods in it, so that any time you want something to accept many kinds of classes, you can do so by accessing the interface methods. It could be think of a replacement for base classes since you can only have one base class but you can implement any number of interfaces.

An abstract class is a class that cannot be instantiated but can be used as a superclass for inheritance.

An abstract method is a method that has no implementation but must be overridden by subclasses that inherit from the abstract class. An abstract class can have both abstract and concrete methods, as well as instance variables and constructors.

It can be used to define a common interface or behavior for a group of subclasses that share some characteristics but differ in some details.

Differences between Abstract class and Interface

| **Abstract class** | **Interface** |
| --- | --- |
| 1) Abstract classes can **have abstract and non-abstract** methods. | Interfaces can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static, and non-static variables**. | Interfaces have **only static and final variables**. |
| 4) Abstract class **can provide the implementation of an interface**. | Interface **can't provide the implementation of an abstract class**. |
| 5) The **abstract keyword** is used to declare an abstract class. | The **interface keyword** is used to declare an interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using the keyword "extends". | An **interface** can be implemented using the keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:**  public abstract class Shape{  public abstract void draw();  } | **Example:**  public interface Drawable{  void draw();  } |

1. SOLID

An acronym for five principles of OOP that help developers design software that is easy to understand, maintain, and extend. Those principles are:

**S**ingle Responsibility: A class should have only one reason to change, meaning that it should have only one job or responsibility.

**O**pen-Close: A class should be open for extension, but closed for modification, meaning that it should allow adding new features without changing the existing code.\

**L**iskov Substitution: A subclass should be able to replace its superclass without affecting the functionality of the program, meaning that the subclass should follow the contract of the superclass.

**I**nterface Segregation: A class should not depend on methods that it does not use, meaning that it should have multiple specific interfaces rather than one general interface.

**D**ependency Inversion: A class should depend on abstractions rather than concretions, meaning that it should rely on interfaces or abstract classes rather than concrete classes.

1. Generics in Java

Generics are a way to parameterize types, meaning that you can specify the type of objects that a class, method, or interface can work with. Generics enable you to write code that is more type-safe, flexible, and reusable.

For example, you can use generics to create a generic collection class that can store any type of object, instead of creating separate classes for each type.

Some benefits:

* Avoid casting objects, which can cause runtime errors if done incorrectly.
* Catch type errors at compile time, which makes debugging easier.
* Write generic algorithms that can work with different types of objects, without repeating code.

**tl;dr:** Generic methods and generic classes (and interfaces) enable you to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows you to catch invalid types at compile-time

1. Exception Handling

Exception handling is a mechanism to handle runtime errors and maintain the normal flow of the application.

An exception is an event that disrupts the normal flow of the program and creates an exception object that contains information about the errors. It uses the try-catch block to monitor for exceptional conditions and transfer control to a special exception handling code.

Exceptions can be categorized into two types: checked and unchecked. Checked exceptions are checked at compile-time and must be handled or declared while Unchecked exceptions are not checked at compile-time and are usually caused by programming errors.

Exceptions can also be user-defined by extending the Exception class or its subclasses, and following a hierarchy that is derived from the Throwable class, which is the base class for all errors and exceptions.