Java Programming for Complete Beginners:

This video course of duration 27hr 6min 22sec. And the sections included in this are briefly mentioned below. Eclipse used for compiling Java files.

Introduction to JAVA 11 and OOPS Concepts:

Java is a popular programming language that is widely used for developing applications ranging from small desktop utilities to large-scale enterprise systems. Java 11 is the latest long-term support (LTS) release of Java, which was introduced in September 2018. It includes various new features and improvements, such as support for the HTTP client API, dynamic class-file constants, and improved garbage collection, among others.

Data Types in Java:

Java has eight primitive data types: byte, short, int, long, float, double, char, and boolean. These data types are used to hold different types of values, such as numbers, characters, and true or false values. Java also has reference data types, such as String, which are used to refer to objects created from classes. Additionally, Java supports arrays, which are used to store multiple values of the same data type in a single variable. When writing Java code, it is important to choose the appropriate data type for each variable to ensure that it can hold the necessary information and to optimize the performance of the program.

Operators in java:

Java provides a wide range of operators to perform various operations on variables and values. Arithmetic operators, such as addition and multiplication, are used to perform mathematical calculations. Logical operators, such as AND and OR, are used to combine conditions in conditional statements. Java also provides comparison operators, such as greater than and less than, to compare values and make decisions based on the result. Additionally, Java has bitwise operators, which are used to perform operations on binary values. Understanding and using these operators effectively is an important part of writing efficient and effective Java code.

Java 11 Garbage collector:

Java 11 introduces a new experimental Garbage Collector (GC) called the Z Garbage Collector (ZGC), which is designed to handle very large heap sizes and low pause times. The ZGC is a concurrent, compacting, and low-latency GC, which means it works while the application is running, rearranging and freeing memory as needed without stopping the application for long periods.

This garbage collector is designed to handle large heaps (up to 16 terabytes) with very low pause times (less than 10 milliseconds). It uses a concurrent garbage collection algorithm and is designed to work well with modern hardware. It is particularly well-suited for applications that require large amounts of memory, such as big data and machine learning applications.

Object Oriented Programming System (OOPS):

Java 11 is an object-oriented programming language that supports various Object-Oriented Programming (OOPs) concepts. By using these OOPs concepts, developers can create modular, reusable, and extensible code, which is easier to maintain and modify over time.



One of the key OOPs concepts in Java 11 is encapsulation. Encapsulation is the process of wrapping data and behavior within a single unit, known as a class, and restricting direct access to the data. In Java, encapsulation is achieved through access modifiers such as public, private, protected, and default.

Another important OOPs concept in Java 11 is inheritance. Inheritance is the process of deriving new classes from existing ones, which helps in reusing code and creating class hierarchies. In Java, inheritance is achieved through the **extends** keyword.

Polymorphism is another important OOPs concept in Java 11. Polymorphism is the ability of an object to take on multiple forms, which helps in creating flexible and extensible code. In Java, polymorphism is achieved through method overloading and method overriding.

1. Method Overriding: Method overriding is the process of providing a new implementation for an inherited method in a subclass. In Java, method overriding is achieved by defining a method with the same name and signature as the superclass method in the subclass.
2. Method Overloading: Method overloading is the process of defining multiple methods with the same name but different parameters. In Java, method overloading is achieved by defining methods with the same name but different parameter lists.

Abstraction is achieved in Java 11 through abstract classes and interfaces. Abstract classes are classes that cannot be instantiated and are typically used as a base class for other classes. They may contain both abstract and non-abstract methods, and abstract methods have no implementation and must be overridden in subclasses. Abstract classes may also contain instance variables and constructors, which can be used by subclasses.

Interfaces, on the other hand, are similar to abstract classes, but they only contain abstract methods and constants. They cannot have instance variables or constructors, and they cannot be instantiated. Interfaces are typically used to define a set of behaviors that a class must implement. A class can implement multiple interfaces, but can only extend one class.

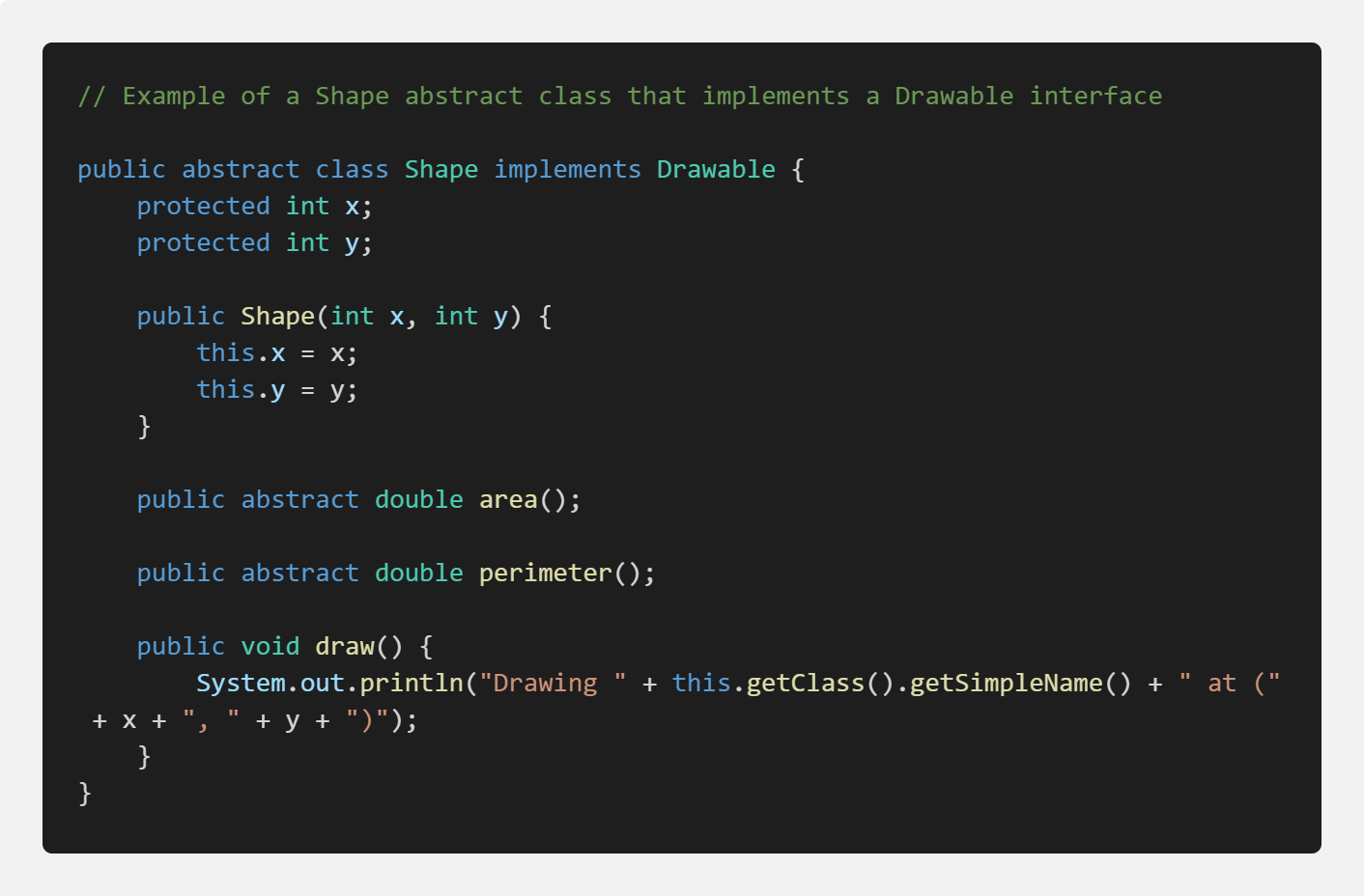
By using abstract classes and interfaces, developers can define a set of common methods and behaviors that can be shared by multiple classes. They can also promote modularity, reusability, and extensibility. This allows developers to create code that is more flexible and scalable, and easier to maintain and modify over time.

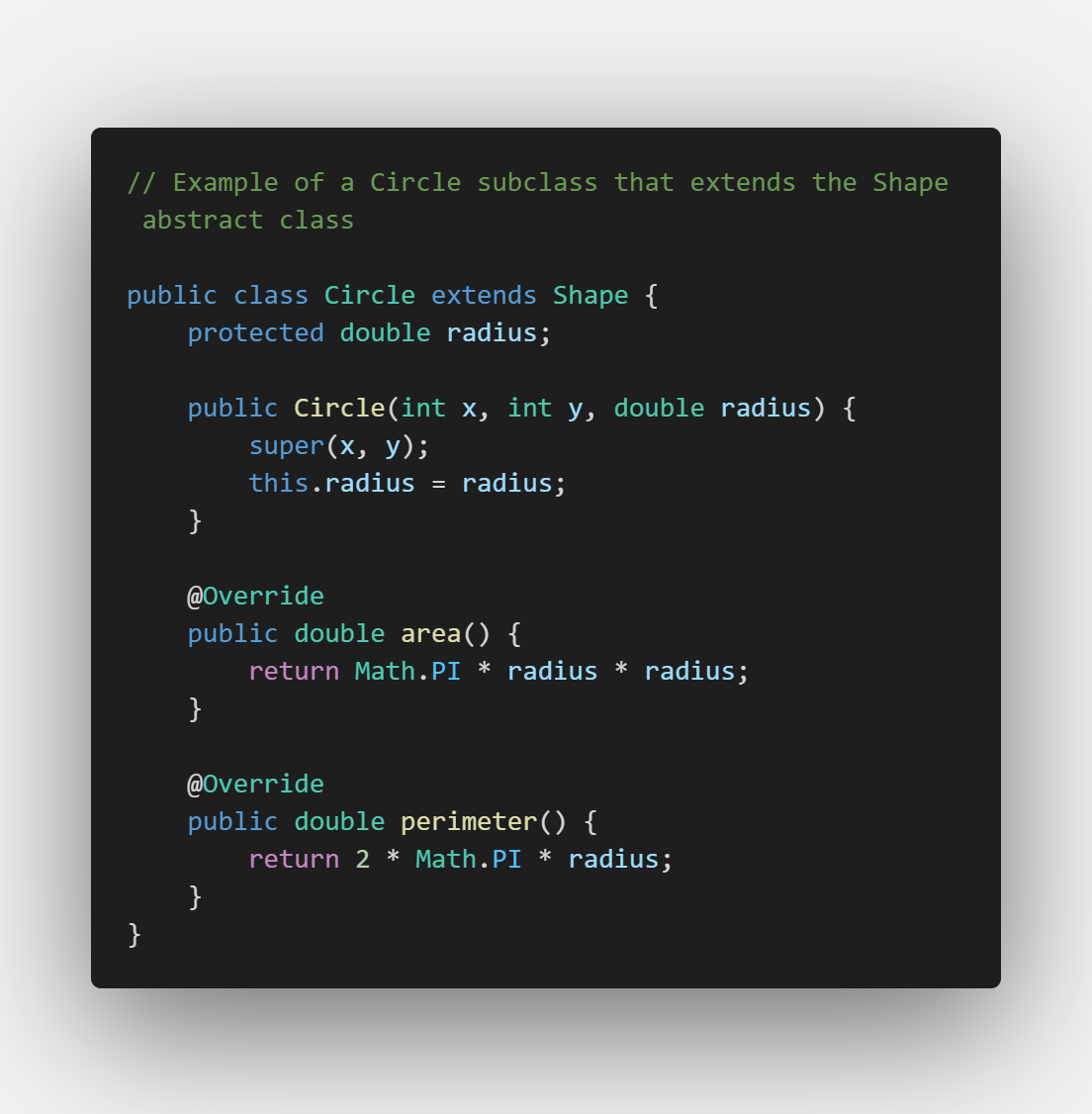
In addition to abstract classes and interfaces, Java 11 also supports other forms of abstraction, such as access modifiers (public, private, protected, and default), which allow developers to control the visibility and accessibility of class members.

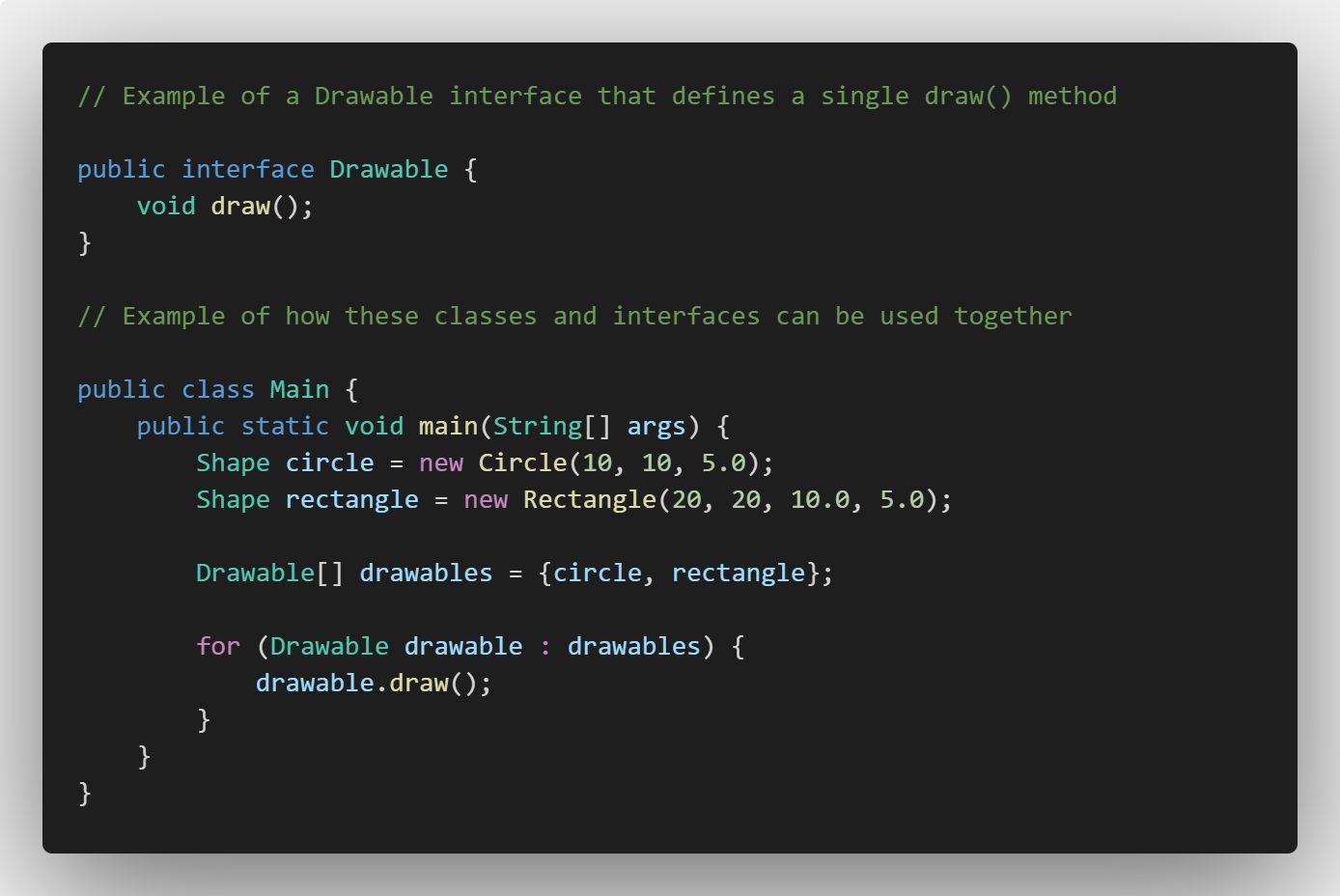
Lastly, class and objects are fundamental OOPs concepts in Java 11. A class is a blueprint or template for creating objects, which are instances of the class. In Java, classes are defined using the class keyword.

By using these OOPs concepts in Java 11, developers can create modular, reusable, and extensible code that is easier to maintain and modify over time. These concepts also help in creating code that is more flexible, robust, and scalable, which can lead to higher quality software applications.

Here is an example of implementation java oops concepts





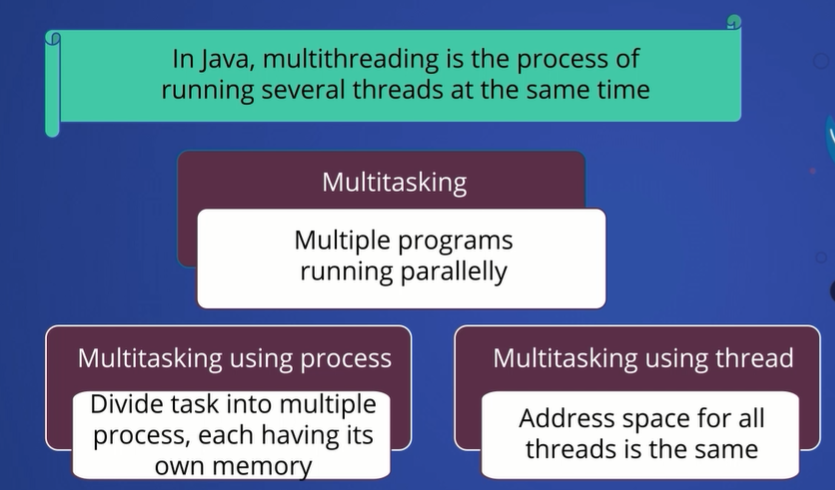


In this example, the **Shape** abstract class defines common properties and methods for different shapes, while the **Circle** and **Rectangle** subclasses implement their own specific **area()** and **perimeter()** methods. The **Drawable** interface defines a **draw()** method that is implemented by both the **Shape** abstract class and its subclasses. Finally, the **Main** class shows how these classes and interfaces can be used together to create a list of drawable shapes and display them using the **draw()** method. This example demonstrates the use of inheritance, abstraction, polymorphism, and interfaces in Java 11 OOPs concepts.

Threading in JAVA:

In Java 11, multithreading is an important concept that allows developers to create programs that can perform multiple tasks concurrently, increasing performance and efficiency. Here are some of the key multithreading concepts in Java 11:

Thread: A thread is a lightweight unit of execution that runs concurrently with other threads. Threads are created using the **Thread** class, and can be used to perform tasks in the background while the main program continues to run.



In Java, there are two ways to create threads: by implementing the `Runnable` interface and by extending the `Thread` class. Here are the main differences between the two approaches:

1. Extending the Thread class: When you extend the `Thread` class, your class is a thread and a new instance of your class will be a new thread. However, this approach has a drawback that you can only extend one class in Java, which means you can't extend any other class if you want to create a thread.

Example of thread creation using extends:



In this example, we define a new class called **MyThread** that extends the **Thread** class. The **run()** method is overridden to define the thread's logic, which in this case simply prints a message to the console.

In the **Main** class, we create a new instance of **MyThread** and call its **start()** method to start the thread. When the thread is started, it will run its **run()** method and print the "Hello from MyThread!" message to the console.

2. Implementing the Runnable interface: When you implement the `Runnable` interface, you can still extend other classes if needed, because your class does not become a thread. Instead, you need to create a new `Thread` object and pass your `Runnable` implementation to its constructor.



In this example, we define a new class called **MyRunnable** that implements the **Runnable** interface. The **run()** method is overridden to define the thread's logic, which in this case simply prints a message to the console.

In the **Main** class, we create a new instance of **MyRunnable** and pass it to the constructor of a new **Thread** object. We then call the **start()** method on the **Thread** object to start the thread. When the thread is started, it will run the **run()** method of the **MyRunnable** object and print the "Hello from MyRunnable!" message to the console.

3. Thread safety: Implementing the `Runnable` interface is considered more thread-safe, because it separates the thread logic from the thread creation. In the case of extending the `Thread` class, the thread and its logic are tightly coupled, which can lead to thread-safety issues.

In general, it is recommended to use the `Runnable` interface to create threads in Java, unless you need to extend other classes. By using the `Runnable` interface, you can separate the thread logic from the thread creation, which can make your code more modular and easier to maintain.