Packages- Defining a Package, CLASSPATH, Access protection, importing packages. Interfaces- defining an interface, , Nested interfaces, applying interfaces, variables in interfaces and extending interfaces. **Stream based I/O (java.io)–** The Stream classes-Byte streams and Character streams, Reading console Input and Writing Console Output, File class, Reading and writing Files, Random access file operations, The Console class, Serialization, Enumerations, auto boxing, generics.

Package

Defining Packages In Java

In Java, a package is a way to organize and group related classes, interfaces, and other resources. Packages provide a mechanism for namespace management, encapsulation, and access control in Java programs. Here are some important points about packages in Java:

1. Package Declaration: At the top of each Java source file, you typically include a package declaration that specifies the package to which the file belongs. It is defined using the `package` keyword followed by the package name. For example:

   ```
   package com.example.myapp;
   ```

   The package name conventionally uses a reversed domain name to ensure uniqueness.

2. Package Structure: Packages are organized hierarchically, forming a directory structure on the file system. Each level of the package hierarchy corresponds to a subdirectory. For example, the package `com.example.myapp` would be stored in the directory structure `com/example/myapp/`.
3. Package Import: To use classes from another package, you need to import them using the `import` statement. You can import a specific class or import all classes within a package using the wildcard (`*`). For example:

   ```
   import com.example.otherpackage.SomeClass;

   import java.util.*;
   ```

4. Default Package: If you don't specify a package for a class, it becomes part of the default package, which has no explicit name. However, it's considered a good practice to always define packages for your classes.
5. Access Modifiers: Packages provide a level of access control through the use of access modifiers like `public`, `protected`, and `private`. Classes and members (methods, fields, inner classes) can be declared with different access levels to control their visibility within and outside the package.
6. JAR Files: Java Archive (JAR) files are used to bundle multiple compiled classes and resources from a package into a single file. JAR files are a common way to distribute and deploy Java libraries or applications.

Packages play a crucial role in organizing and managing large-scale Java projects. They help avoid naming conflicts, provide structure, and enhance code maintainability and reusability.

CLASSPATH

In Java, the CLASSPATH is an environment variable that specifies the directories or JAR files where Java should look for classes and resources at runtime. It tells the Java Virtual Machine (JVM) where to find

the necessary dependencies for executing a Java program. Here are some important points about the CLASSPATH in Java:

1. Classpath Entries: The CLASSPATH consists of multiple entries, each representing a directory or a JAR file. Multiple entries are separated by the platform-specific path separator character (":" on Unix-like systems, ";" on Windows).
2. Default Classpath: By default, Java automatically includes the current directory (where the program is executed) in the classpath. This means that classes in the current directory can be directly referenced without the need for explicit classpath configuration.

Setting the Classpath: The CLASSPATH can be set in several ways:

3. Command Line: You can set the classpath using the `-cp` or `-classpath` option when executing a Java program. For example:

java -cp /path/to/classes:/path/to/library.jar MyClass

Environment Variable: You can set the CLASSPATH environment variable using the command specific to your operating system. For example, on Unix-like systems:

export CLASSPATH="/path/to/classes:/path/to/library.jar"

Manifest File: When creating JAR files, you can specify the classpath in the JAR's manifest file (`META-INF/MANIFEST.MF`). The manifest file can contain a line like this:

Class-Path: /path/to/classes/ library.jar

IDE Configuration: Integrated Development Environments (IDEs) often provide a way to configure the classpath for a project or module. This allows you to manage dependencies within the IDE without explicitly setting the classpath.

1. Classpath Resolution: When a Java program is executed, the JVM searches for classes and resources in the specified classpath entries. It looks for the required classes in the order specified in the classpath. If a class or resource is found in one of the entries, it is loaded and used by the program.
2. Classpath Priority: If multiple classpath entries contain classes or resources with the same name, the JVM loads the one that appears first in the classpath. Therefore, the order of entries in the classpath can affect which classes are used.
3. Classpath Wildcards: Instead of specifying each JAR file or directory individually, you can use a wildcard (*****) to include all JAR files or classes within a directory. For example:

Access protection in java

In Java, access protection refers to the mechanisms that control the visibility and accessibility of classes, methods, and fields within a program. Access modifiers are used to specify the level of access for different elements, allowing you to control how they can be accessed from other parts of the program or from external code. Java provides four access modifiers:

1. `public`: The `public` modifier provides the highest level of accessibility. Public classes, methods, and fields can be accessed from anywhere, including other classes in the same package, classes in different packages, and even from external code.

2. `protected`: The `protected` modifier allows access from within the same package and from subclasses, even if they are in different packages. This modifier is often used when you want to provide limited access to certain members while still maintaining encapsulation.
3. (default): If no access modifier is specified, it is considered the default access level. Default access allows access within the same package but restricts access from classes in different packages. Classes, methods, and fields with default access are not accessible from external code.
4. `private`: The `private` modifier provides the most restrictive level of access. Private members can only be accessed within the same class. They are not accessible from subclasses or any other classes, even if they are in the same package.

The following table summarizes the access levels for different access modifiers:

Access Modifier	Same Class	Same Package	Subclass	Different Package
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
(default)	Yes	Yes	No	No
`private`	Yes	No	No	No

By using access modifiers appropriately, you can enforce encapsulation, limit the visibility of implementation details, and define clear boundaries for your code. It is considered good practice to use the most restrictive access level that still allows other parts of the program to access what they need. This helps in maintaining code integrity, reducing dependencies, and facilitating easier maintenance and modification.

Importing Packages In Java

In Java, importing packages allows you to use classes, interfaces, and other types defined in external packages in your Java program. By importing packages, you can reference the classes directly by their simple names instead of specifying the fully qualified names (package name + class name) every time. Here's how you can import packages in Java:

1. Importing a Specific Class: If you want to import a specific class from a package, you can use the `import` statement followed by the fully qualified name of the class. For example:

```
import java.util.ArrayList;

import java.time.LocalDate;
```

With these imports, you can directly use the `ArrayList` and `LocalDate` classes in your code.

2. Importing All Classes from a Package: Instead of importing individual classes, you can import all the classes within a package using the wildcard (`*`). This is useful when you need to use multiple classes from the same package. For example:

```
import java.util.*;
```

This import statement allows you to use any class from the `java.util` package without explicitly specifying the package name.

3. Importing Nested Packages: If a package contains nested subpackages, you can import them using the dot notation. For example:

```
import com.example.myapp.util.*;
```

This import statement imports all classes from the `com.example.myapp.util` package

and its subpackages.

4. Static Import: In addition to importing classes, you can also import static members (fields and methods) of a class using the `import static` statement. This allows you to use those static members without qualifying them with the class name. For example:

```
import static java.lang.Math.PI;

import static java.lang.Math.sqrt;
```

With these static imports, you can directly use the constants `PI` and the method `sqrt()` from the `Math` class.

It's important to note that if you don't import a class explicitly, you can still use it by specifying the fully qualified name (package name + class name) whenever you reference it in your code. However, importing packages and classes makes the code more concise and readable.

By organizing your imports properly, you can avoid naming conflicts, improve code clarity, and make it easier to maintain and understand your Java programs.

Interfaces

Defining an interface in java:

In Java, an interface is a reference type that defines a contract for classes to implement. It specifies a set of methods that must be implemented by any class that claims to implement the interface.

An interface declaration in Java typically includes the following components:

1. Interface Keyword: The `interface` keyword is used to declare an interface in Java.
2. Interface Name: You provide a name for the interface, following the same naming conventions as class names. By convention, interface names are typically nouns or noun phrases representing the behavior or capabilities defined by the interface. For example:

```
public interface Printable {
    // Interface methods will be defined here
}
```

3. Interface Methods: Inside the interface, you define the methods that classes implementing the interface must implement. These methods are declared without a body (no implementation) and are by default `public` and `abstract`. For example:

```
public interface Printable {
```

```
    void print();
    void getInfo();
}
```

In this example, the `Printable` interface defines two methods: `print()` and `getInfo()`. Any class implementing this interface must provide an implementation for these methods.

4. Constants: Interfaces can also define constants, which are implicitly `public`, `static`, and `final`. Constants are typically used to define shared values or configuration settings. For example:

```
public interface Constants {
    int MAX_SIZE = 100;
    String DEFAULT_NAME = "Untitled";
}
```

In this case, the `Constants` interface defines two constants: `MAX_SIZE` and `DEFAULT_NAME`.

5. Default Methods (Java 8 and later): Starting from Java 8, interfaces can have default methods. A default method is a method with an implementation in the interface itself. It provides a default behavior that implementing classes can use. Default methods are declared using the `default` keyword. For example:

```
public interface Printable {
    void print();
      default void log() {
      System.out.println("Logging...");
    }
}
```

In this example, the `Printable` interface has a default method `log()`, which is automatically available to implementing classes.

Interfaces play a crucial role in Java's implementation of abstraction and polymorphism. They provide a way to define contracts for classes, enabling loose coupling and flexibility in designing and implementing software components. Classes can implement multiple interfaces, allowing them to exhibit behavior defined by multiple interfaces simultaneously.

To implement an interface in a class, you use the `implements` keyword followed by the interface name(s). The class must provide an implementation for all the methods declared in the interface. For example:

```
public class Printer implements Printable {
    public void print() {
      // Implementation for print method
```

```
    }
        public void getInfo() {
        // Implementation for getInfo method
    }
}
```

In this example, the `Printer` class implements the `Printable` interface and provides the required implementations for the `print()` and `getInfo()` methods.

It's important to note that interfaces cannot be instantiated directly. They are used as contracts that classes can adhere to by implementing them.

Nested interfaces in java

In Java, it is possible to define interfaces within other interfaces, resulting in nested interfaces. A nested interface is a way to logically group related interfaces within a containing interface. Here's an example of defining a nested interface in Java:

```
public interface OuterInterface {
    // OuterInterface methods
    void outerMethod();
    // Nested Interface declaration
    interface NestedInterface {
        // NestedInterface methods
        void nestedMethod();
    }
}
```

Applying Interfaces In Java

In Java, an interface is a reference type that defines a contract for classes to follow. It specifies a set of methods that a class implementing the interface must implement. Interfaces provide a way to achieve abstraction and define the behavior of objects without specifying their implementation details.

To apply interfaces in Java, you need to follow these steps:

1. Declare an interface: To create an interface, use the `interface` keyword followed by the interface name. Declare the methods that the implementing classes should implement. Here's an example:

```
public interface Printable {
    void print();
```

```
}
```

2. Implement an interface: To implement an interface, a class must use the `implements` keyword followed by the interface name. The class must provide implementations for all the methods declared in the interface. Here's an example:

```
public class Printer implements Printable {
    @Override
    public void print() {
        // Implementation of the print method
        System.out.println("Printing...");
    }
}
```

3. Use the interface: Once you have implemented the interface in a class, you can create objects of that class and refer to them using the interface type. This allows you to treat objects of different classes that implement the same interface uniformly. Here's an example:

```
public class Main {
    public static void main(String[] args) {
        Printable printable = new Printer();
        printable.print();  // Calling the print method through the interface
    }
}
```

In the example above, the `Main` class creates an object of the `Printer` class and assigns it to the `Printable` interface type. It can then call the `print()` method on the `printable` object, even though the reference type is an interface.

Interfaces can also be extended by other interfaces using the `extends` keyword. This allows you to build a hierarchy of interfaces. A class can implement multiple interfaces by separating them with commas in the `implements` clause.

Remember that interfaces only define the contract and method signatures, not the method implementations. Each class implementing the interface must provide its own implementation of the methods.

Variables In Interfaces

In Java, interfaces can define variables, but they are implicitly considered as `public`, `static`, and `final`. These variables are often referred to as "constants" or "static final fields" within an interface.

Here's an example of how you can declare and use variables within an interface:

```
public interface MyInterface {
    int MAX_COUNT = 100;  // Constant variable declaration
```

```
    void myMethod();  // Method declaration
}
```

In the example above, `MAX_COUNT` is a constant variable declared within the interface `MyInterface`. It is implicitly `public`, `static`, and `final`, meaning it can be accessed directly using the interface name (`MyInterface.MAX_COUNT`) and its value cannot be modified.

You can use this constant within the interface itself or by referring to it through the implementing classes. For example:

```
public class MyClass implements MyInterface {
    public void myMethod() {
        System.out.println("Max count: " + MAX_COUNT);
    }
}
```

In this case, the `MyClass` implements the `MyInterface` and overrides the `myMethod()` method. It can access the constant variable `MAX_COUNT` defined in the interface and use its value.

It's important to note that interfaces cannot have instance variables (non-static variables) because they define a contract rather than a concrete implementation.

Extending Interfaces.

In Java, interfaces can be extended by other interfaces using the `extends` keyword. This allows you to create a hierarchy of interfaces and inherit method declarations from one interface to another. The syntax for extending an interface is similar to extending a class.

Here's an example of how to extend an interface:

```
public interface Shape {
    void draw();
}


public interface Circle extends Shape {
    double calculateArea();
}
```

In the example above, the `Shape` interface defines a method `draw()`. The `Circle` interface extends the `Shape` interface using the `extends` keyword. By extending `Shape`, the `Circle` interface inherits the `draw()` method declaration from `Shape`. Additionally, the `Circle` interface declares its own method `calculateArea()`.

Classes implementing the `Circle` interface would be required to implement both the `draw()` method inherited from `Shape` and the `calculateArea()` method declared in `Circle`.

Here's an example of a class implementing the `Circle` interface:

```
public class CircleImpl implements Circle {
    public void draw() {
        System.out.println("Drawing a circle");
    }

    public double calculateArea() {
        // Calculate and return the area of the circle
    }
}
```

In this example, the `CircleImpl` class implements the `Circle` interface and provides the required implementations for both the `draw()` and `calculateArea()` methods.

By extending interfaces, you can create a hierarchical structure and define more specialized interfaces that build upon the functionality of the base interfaces. This allows for code reusability and promotes a modular design approach.

Stream based I/O (java.io)

The Stream classes: In Java, stream classes are part of the Stream API, which was introduced in Java 8 to provide a functional programming style for working with sequences of elements. Streams allow you to perform various operations on data, such as filtering, mapping, and reducing, in a concise and declarative manner.

The key classes and interfaces related to streams in Java are:

`java.util.stream.Stream`: This interface represents a sequence of elements that can be processed in parallel or sequentially. It provides methods to perform various operations on the stream, such as filtering, mapping, and reducing.

`java.util.stream.Collectors`: This class provides a set of predefined collectors for aggregating elements from a stream into various data structures, such as lists, sets, and maps.

`java.util.stream.IntStream`, `java.util.stream.LongStream`, `java.util.stream.DoubleStream`: These are specialized stream interfaces for working with primitive data types (`int`, `long`, and `double`) without the need for autoboxing.

`java.util.Optional`: This class represents an optional value, which may or may not be present. It is commonly used in stream operations to handle cases where a value may be absent.

`java.util.function.Predicate`: This functional interface represents a predicate (a boolean-valued function) that can be used for filtering elements in a stream.

`java.util.function.Function`: This functional interface represents a function that accepts one argument and produces a result. It is commonly used for mapping elements in a stream.

`java.util.function.Consumer`: This functional interface represents an operation that accepts a single input argument and returns no result. It is often used for consuming elements in a stream.

These are just a few of the key classes and interfaces related to Java streams. The Stream API provides many more classes and methods for performing complex operations on data in a streamlined and efficient manner.

Byte streams and Character streams:

In Java, byte streams and character streams are two types of I/O streams used for reading and writing data. They provide a way to handle different types of data, such as binary data (bytes) and text data (characters), respectively.

Byte Streams:

`java.io.InputStream` is the abstract base class for all byte input streams. It provides methods for reading bytes from a source.

`java.io.OutputStream` is the abstract base class for all byte output streams. It provides methods for writing bytes to a destination.

Example classes: `FileInputStream`, `FileOutputStream`, `ByteArrayInputStream`, `ByteArrayOutputStream`.

Character Streams:

`java.io.Reader` is the abstract base class for all character input streams. It provides methods for reading characters from a source.

`java.io.Writer` is the abstract base class for all character output streams. It provides methods for writing characters to a destination.

Example classes: `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`.

The key differences between byte streams and character streams are:

Byte streams operate on binary data and read/write data in the form of bytes. Character streams operate on text data and read/write data in the form of characters.

Byte streams are suitable for handling raw binary data, such as images or sound files. Character streams are designed for handling textual data, where character encoding and decoding are performed automatically.

Character streams are typically built on top of byte streams and provide additional functionality, such as buffering and character encoding/decoding.

Character streams are preferred when dealing with text data because they handle character encoding automatically, ensuring proper conversion between bytes and characters based on the specified character encoding.

When working with text files, it is generally recommended to use character streams, as they provide a higher level of abstraction and handle character encoding transparently. However, when dealing with binary files or raw byte data, byte streams are more appropriate.

Reading console Input and Writing Console Output

1. Reading Console Input: To read input from the console, you can use the `java.util.Scanner` class. Here's an example of how to read a line of text from the console:

```
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a line of text: ");
        String input = scanner.nextLine();

        System.out.println("You entered: " + input);

        scanner.close();
    }
}
```

In the above example, `System.in` represents the standard input stream, which is connected to the console. The `Scanner` class is used to read input from this stream. The `nextLine()` method reads a line of text from the console.

Writing Console Output: To write output to the console, you can use the `java.io.PrintStream` class, which is represented by the `System.out` object. Here's an example of how to write output to the console:

```
public class ConsoleOutputExample {
    public static void main(String[] args) {
        System.out.println("This is a line of output.");
        System.out.print("This is ");
        System.out.print("another line ");
        System.out.println("of output.");
    }
}
```

In the above example, **`System.out`** represents the standard output stream, which is connected to the console. The **`println()`** method is used to write a line of output, and the **`print()`** method is used to write output without a line break.

File class, Reading and writing Files, Random access file operations

In Java, the **`java.io.File`** class is used to represent file and directory paths. It provides methods for performing operations related to files and directories, such as creating, deleting, renaming, and checking for existence.

Here's an example of using the **`File`** class to perform file-related operations:

```
import java.io.File;

import java.io.IOException;


public class FileExample {

   public static void main(String[] args) {

      File file = new File("path/to/file.txt");


      // Check if the file exists

      if (file.exists()) {

         System.out.println("File exists");

      } else {

         System.out.println("File does not exist");

      }


      // Get the file name

      String fileName = file.getName();

      System.out.println("File name: " + fileName);


      // Get the absolute file path

      String absolutePath = file.getAbsolutePath();

      System.out.println("Absolute path: " + absolutePath);
```

```
        // Create a new file
        try {
            boolean created = file.createNewFile();
            if (created) {
                System.out.println("File created successfully");
            } else {
                System.out.println("File already exists");
            }
        } catch (IOException e) {
            System.out.println("Error creating file: " + e.getMessage());
        }

        // Delete the file
        boolean deleted = file.delete();
        if (deleted) {
            System.out.println("File deleted successfully");
        } else {
            System.out.println("Failed to delete file");
        }
    }
}
```

Reading and Writing Files: To read and write the contents of a file, you can use various classes in the `java.io` package, such as `FileInputStream`, `FileOutputStream`, `BufferedReader`, `BufferedWriter`, etc.

Here's an example of reading the contents of a file using `BufferedReader`:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFileExample {
    public static void main(String[] args) {
```

```
        try (BufferedReader reader = new BufferedReader(new FileReader("path/to/file.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

And here's an example of writing to a file using `BufferedWriter`:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("path/to/file.txt"))) {
            writer.write("Hello, world!");
            writer.newLine();
            writer.write("This is a new line.");
        } catch (IOException e) {
            System.out.println("Error writing to file: " + e.getMessage());
        }
    }
}
```

Random Access File Operations:

To perform random access file operations, where you can read and write data at any position within a file, you can use the `java.io.RandomAccessFile` class.

Here's an example of performing random access operations on a file:

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {
   public static void main(String[] args) {
      try (RandomAccessFile file = new RandomAccessFile("path/to/file.txt", "rw")) {
         // Write data at a specific position
         file.seek(10);
         file.writeBytes("Hello");

         // Read data from a specific position
         file.seek(5);
         byte[] buffer = new byte[10];
         int bytesRead = file.read(buffer);
         String data = new String(buffer, 0, bytesRead);
         System.out.println("Data read: " + data);
      } catch (IOException e) {
         System.out.println("Error accessing file: " + e.getMessage());
      }
   }
}
```

In the above example, the `RandomAccessFile` is opened in "rw" mode, which allows both reading and writing. The `seek()` method is used to move the file pointer to a specific position, and `writeBytes()` is used to write data at that position. Similarly, `seek()` is used to move the file pointer for reading, and `read()` is used to read data into a buffer.

The Console class, Serialization, Enumerations, auto boxing, generics

The Console Class:

The `java.io.Console` class provides methods for reading input and writing output on the console. It is typically used in command-line programs. Here's an example of reading input from the console using the `Console` class:

```
import java.io.Console;

public class ConsoleExample {
   public static void main(String[] args) {
      Console console = System.console();

      if (console != null) {
         String input = console.readLine("Enter your name: ");
         console.writer().println("You entered: " + input);
      } else {
         System.out.println("Console is not available.");
      }
   }
}
```

Serialization:

Serialization is the process of converting an object into a stream of bytes so that it can be stored in a file, sent over a network, or saved in persistent storage. In Java, serialization is achieved using the `java.io.Serializable` interface. Here's an example of serializing and deserializing an object:

```
import java.io.*;

class Person implements Serializable {
   private String name;
   private int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public String getName() {
```

```
        return name;
    }

    public int getAge() {
        return age;
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        Person person = new Person("John Doe", 30);

        // Serialize the object
        try (FileOutputStream fileOut = new FileOutputStream("person.ser");
             ObjectOutputStream objectOut = new ObjectOutputStream(fileOut)) {
            objectOut.writeObject(person);
            System.out.println("Object serialized successfully.");
        } catch (IOException e) {
            System.out.println("Error serializing object: " + e.getMessage());
        }

        // Deserialize the object
        try (FileInputStream fileIn = new FileInputStream("person.ser");
             ObjectInputStream objectIn = new ObjectInputStream(fileIn)) {
            Person deserializedPerson = (Person) objectIn.readObject();
            System.out.println("Deserialized object: " + deserializedPerson.getName() + ", "
+ deserializedPerson.getAge());
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Error deserializing object: " + e.getMessage());
        }
```

```
    }
}
```

Enumerations:

Enumerations, often referred to as enums, are special types in Java that represent a fixed set of constants. Enumerations provide type safety and can be used in switch statements. Here's an example of defining and using an enum:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumExample {
    public static void main(String[] args) {
        Day today = Day.TUESDAY;

        switch (today) {
            case MONDAY:
                System.out.println("It's Monday.");
                break;
            case TUESDAY:
                System.out.println("It's Tuesday.");
                break;
            case WEDNESDAY:
                System.out.println("It's Wednesday.");
                break;
            default:
                System.out.println("It's some other day.");
        }
    }
}
```

Auto Boxing:

Auto boxing is the automatic conversion of primitive data types to their corresponding wrapper classes, and vice versa. It allows you to use primitive types and wrapper classes interchangeably. For example:

```
int num = 42; // primitive int
Integer numObj = num; // auto boxing: int to Integer
System.out.println(numObj); // Integer to String (auto unboxing) and print
```

Generics:

Generics allow you to create classes and methods that can work with different types without sacrificing type safety. They provide compile-time type checking and enable you to write reusable code. Here's an example of a generic class and method:

```
class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}

public class GenericsExample {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setItem("Hello");
        String item = stringBox.getItem();
        System.out.println(item);
```

```
        Box<Integer> integerBox = new Box<>();
        integerBox.setItem(42);
        int num = integerBox.getItem();
        System.out.println(num);
    }
}
```

In the above example, the `Box` class is a generic class that can hold an item of any type. The type parameter `T` is specified when creating an instance of the class (`Box<String>` and `Box<Integer>`). The `setItem()` and `getItem()` methods can work with the specified type, ensuring type safety at compile time.

Design an interface called Shape with methods draw() and getArea(). Further design two classes called Circle and Rectangle that implements Shape to compute area of respective shapes. Use appropriate getter and setter methods. Write a java program for the same.

```
// Shape interface
interface Shape {
    void draw();
    double getArea();
}

// Circle class implementing Shape
class Circle implements Shape {
    private double radius;

    // Constructor
    public Circle(double radius) {
        this.radius = radius;
    }
```

```
    // Getter and Setter for radius
    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    // Implementing draw() method
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }

    // Implementing getArea() method
    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}

// Rectangle class implementing Shape
class Rectangle implements Shape {
    private double length;
    private double width;

    // Constructor
    public Rectangle(double length, double width) {
```

```
        this.length = length;
        this.width = width;
    }

    // Getters and Setters for length and width
    public double getLength() {
        return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    // Implementing draw() method
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }

    // Implementing getArea() method
    @Override
```

```
    public double getArea() {
        return length * width;
    }
}

// Main class
public class ShapeExample {
    public static void main(String[] args) {
        // Creating Circle object
        Circle circle = new Circle(5.0);
        circle.draw();
        System.out.println("Circle Area: " + circle.getArea());

        // Creating Rectangle object
        Rectangle rectangle = new Rectangle(4.0, 6.0);
        rectangle.draw();
        System.out.println("Rectangle Area: " + rectangle.getArea());
    }
}
```

In this program, the `Shape` interface declares two methods: `draw()` and `getArea()`. The `Circle` and `Rectangle` classes implement this interface and provide their own implementations of these methods.

The `Circle` class has a private `radius` field and implements the formulas for drawing a circle and calculating its area based on the radius.

The `Rectangle` class has private `length` and `width` fields and implements the formulas for drawing a rectangle and calculating its area based on the length and width.

In the `main()` method of the `ShapeExample` class, we create instances of the `Circle` and `Rectangle` classes and invoke the `draw()` and `getArea()` methods on them to demonstrate their functionality.

Reading console Input and Writing Console Output in java

In Java, you can read input from the console and write output to the console using the standard input and output streams provided by the `System` class. Here's how you can do it:

1. Reading Console Input: To read input from the console, you can use the `Scanner` class or the `BufferedReader` class. Here's an example using `Scanner`:

```
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);

        scanner.close();
    }
}
```

In this example, we create a `Scanner` object with `System.in` as the input source. We use the `nextLine()` method to read a line of input for the name and `nextInt()` to read an integer input for the age.

2. Writing Console Output: To write output to the console, you can use the `System.out` stream. You can use the `print()` or `println()` methods to display the output. Here's an example:

```
public class ConsoleOutputExample {
    public static void main(String[] args) {
        String name = "John Doe";
        int age = 25;

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

In this example, we simply use the `println()` method to display the name and age on separate lines. The output will be printed to the console.

Remember to import the necessary classes (`java.util.Scanner` and `java.io.BufferedReader`) before using them in your code.