Inheritance

- In Java, Inheritance is an important pillar of OOP(Object-Oriented Programming).
- It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.
- In Java, Inheritance means creating new classes based on existing ones.
- A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Important Terminologies Used in Java Inheritance

Class: Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is just a template or blueprint or prototype from which objects are created.

Super Class/Parent Class: The class whose features are inherited is known as a superclass(or a base class or a parent class).

Sub Class/Child Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to Use Inheritance in Java?

The **extends keyword** is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, "extends" refers to increased functionality.

Syntax :

```
class DerivedClass extends BaseClass
{
  //methods and fields
}
```



```
// Parent class
class Animal {
    // Method in the parent class
    public void sound() {
        System.out.println("Animals make different sounds.");
    }
}

// Child class that extends Animal (Single Inheritance)
class Dog extends Animal {
    // Method in the child class
    public void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of the Dog class
        Dog dog = new Dog();

        // Calling methods from both parent and child class
        dog.sound();  // Method inherited from Animal class
        dog.bark();   // Method from Dog class
```

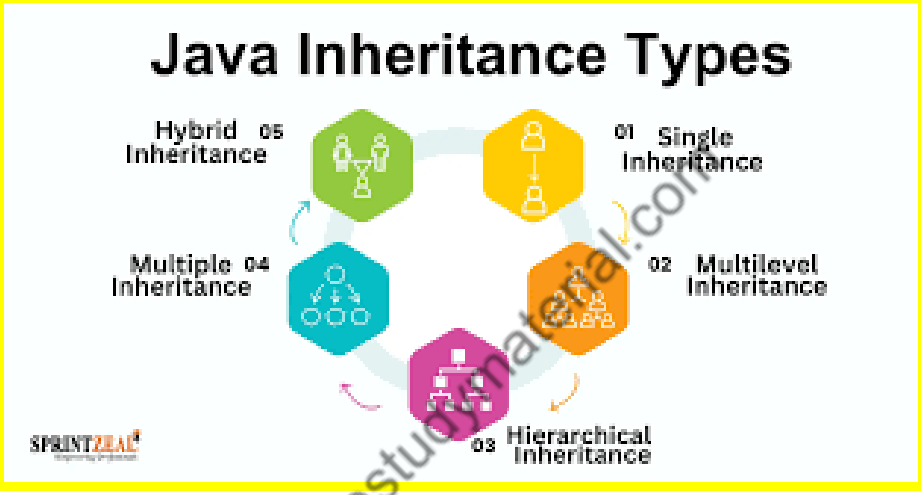
```
    }
}
```

Output:
Animals make different sounds.
The dog barks.

Types of Inheritance in Java

In object-oriented programming (OOP), inheritance allows a class (called a child or subclass) to inherit properties and behaviors (methods) from another class (called a parent or superclass).

There are several types of inheritance, depending on how classes are related. Here's a breakdown:



1. **Single Inheritance**

Single inheritance is when **one child class inherits from one parent class**. It's the most basic and most common type of inheritance.

Need of single inheritance:

To avoid code duplication and **reuse existing features** in a new class.

When to use?

Use this when you have a clear one-to-one relationship like:

- A Car **is-a** Vehicle
- A Student **is-a** Person

It's suitable when your **design is simple and focused**.

Code:

```
class Vehicle {
    void displayDetails() {
        System.out.println("I am a Vehicle.");
    }
}

class Car extends Vehicle {
    void carType() {
        System.out.println("I am a Car.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car obj = new Car();
        obj.displayDetails();
        obj.carType();
    }
}
```


Output:

I am a Vehicle.

I am a Car

2. Multilevel Inheritance

In this inheritance, a **class inherits from a child class**, which itself inherited from another class. It's like a chain or ladder.

Why we need it?

To create **progressive or layered classes**, where each level adds more functionality on top of the previous one.

This helps in building a **step-by-step specialization**.

When to use?

Use this when you want to build **specialized versions** of a class step-by-step.

Example:

- Vehicle → Car → ElectricCar

Every new level adds a new feature.

Code:


```
class Vehicle {
    void displayDetails() {
        System.out.println("I am a Vehicle.");
    }
}


class Car extends Vehicle {
    void carType() {
```

```
        System.out.println("I am a Car.");
    }
}

class ElectricCar extends Car {
    void batteryStatus() {
        System.out.println("Battery is fully charged.");
    }
}

public class Main {
    public static void main(String[] args) {
        ElectricCar obj = new ElectricCar();
        obj.displayDetails();
        obj.carType();
        obj.batteryStatus();
    }
}
```


Output:

I am a Vehicle.

I am a Car.

Battery is fully charged.

3. Hierarchical Inheritance

Multiple child classes inherit from the same parent class. So the parent's properties are shared across multiple child classes.

Why we need it?

To create **varieties of objects** that share common features from a single parent.

This helps in organizing code logically and avoiding repetition.

When to use?

Use it when you have multiple related classes that have some **common properties**, like:

- Car, Bike, Truck are all types of Vehicle.

Each has some unique behavior, but they also share some common features like speed, color, etc.

Code:

```
class Vehicle {
    void displayDetails() {
        System.out.println("I am a Vehicle.");
    }
}


class Car extends Vehicle {
    void carType() {
        System.out.println("I am a Car.");
    }
}
```

```
class Bike extends Vehicle {
    void bikeType() {
        System.out.println("I am a Bike.");
    }
}

class Truck extends Vehicle {
    void truckType() {
        System.out.println("I am a Truck.");
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        Bike bike = new Bike();
        Truck truck = new Truck();

        car.displayDetails();
        car.carType();
        bike.displayDetails();
        bike.bikeType();
        truck.displayDetails();
```

```
        truck.truckType();
    }
}
```

Output:

I am a Vehicle.

I am a Car.

I am a Vehicle.

I am a Bike.

I am a Vehicle.

I am a Truck.

4. Multiple Inheritance (via interfaces in Java)

When a class inherits **features from more than one parent**, it's called **multiple inheritance**. Java supports it **only via interfaces**.

Why we need it?

To build a class that needs to inherit behaviors from **multiple sources**. For example:

- A Car needs both Engine and Wheels.

This lets us **modularize functionality** into separate interfaces and **combine** them.

When to use?

Use it when:

- You want a class to have **multiple capabilities**
- You're working with **independent features**

Example:

- Car is a Vehicle, but also has an Engine and Wheels.

This avoids the **diamond problem** (conflict from multiple parents).

Diamond Problem:

The **Diamond Problem** occurs in **multiple inheritance** when **a class inherits from two classes that both inherit from the same base class**. This creates a **diamond-shaped hierarchy**, leading to **ambiguity**: the compiler doesn't know **which version** of the base class method or property to use.

Code:

```
interface Engine {
    void startEngine();
}

interface Wheels {
    void rotateWheels();
}

class Car implements Engine, Wheels {
    public void startEngine() {
        System.out.println("Engine started.");
    }

    public void rotateWheels() {
        System.out.println("Wheels are rotating.");
```

```
    }

    void carDetails() {
        System.out.println("I am a Car.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.startEngine();
        car.rotateWheels();
        car.carDetails();
    }
}
```


Output:

Engine started.

Wheels are rotating.

I am a Car.

5. Hybrid Inheritance

Hybrid inheritance is a **combination** of multiple types of inheritance, such as **multilevel + hierarchical** or **hierarchical + multiple**. Java allows this **only via interfaces**.

Why we need it?

Sometimes real-world models are complex. You may need a class to:

- Inherit from a class (like Vehicle)
- And also implement multiple interfaces (like Engine, Wheels)

Hybrid inheritance helps create **flexible and powerful systems** by combining different inheritance patterns.

When to use?

Use hybrid inheritance when:

- You need both **shared properties** (from classes)
- And **independent features** (from interfaces)
- You are building **large-scale systems** that need layered and mix-match design

Example:

- Car is a Vehicle
- It also has an Engine and Wheels behavior

Code :

```
class Vehicle {
   void displayDetails() {
      System.out.println("I am a Vehicle.");
   }
}

interface Engine {
   void startEngine();
}

interface Wheels {
   void rotateWheels();
```

```
}

class Car extends Vehicle implements Engine, Wheels {
    public void startEngine() {
        System.out.println("Car engine started.");
    }

    public void rotateWheels() {
        System.out.println("Car wheels rotating.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.displayDetails();
        car.startEngine();
        car.rotateWheels();
    }
}
```

Output:
I am a Vehicle.
Car engine started.
Car wheels rotating.

Summary

Type	Description	Example
Single	One child inherits one parent	Car inherits Vehicle
Multilevel	Chain of inheritance	ElectricCar → Car → Vehicle
Hierarchical	Multiple classes inherit one parent	Car, Bike, Truck ← Vehicle
Multiple (via Interface)	Class inherits from multiple interfaces	Car implements Engine, Wheels

Hybrid (via Interface)	Combination of hierarchical & multiple	Car extends Vehicle, implements Engine, Wheels

Super Keyword in Java

The super keyword in Java is used to refer to the immediate parent class of a subclass. It serves three main purposes:

- Uses of super keyword:

1. To access the parent class data member (variable)
2. To access the parent class method
3. To call the parent class constructor

---

1. To Access Parent Class Data Members (Variables)

If a subclass has a field with the same name as the parent class, the super keyword can be used to refer to the parent class field.

Example:


```
class Parent {
    int x = 100;
}

class Child extends Parent {
    int x = 200;

    void display() {
        System.out.println("Child x: " + x);        // Accessing child class variable
        System.out.println("Parent x: " + super.x); // Accessing parent class variable
```

```
    }

    public static void main(String[] args) {
        Child obj = new Child();
        obj.display();
    }
}
```

OUTPUT:-

Child x: 200
Parent x: 100

2. To Invoke Parent Class Methods

When a subclass overrides a method from its superclass, you can use super.methodName() to call the **parent class version** of the method.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    void sound() {
```

```
        super.sound(); // Calling parent class method
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.sound();
    }
}
```

Output:

```
Animal makes a sound
Dog barks
```


3. To Invoke Parent Class Constructor

You can use super() to call the **constructor of the parent class**. This must be the **first statement** in the subclass constructor.

Example:

```
class Vehicle {
    Vehicle() {
        System.out.println("Vehicle constructor");
    }
```

```
}

class Car extends Vehicle {
    Car() {
        super(); // Calls Vehicle constructor
        System.out.println("Car constructor");
    }
 public static void main(String[] args) {
        Car obj = new Car();
    }
}
```

Output:

```
Vehicle constructor
Car constructor
```

Important Points

- super() must be the **first line** in a subclass constructor if used.
- If super() is not explicitly written, Java implicitly calls the **no-arg constructor** of the superclass.
- Cannot be used in static methods.

Constructor Call Sequence

What is a Constructor?

A **constructor** is a special method in a class that is automatically called when an object is created. It is used to **initialize the object**.

What is Constructor Call Sequence?

When you create an object of a class that **inherits from another class**, Java makes sure that:

- **The parent class constructor is called before the child class constructor.**

This rule ensures that the **base class part** of the object is initialized **before** the extended (child) part.

Real-Life Analogy: Family Hierarchy

Imagine three generations:

- Grandpa builds the house (base setup)
- Dad adds furniture (additional setup)
- You add your gaming PC (custom setup)

If *you* try to set up your PC before the house is ready, it doesn't make sense! Similarly, Java ensures:

Parent constructor → Child constructor → Grandchild constructor

---

Why Constructor Call Sequence is Important?

1. It ensures that **common features** of all child classes (like number of wheels in vehicles) are initialized first.

2. It builds a **strong base** before adding specialized behaviors.

3. Prevents runtime errors due to **uninitialized fields or logic**.

Let's Break Down the Types with Examples:

✅ 1. Single-Level Inheritance with Default Constructor


```
class Vehicle {
    Vehicle() {
        System.out.println("Vehicle Constructor Called");
    }
}

class Car extends Vehicle {
    Car() {
        System.out.println("Car Constructor Called");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
    }
```

```
}
```

▸ **Output:**

```
Vehicle Constructor Called
Car Constructor Called
```

Explanation:

- You created a Car object.
- Java **first** called the parent (Vehicle) constructor using super().
- Then it ran the Car constructor.

Even if we don't write super(), **Java automatically adds it** as the **first line**.

---

2. Constructor with Parameters

```
class Vehicle {
   Vehicle(String type) {
      System.out.println("Vehicle Type: " + type);
   }
}

class Car extends Vehicle {
   Car(String type) {
      super(type); // Must be FIRST line!
      System.out.println("Car Constructor Called");
   }
}

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

```
        Car c = new Car("SUV");
    }
}
```

▸ **Output:**

Vehicle Type: SUV
Car Constructor Called

Explanation:

- Since the parent Vehicle has **no default constructor**, you MUST call the parameterized constructor using super(type).
- If you **don't**, Java throws an error.

What if we forget super() here?

You'll get:

Constructor Vehicle in class Vehicle cannot be applied to given types

Because Java doesn't know **which constructor** to call without default.

✅ 3. Multilevel Inheritance Constructor Sequence

```
class Vehicle {
```

```
    Vehicle() {
        System.out.println("Vehicle Constructor");
    }
}

class Car extends Vehicle {
    Car() {
        System.out.println("Car Constructor");
    }
}

class ElectricCar extends Car {
    ElectricCar() {
        System.out.println("ElectricCar Constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        ElectricCar e = new ElectricCar();
    }
}
```


▸ **Output:**

```
Vehicle Constructor
Car Constructor
ElectricCar Constructor
```

Explanation:

- ElectricCar() calls Car() → Car() calls Vehicle()

- Constructors are called **from parent to child**
- This is called **Constructor Chaining**

More on super() and this()

Keyword	Purpose	Where it goes	Used for
super()	Calls **parent class constructor**	First line of child constructor	Inheritance
this()	Calls **another constructor in the same class**	First line of current class constructor	Constructor overloading

Method Overriding in Java

Method Overriding means **redefining a method in the child class** that is already defined in the parent class.

It allows a **child class** to provide its **own version** of a method that is already present in its **parent class**.

Why Do We Need Method Overriding?

- To **change or extend the behavior** of a method for a specific subclass.
- To achieve **runtime polymorphism** (method call decided at runtime).
- To make the program more **dynamic and flexible**.

Real-Life Example:

Suppose you have a class Animal with a method makeSound(). But all animals make different sounds.

```
class Animal {
   void makeSound() {
      System.out.println("Animal makes a sound");
   }
}
```

Now, Dog and Cat can **override** this method to give their own behavior.

Method Overriding – Rules

Rule	Description
Method name	Must be the same
Parameters	Must be the same (same number, type, order)
Return type	Must be same or covariant
Access level	Can be more visible (e.g., protected → public)
Final method	❌ Cannot override a final method
Static method	❌ Static methods are not overridden, they are hidden
Constructor	❌ Constructors cannot be overridden

Basic Example: Method Overriding

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Dog d = new Dog();
        d.makeSound();  // Dog's version is called
    }
}
```

▸ **Output:**

Dog barks

Even though makeSound() is defined in Animal, the Dog class overrides it with its own version.

Polymorphism Using Overriding

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Cat();  // Parent reference, child object
        a.makeSound();         // Calls Cat’s version
    }
}
```


▸ **Output:**

Cat meows

This is **runtime polymorphism** — Java decides which method to call **at runtime** based on the actual object type.

What If We Don't Override?

```
class Animal {
    void makeSound() {
        System.out.println("Generic animal sound");
    }
}

class Cow extends Animal {
    // No overriding
}

public class Main {
    public static void main(String[] args) {
        Cow c = new Cow();
        c.makeSound();  // Will use Animal's version
    }
}
```


▸ **Output:**

Generic animal sound

Example with super Keyword

You can call the parent class version of the method using super.

```
class Vehicle {
   void start() {
      System.out.println("Vehicle is starting...");
   }
}

class Car extends Vehicle {
   void start() {
      super.start();  // Calls Vehicle's method
      System.out.println("Car is ready to go!");
   }
}

public class Main {
   public static void main(String[] args) {
      Car c = new Car();
      c.start();
   }
}
```


▸ **Output**

```
Vehicle is starting...
Car is ready to go!
```

What Can’t Be Overridden?

Type	Can it be overridden?

final method	❌ No
static method	❌ No (it's method hiding, not overriding)
private method	❌ No (not visible in child)
Constructor	❌ No (not inherited)

Overriding vs Overloading

Feature	**Overriding**	**Overloading**
Purpose	Modify parent method	Define multiple methods
Parameters	Must be same	Must be different
Inheritance	Required	Not required
Runtime/Compile	Runtime	Compile time

Dynamic Method Dispatch (Runtime Polymorphism)

Dynamic Method Dispatch is the process through which a call to an **overridden method** is resolved **at runtime**, not compile time. It's how **Java decides which method version to call** when a superclass reference refers to a subclass object.

Java Example: Dynamic Method Dispatch

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a;        // Parent reference

        a = new Dog();    // Dog object
```

```
        a.makeSound();    // Dog's version is called
        a = new Cat();    // Cat object
        a.makeSound();    // Cat's version is called
    }
}
```

▸ **Output:**

```
Dog barks
Cat meows
```

Even though the reference type is Animal, the method that gets executed is based on the **actual object** type (Dog or Cat) at **runtime**.

Method Overriding vs. Dynamic Method Dispatch

Feature	Method Overriding	Dynamic Method Dispatch
What is it?	Redefining a method in the subclass	Process of selecting the overridden method
When?	Defined at coding time	Happens during program execution (runtime)
Involves?	Method signature and body	Object reference and actual object
Polymorphism?	Enables it	Implements it
Focus	On defining methods	On calling the correct method at runtime
Reference type needed?	No (works in child class too)	Yes, reference should be of parent class

Abstract Class in Java

An **abstract class** in Java is a class that **cannot be instantiated** on its own. It is meant to be a **base class** for other classes. It may contain:

- **Abstract methods** (without a body)
- **Concrete methods** (with a body)
- Constructors
- Static methods
- Final methods

It is declared using the abstract keyword:

```
abstract class Shape {
    abstract void draw(); // abstract method

    void display() {
        System.out.println("Displaying shape");
    }
}
```

Important Points:

1. **Cannot create objects** of abstract class directly.

2. Can **have constructors**, which are called when subclass objects are created.
3. Abstract class can **have both abstract and non-abstract methods**.
4. A class must be declared abstract if it contains one or more abstract methods.
5. Subclasses must **implement all abstract methods** unless they are also declared abstract.

When to Use Abstract Classes?

- When you want to **provide a common base** with **shared code**.
- When you want to **enforce certain methods** in derived classes.
- When classes are **closely related** and share a common structure or behavior.

For Example, imagine **"Vehicle"** as an abstract class. You never say “I have a vehicle.” You say “I have a bike” or “I have a car.” But “vehicle” gives a **common base** – all vehicles can move, but the way they move may differ.

Why Do We Need Abstract Classes?

1. To Provide a Common Base for Related Classes

An abstract class lets you define a common **template** (structure + partial implementation) for all its subclasses.

2. To Enforce Method Implementation in Subclasses

By declaring methods as `abstract`, you force all subclasses to **override and implement them**, ensuring a consistent structure.

3. To Achieve Partial Abstraction

Unlike interfaces (which are 100% abstract in older Java versions), abstract classes allow **partial abstraction**:

- Some methods can have implementations (concrete methods).
- Others can be left abstract for subclasses to define.

This is helpful when some functionality should be reused as-is, and other functionality should be customizable.

4. To Avoid Object Creation of Incomplete Classes

If a class is **not complete on its own**, it should not be instantiated. Making it abstract ensures that only its concrete subclasses can be used to create objects.

5. To Use Constructors in Base Classes

Abstract classes can have constructors, which can be used to initialize common fields. Interfaces cannot have constructors.

6. To Create a Framework or Template

Abstract classes are often used in large frameworks and libraries to define a **template method** where the basic structure is fixed, but certain steps can be overridden by subclasses.

Abstract Class:

```
abstract class Shape {
    String color;

    Shape(String color) {
        this.color = color;
    }

    // Abstract method to be implemented by all shapes
    abstract void draw();

    // Common method
    void displayColor() {
        System.out.println("Color: " + color);
    }
}
```

Concrete Class:

```
class Circle extends Shape {
    int radius;

    Circle(String color, int radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    void draw() {
        System.out.println("Drawing a Circle with radius " + radius);
    }
}

class Rectangle extends Shape {
    int length, width;

    Rectangle(String color, int length, int width) {
        super(color);
        this.length = length;
        this.width = width;
    }

    @Override
    void draw() {
        System.out.println("Drawing a Rectangle of size " + length + "x" + width);
    }
}
```

```
public class DrawingApp {
    public static void main(String[] args) {
        Shape circle = new Circle("Red", 5);
        Shape rectangle = new Rectangle("Blue", 10, 4);

        circle.draw();
        circle.displayColor();

        rectangle.draw();
        rectangle.displayColor();
    }
}
```

Output:

```
Drawing a Circle with radius 5
Color: Red
Drawing a Rectangle of size 10x4
Color: Blue
```

PPL- Unit 4
Managing I/O: Streams, Byte Streams and Character Streams

Streams: In Java, a stream is a sequence of data used to perform input and output (I/O) operations. It acts like a pipeline through which data flows.

Types of Streams in Java

1. Byte Streams
2. Character Streams

1. Byte Streams

Used to perform **input and output of 8-bit bytes**, suitable for handling **binary data** such as images, audio, and video files.

Base Classes:

- `InputStream` (abstract class for reading byte data)
- `OutputStream` (abstract class for writing byte data)

Class Name	Description
FileInputStream	Reads raw bytes from a file
FileOutputStream	Writes raw bytes to a file
BufferedInputStream	Adds buffering to InputStream for faster reading
BufferedOutputStream	Adds buffering to OutputStream for faster writing

DataInputStream	Reads Java primitive data types in a machine-independent way
DataOutputStream	Writes Java primitive data types

Example of ByteStream Class:

```
FileInputStream fis = new FileInputStream("image.jpg");
FileOutputStream fos = new FileOutputStream("copy.jpg");

int byteData;
while ((byteData = fis.read()) != -1) {
    fos.write(byteData);
}
fis.close();
fos.close();
```


2. Character Streams

Used to perform **input and output of 16-bit characters**, suitable for handling **text data** (strings, characters, etc.).

Base Classes:

- Reader (abstract class for reading characters)
- Writer (abstract class for writing characters)

Common Character Stream Classes:

Class Name	Description
FileReader	Reads characters from a file
FileWriter	Writes characters to a file

BufferedReader	Buffers characters for efficient reading
BufferedWriter	Buffers characters for efficient writing
PrintWriter	Writes formatted text (console or file)
CharArrayReader	Reads characters from a character array
CharArrayWriter	Writes characters to a character array

Example of CharacterStream Class:

```
FileReader fr = new FileReader("notes.txt");
BufferedReader br = new BufferedReader(fr);
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
br.close();
```