Course-level coding cheat sheet

This course-level reading provides a reference list of code that you'll encounter as you work with object-oriented coding in Java. Use this cheat sheet code as an ongoing resource to help you write and debug object-oriented Java programs.

Select the hamburger menu located at the top left of the window to quickly locate code and explanations based on the video name. You can also use the forward and back arrows to navigate between pages.

This course-level cheat sheet includes code and related explanations from the following videos.

- Working with Classes and Objects
- Working with Access and Non-access Modifiers
- Using Encapsulation
- Using Constructors
- Inheritance in Java
- Polymorphism in Java
- Designing interfaces and Abstract Classes in Java
- Inner classes in Java
- Java Collections Framework (JCF)
- Working with lists
- HashSet and TreeSet
- Implementing Queues in Java
- Using HashMap and TreeMap
- Using Java collections in the Real World
- Java File Handling / Working with File Input and Output Streams
- Using Java Byte Streams
- Managing Directories in Java
- Using Java Date and Time Classes
- Formatting Dates in Java
- Using Timezones in Java
- Parsing Dates from Strings in Java

Working with Classes and Objects

Creating a class

Description	Example
Create a `Car` class, which serves as a blueprint for creating Car objects.	`public class Car {`

Define attributes of the `Car` class. The variables `color`, `model`, and `year` store the car's color, model, and year, respectively.	`String color;`
	`String model;`
	`int year;`

Include the method `displayInfo()` to print car objects.	`void displayInfo() {`

Print the car details to the console using the `System.out.println()` function.	`System.out.println("Car Model: " + model);`
	`System.out.println("Car Color: " + color);`
	`System.out.println>System.out.println("Car Year: " + year);`

Close curly braces to end the `Car` class definition.	`}` `}`

Explanation: This example creates a class named `Car` and defines three attributes for the `Car` class: `model`, `color`, and `year`. The `displayInfo()` method prints the car details.

Creating an object

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an object of the `Car` class.	`Car myCar = new Car();`

Assign values to the object's attributes.	`myCar.color = "Red";`
	`myCar.model = "Toyota";`
	`myCar.year = 2020;`

Call the `displayInfo()` method to print the object details.	`myCar.displayInfo();`
Close curly braces to end the `main` method and class definition.	`}` `}`

Explanation: This example declares a reference variable named `myCar` of type `Car`. `new Car()` creates a new object of the `Car` class and assigns values to the object's attributes: `color`, `model` and `year`. The `displayInfo()` method prints the car details.

Working with access and non-access modifiers

Public access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning it can be accessed directly from outside the class.	`public class Car {`
A Java statement to declare a `String` variable named `model` to store the car's model name.	`public String model;`
Close curly braces to end the class definition.	`}`

Private access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning that it can be accessed directly from outside the class.	`public class Car {`
A Java statement to declare a private `String` variable named `color` to store the car's color. The private modifier ensures the `color` variable can be accessed and modified only within the `Car` class.	`private String color;`

Description	Example
Call the `displayColor()` method with the private access modifier. This ensures the method can be called only within the `Car` class and not from other classes.	`private void displayColor() {`
Print the car's color to the console using the `System.out.println()` function.	`System.out.println("Car Color: " + color);`
Close curly braces to end the class definition.	`}` `}`

Protected access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning that it can be accessed directly from outside the class.	`public class Car {`
A Java statement to declare a protected `int` variable named year to store the car's year. The protected modifier ensures the year variable is accessible within the same package (default package access) and by subclasses, even if they are in different packages.	`private String year;`
Call the `displayYear()` method with the protected access modifier. This ensures the method can be called within the same package and by subclasses, even if they are in different packages.	`private void displayYear() {`
Print the car's year to the console using the `System.out.println()` function.	`System.out.println("Car Year: " + year);`
Close curly braces to end the class definition.	`}` `}`

Default access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects.	`class Car {`
A Java statement to declare a `String` variable named `model` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String model;`
Call the `displayModel()` method without any access modifier.	`void displayModel() {`
Print the car's model to the console using the `System.out.println()` function.	`System.out.println("Car Model: " + model);`
Close curly braces to end the class definition.	`}` `}`

Static non-access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning that it can be accessed directly from outside the class.	`public class Car {`
A Java statement to declare a static `int` variable named `numberOfCars` to keep track of the total number of `Car` objects created. Since it's static, its value is shared among all instances of `Car`.	`static int numberOfCars = 0;`
A Java statement to declare a constructor. Every time a new `Car` object is created, this constructor runs.	`public Car() {`
A Java statement to increment the `numberOfCars` variable that keeps track of how many cars have been instantiated.	`numberOfCars++;`
Close curly braces to end the class definition.	`}`
Call the `displayCount()` method without creating an instance of the `Car` class. This method can only access static variables such as `numberOfCars`, not instance variables.	`private void displayCount() {`

Description	Example
Print the total number of cars to the console using the `System.out.println()` function.	`System.out.println("Total Cars: " + numberOfCars);`
Close curly braces to end the class definition.	`}` `}`

Final non-access modifier

Description	Example
A Java statement used to define a final class named `Vehicle`, which acts as a blueprint for creating `Car` objects. The final class cannot be extended (inherited) by any other class. This means no subclasses can be created from `Vehicle`.	`public final class Vehicle {`
A Java statement to declare a final `int` variable named `maxSpeed` with the value 120. The final variable is a constant, meaning that its value cannot be changed once it is assigned. Trying to modify `maxSpeed` later in the code will cause a compilation error.	`final int maxSpeed = 120;`
A Java statement to declare a final method named `displayMaxSpeed()`. The final method cannot be overridden by subclasses. This ensures the behavior of `displayMaxSpeed` remains the same in all instances.	`final void displayMaxSpeed() {`
Print the maximum car speed to the console using the `System.out.println()` function.	`System.out.println("Max Speed: " + maxSpeed);`
Close curly braces to end the class definition.	`}` `}`

Abstract non-access modifier

Description	Example
A Java statement used to define an abstract class named `Shape`. This is an abstract class, meaning that it cannot be instantiated (you cannot create `Shape` objects directly). It works as a blueprint from which other classes can inherit.	`public abstract class Shape {`
A Java statement used to define a final class named `Vehicle`, which acts as a blueprint for creating `Car` objects. The final class cannot be extended (inherited) by any other class. This means no subclasses can be created from `Vehicle`.	`abstract void draw();`
Close curly braces to end the class definition.	`}`
A Java statement to describe `Circle` that extends the `Shape` class and provides an implementation of the `draw()` method.	`public class Circle extends Shape {`
A Java annotation to tell the compiler the `draw()` method in `Circle` is an override of the abstract method in `Shape`.	`@Override`
A Java statement saying the `draw()` method is now fully implemented.	`void draw()`
Print the string `Drawing Circle` to the console using the `System.out.println()` function.	`System.out.println("Drawing Circle");`
Close curly braces to end the class definition.	`}` `}` `}`

Using encapsulation

Creating an encapsulated class

Description	Example
Create the `Person` class, which serves as a blueprint for creating Person objects.	`class Person {`
Create private attributes `name` and `age` to store the person's name and age. The `name` and `age` attributes cannot be accesse directly from outside the class.	`private String name;` `private int age;`
Use the Java constructor to initialize the `name` and `age` variables when a `Person` object is created.	`public Person(String name, int age) {`
The keyword `this` refers to the current object's instance variables. It differentiates instance variables from method parameters.	`this.name = name;` `this.age = age;`

Description	Example
Close curly braces to end the class definition.	`}`
Use the Java public method (Getter) to obtain read access to private variables.	`public String getName() {`
`getName()` returns the value of `name`.	`return name;`
Close curly braces to end the class definition.	`}`
Use the Java public method (Setter) to obtain write access to private variables.	`public void setName(String name) {`
`setName()` updates `name`.	`this.name = name;`
Use the Java public method (Getter) to obtain read access to private variables.	`public int getAge() {`
`getAge()` returns the value of `age`.	`return age;`
Close curly braces to end the class definition.	`}`
Use the Java public method (Setter) to obtain write access to private variables.	`public void setAge(int age) {`
Use the Java `if` statement to ensure `age` is not negative before assigning.	`if (age >= 0) {`
Update the age variable.	`this.age = age;`
Use the Java `else` statement to specify what to do when the age is negative.	`} else {`
Print the string `Age cannot be negative` to the console using the `System.out.println()` function.	`System.out.println("Age cannot be negative.");`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates a `Person` class in which the `name` and `age` attributes are declared as private, meaning they cannot be accessed directly from outside the `Person` class. The constructor `Person(String name, int age)` initializes the attributes when a new object of the class is created. `getName()` and `getAge()` are getter methods that allow other classes to read the values of `name` and `age`. `setName(String name)` and `setAge(int age)` are setter methods that allow other classes to modify the values of `name` and `age`. The setter for `age` includes validation to ensure `age` cannot be set to a negative number.

Using an encapsulated class

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create a new instance of the `Person` class. Assign the value "Alice" to the `name` attribute and the value "30" to the `age` attribute.	`Person person = new Person("Alice", 30);`
Use the `getName()` getter to obtain and print the value of the `name` attribute.	`System.out.println("Name: " + person.getName());`
Use the `getAge()` getter to obtain and print the value of the `age` attribute.	`System.out.println("Age: " + person.getAge());`

Description	Example
Use the `setName()` setter to assign the value of `name` attribute to "Bob" and `age` attribute to "25".	`person.setName("Bob");` `person.setAge(25);`
Use the `getName()` getter to obtain and print the updated value of the `name` attribute.	`System.out.println("Updated Name: " + person.getName());`
The `setAge(-5)` call attempts to set an invalid age. Since `setAge()` has validation logic, it will print "Age cannot be negative."	`System.out.println("Updated Age: " + person.getAge());`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates an instance of the `Person` class with the name "Alice" and age "30". We call the `getName()` and `getAge()` getter methods to print the values. We then update the name and age attributes usint the `setName()` and `setAge()` setter methods. When we attempt to set a negative age with setAge(-5), it prints an error massage because of validation included in the setter method.

Using constructors

Creating a default constructor

Description	Example
A Java statement used to define a class named `Dog`, which acts as a blueprint for creating `Dog` objects.	`class Dog {`
A Java statement to declare a `String` variable named `name` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String name;`
This is the default constructor. It takes no arguments.	`Dog() {`
The default constructor initializes the `name` variable with the value "Unknown". This ensures every new `Dog` object always has a name, even if the user doesn't provide one.	`name = "Unknown";`
Close curly braces to end the class definition.	`}`
Call the `display()` method without any access modifier.	`void display() {`
Print the dog's name to the console using the `System.out.println()` function. Since `name` was initialized in the constructor, it always has a value.	`System.out.println("Dog's name: " + name);`
Close curly braces to end the class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of the `Dog` class using the default constructor. The `name` variable is automatically set to "Unknown".	`Dog myDog = new Dog();`
Call the `display()` method to print the dog's name.	`myDog.display();`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates an instance of the `Dog` class with a default constructor that initializes the `name` attribute to "Unknown". When we create the instance, the default constructor is invoked automatically.

Creating a parameterized constructor

Description	Example
A Java statement used to define a class named `Dog`, which acts as a blueprint for creating `Dog` objects.	`class Dog {`
A Java statement to declare a `String` variable named `name` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String name;`
This is the parameterized constructor that takes one argument `dogName`.	`Dog(String dogName) {`

Description	Example
When the Dog object is created, the provided dogName value is assigned to the name variable. Parameterized constructors let you assign a unique name to each Dog object when it is created.	`name = dogName;`
Close curly braces to end the class definition.	`}`
Call the `display()` method without any access modifier.	`void display() {`
Print the dog's name to the console using the `System.out.println()` function. Since name was initialized in the constructor, it always has a value.	`System.out.println("Dog's name: " + name);`
Close curly braces to end the class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of the `Dog` class. "Buddy" is passed as an argument to the constructor, setting name to "Buddy".	`Dog myDog = new Dog("Buddy");`
Call the `display()` method to print the dog's name.	`myDog.display();`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates an instance of the Dog class with a parameterized constructor that takes a `string` parameter `dogName`. When we create a Dog instance with the name "Buddy", the constructor initializes the name attribute with that value.

Creating a no-arg constructor

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects.	`class Car {`
A Java statement to declare a `String` variable named `model` and an `int` variable named `year` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String model;` `int year;`
This is a no-argument constructor that takes no parameters.	`Car() {`
When the `Car` object is created, it automatically assigns the value "Default Model" to `model` and 2020 to `year`.	`model = "Default Model";` `year = 2020;`
Close curly braces to end the class definition.	`}`
Call the `display()` method without any access modifier.	`void display() {`
Print the car's model and year to the console using the `System.out.println()` function.	`System.out.println("Car Model: " + model + ", Year: " + year);`
Close curly braces to end the class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`

Description	Example
Create an instance of the `Car` class. The no-argument constructor is called, setting `model` = "Default Model" and `year` = 2020.	`Car myCar = new Car();`
Call the `display()` method to print the model and year of the car.	`myCar.display();`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates an instance of the `Car` class with two attributes `model` and `year`. The Car() constructor initializes the model to "Default Model" and year to 2020. When we create an instance of the `Car` class with new `Car()`, the no-arg constructor is called automatically, and the default values are assigned to the attributes. The `display()` method prints the `model` and year of the car.

Constructor overloading

Description	Example
A Java statement used to define a class named `Dog`, which acts as a blueprint for creating `Dog` objects.	`class Dog {`
A Java statement to declare a `String` variable named `name` and an `int` variable named `age` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String name;` `int age;`
This is the default constructor. It takes no arguments.	`Dog() {`
The default constructor initializes the `name` variable with the value "Unknown" and `age` variable with the value 0. This ensures every new `Dog` object always has a name and age, even if the user doesn't provide one.	`name = "Unknown";` `age = 0;`
Close curly braces to end the class definition.	`}`
This is the parameterized constructor that takes one argument `dogName`.	`Dog(String dogName) {`
When the `Dog` object is created, the provided `dogName` value is assigned to `name` while keeping the `age` as 0 by default. Parameterized constructors let you assign a unique name to each `Dog` object when it is created.	`name = dogName;` `age = 0;`
Close curly braces to end the class definition.	`}`
This is the parameterized constructor that takes two arguments `dogName` and `dogAge`.	`Dog(String dogName, int dogAge) {`
When the `Dog` object is created, the constructor allows the user to specify both `name` and `age`.	`name = dogName;` `age = dogAge;`
Close curly braces to end the class definition.	`}`
Call the `display()` method without any access modifier.	`void display() {`
Print the dog's `name` and `age` to the console using the `System.out.println()` function. Since `name` and `age` were initialized in the constructor, they always have a value.	`System.out.println("Dog's name: " + name + ", Age: " + age);`
Close curly braces to end the class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create the `dog1` object using the default constructor `Dog()`. So, `name` = "Unknown" and `age` = 0.	`Dog dog1 = new Dog();`

Description	Example
Create the dog2 object using the one-parameter constructor `Dog("Charlie")`. So, name = "Charlie" and age = 0 (default).	`Dog dog2 = new Dog();`
Create the dog3 object using the two-parameter constructor `Dog("Max", 5)`. So, name = "Max" and age = 5.	`Dog dog3 = new Dog();`

Call the `display()` method on each object to print their details.	`dog1.display();`
	`dog2.display();`
	`dog3.display();`

Close curly braces to end the class definition.	`}` `}`

Explanation: This example has three constructors of the Dog class. Depending on the number of parameters provided when creating an object, the corresponding constructor is called.

- Inheritance in Java
- Polymorphism in Java
- Interfaces and abstract classes in Java
- Inner classes in Java

Keep this summary reading available as a reference as you progress through your course, and refer to this reading as you begin coding with Java after this course!

Inheritance in Java

Creating a superclass

Description	Example
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`
Define a String variable name to store the name of the animal.	`String name;`
Include a method `eat()` to print the message that the animal is eating.	`void eat() {`
Print the message to the console using the `System.out.println()` function. The animal name is displayed dynamically.	`System.out.println(name + " is eating.");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Creating a subclass

Description	Example
The Dog class inherits from the `Animal` class , meaning it automatically gets all properties and methods from `Animal`.	`class Dog extends Animal {`
Include a method `bark()` to print the message that the dog is barking.	`void bark() {`
Print the message to the console using the `System.out.println()` function. The animal name is displayed dynamically.	`System.out.println(name + " says woof!");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Using inheritance

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Creates an instance of the Dog class. The Dog class inherits from the `Animal` class.	`Dog myDog = new Dog();`
Assigns "Buddy" to the name variable inherited from `Animal`.	`myDog.name = "Buddy";`
Calls the `eat()` method from the `Animal` class, which prints "Buddy is eating.".	`myDog.eat();`

Description	Example
Calls the `bark()` method from the `Dog` class, which prints "Buddy says woof!".	`myDog.bark();`
Close curly braces to end the `Main` class definition.	`}` `}`

Using multilevel inheritance

Description	Example
The `Puppy` class inherits from the `Dog` class. Since `Dog` already inherits from `Animal`, `Puppy` indirectly inherits all properties and methods from `Animal` as well.	`class Puppy extends Dog {`
This method adds a new behavior specific to the `Puppy` class.	`void weep() {`
Print the message to the console using the `System.out.println()` function. The animal `name` is displayed dynamically.	`System.out.println(name + " is weeping.");`
Close curly braces to end the `Puppy` class definition.	`}` `}`

Explanation: This is an example of multilevel inheritance. Animal (Superclass) → Dog (Subclass) → Puppy (Subclass of Dog). The `Animal` class has attribute `name` and method `eat()`. The `Dog` class inherits from `Animal` and adds the `bark()` method. `Puppy` inherits from `Dog` and adds the `weep()` method.

Using hierarchical inheritance

Description	Example
The `Cat` class inherits from the `Animal` class. Since `Animal` contains the `name` variable and `eat()` method, `Cat` inherits those properties.	`class Cat extends Animal {`
This method adds a new behavior specific to the `Cat` class.	`void meow() {`
Print the message to the console using the `System.out.println()` function. The animal `name` is displayed dynamically.	`System.out.println(name + " says meow!");`
Close curly braces to end the `Cat` class definition.	`}` `}`

Explanation: This is an example of hierarchical inheritance because multiple subclasses (`Dog` and `Cat`) inherit from the same superclass (`Animal`). `Animal` has attribute `name` and method `eat()`. `Dog` and `Cat` inherit from `Animal`, but each adds unique behaviors. `Dog` adds the `bark()` method and `Cat` adds the `meow()` method.

Method overriding

Description	Example
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`
Include a `sound()` method. This method is meant to be overridden by subclasses that define more specific behavors.	`void sound() {`
Print the message "Animal makes a sound" to the console using the `System.out.println()` function.	`System.out.println("Animal makes a sound");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Description	Example
The `Dog` class inherits from the `Animal` class.	`class Dog extends Animal {`
`Dog` overrides the `sound()` method to provide a specific implementation: "Dog barks". The `@Override` annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`@Override`
Include a `sound()` method to print the message "Dog barks".	`void sound() {`
Print the message to the console using the `System.out.println()` function.	`System.out.println("Dog barks");`
Close curly braces to end the `Dog` class definition.	`}` `}`

Description	Example

Explanation: In this example, `Dog` provides its own implementation of `sound()`, replacing the one in `Animal`. Method overriding occurs when a subclass provides a specific implementation of a method already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass.

Using overridden methods

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Creates an instance of `Animal` and stores it in a variable `myAnimal`.	`Animal myAnimal = new Animal();`
The `Dog` object is stored in an `Animal` reference. Since `Dog` overrides the `sound()` method, Java uses dynamic method dispatch to call the overridden method in `Dog`, not in `Animal`.	`Animal myDog = new Dog();`
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the `Animal` class.	`myAnimal.sound();`
Since `myDog` refers to a `Dog` object (even though it's declared as `Animal`), it calls the overridden `sound()` method in `Dog` due to polymorphism.	`myDog.sound();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: The `Dog` class inherits from `Animal`, meaning it gets all non-private properties and methods of `Animal`. `Dog` overrides the `sound()` method from `Animal`, providing a more specific implementation. Even though `myDog` is declared as an `Animal`, Java determines the method to call at runtime, not compile time. When calling `myDog.sound()`, Java looks at the actual object type (`Dog`) and calls `sound()` from `Dog`, not `Animal`.

Polymorphism in Java

Compile-time polymorphism

Description	Example
Create a class `MathOperations` that contains multiple methods for performing addition.	`class MathOperations {`
Include an add method that accepts two `int` values (a and b).	`int add(int a, int b) {`
Add the values of a and b and return the sum to the calling method as an `int`.	`return a + b;`
Close curly braces to end the method.	`}`
Include an add method that accepts three `int` values (a, b, and c).	`int add(int a, int b, int c) {`
Add the values of a, b, and c and return the sum to the calling method as an `int`. This method overloads the first `add()` method because it has different number of parameters.	`return a + b + c;`
Close curly braces to end the method.	`}`
Include an add method that accepts two `double` values (a and b).	`int add(double a, double b) {`
Add the values of a and b and return the sum to the calling method as a `double`. This method overloads both of the previous `add()` methods, but it works with double values instead of int.	`return a + b;`
Close curly braces to end the method and the `MathOperations` class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`

Description	Example
Create an instance of the `MathOperations` class and assign it to the `math` object.	`MathOperations math = new MathOperations();`
Calls the method `add(int a, int b)` to add two integers (2 + 3) and print the result to the console.	`System.out.println("Sum of 2 and 3: " + math.add(2, 3));`
Calls the method `add(int a, int b, int c)` to add three integers (2 + 3 + 4) and print the result to the console.	`System.out.println("Sum of 2, 3 and 4: " + math.add(2, 3, 4));`
Calls the method `add(double a, double b)` to add two double values (2.5 + 3.5) and print the result to the console.	`System.out.println("Sum of 2.5 and 3.5: " + math.add(2.5, 3.5));`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: The `add()` method is overloaded three times in the `MathOperations` class. Different number of parameters (int a, int b) versus (int a, int b, int c) and different types of parameters (int versus double). In Java, overloading is based on the method signature, which includes the number and types of parameters. It does not depend on the return type. The correct method is selected at compile time based on the arguments passed to the `add()` method. This is an example of compile-time polymorphism (or static polymorphism).

Using compile-time polymorphism

Description	Example
Create a class `MathOperations` that contains multiple methods for performing addition.	`class MathOperations {`
Include an add method that accepts two `int` values (a and b).	`int add(int a, int b) {`
Add the values of a and b and return the sum to the calling method as an `int`.	`return a + b;`
Close curly braces to end the method.	`}`
Include an add method that accepts two `double` values (a and b).	`int add(double a, double b) {`
Add the values of a and b and return the sum to the calling method as a `double`. This method overloads both of the previous `add()` methods, but it works with double values instead of int.	`return a + b;`
Close curly braces to end the method.	`}`
Include an add method that accepts three `int` values (a, b, and c).	`int add(int a, int b, int c) {`
Add the values of a, b, and c and return the sum to the calling method as an `int`. This method overloads the first `add()` method because it has different number of parameters.	`return a + b + c;`
Close curly braces to end the method and the `MathOperations` class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of the `MathOperations` class and assign it to the `math` object.	`MathOperations math = new MathOperations();`
Calls the method `add(int a, int b)` to add two integers (2 + 3) and print the result to the console.	`System.out.println("Sum of 2 and 3: " + math.add(2, 3));`
Calls the method `add(double a, double b)` to add two double values (2.5 + 3.5) and print the result to the console.	`System.out.println("Sum of 2.5 and 3.5: " + math.add(2.5, 3.5));`
Calls the method `add(int a, int b, int c)` to add three integers (2 + 3 + 4) and print the result to the console.	`System.out.println("Sum of 1, 2 and 3: " + math.add(2, 3, 4));`

Description	Example
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: In this example, the `MathOperations` class has three overloaded `add` methods. Depending on the number and type of arguments passed to `add`, Java determines which method to invoke at compile time. This makes our code more flexible and easier to read.

Using runtime polymorphism

Description	Example
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`
Include a `sound()` method. This method is meant to be overridden by subclasses that define more specific behaviors.	`void sound() {`
Print the message "Animal makes a sound" to the console using the `System.out.println()` function.	`System.out.println("Animal makes a sound");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Description	Example
The `Dog` class inherits from the `Animal` class.	`class Dog extends Animal {`
`Dog` overrides the `sound()` method to provide a specific implementation: "Dog barks". The `@Override` annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`@Override`
Include a `sound()` method to print the message "Dog barks".	`void sound() {`
Print the message to the console using the `System.out.println()` function.	`System.out.println("Dog barks");`
Close curly braces to end the `Dog` class definition.	`}` `}`

Description	Example
The `Cat` class inherits from the `Animal` class.	`class Cat extends Animal {`
`Cat` overrides the `sound()` method to provide a specific implementation: "Cat meows". The `@Override` annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`@Override`
Include a `sound()` method to print the message "Cat meows".	`void sound() {`
Print the message to the console using the `System.out.println()` function.	`System.out.println("Cat meows");`
Close curly braces to end the `Cat` class definition.	`}` `}`

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Creates an instance of `Animal` and stores it in a variable `myAnimal`.	`Animal myAnimal = new Animal();`
The `Dog` object is stored in an `Animal` reference. Since `Dog` overrides the `sound()` method, Java uses dynamic method dispatch to call the overridden method in `Dog`, not in `Animal`.	`myAnimal = new Dog();`

Description	Example
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the `Animal` class.	`myAnimal.sound();`
The `Cat` object is stored in an `Animal` reference. Since Cat overrides the sound() method, Java uses dynamic method dispatch to call the overridden method in `Cat`, not in `Animal`.	`myAnimal = new Cat();`
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the `Animal` class.	`myAnimal.sound();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: In this example, `Animal` is a superclass with a method called `sound()`. Both `Dog` and `Cat` classes extend `Animal`, providing their own implementation of the `sound()` method. When we create an `Animal` reference and assign it to different subclasses (Dog and Cat), the appropriate `sound()` method is called at runtime based on the object type. This allows for more dynamic and flexible code.

Creating virtual methods

Description	Example
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`
Include a `sound()` method. This method is meant to be overridden by subclasses that define more specific behaviors.	`void sound() {`
Print the message "Animal makes a sound" to the console using the `System.out.println()` function.	`System.out.println("Animal makes a sound");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Description	Example
The `Dog` class inherits from the `Animal` class.	`class Dog extends Animal {`
`Dog` overrides the `sound()` method to provide a specific implementation: "Dog barks". The `@Override` annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`@Override`
Include a `sound()` method to print the message "Dog barks".	`void sound() {`
Print the message to the console using the `System.out.println()` function.	`System.out.println("Dog barks");`
Close curly braces to end the `Dog` class definition.	`}` `}`

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Creates an instance of `Animal` and stores it in a variable `myAnimal`.	`Animal myAnimal = new Dog();`
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the `Animal` class.	`myAnimal.sound();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: In this example, even though `myAnimal` is an `Animal`, the `sound()` method from the `Dog` class is called, demonstrating virtual method behavior.

Designing interfaces and Abstract Classes in Java

Creating an interface

Description	Example
Declare an `Animal` interface.	`interface Animal {`

Description	Example
Include a method `sound()`. Any class that implements this interface must provide an implementation of `sound()`.	`void sound();`
Close curly braces to end the interface definition.	`}`

Description	Example
Create a Dog class that implements the `Animal` interface.	`class Dog implements Animal {`
Include a `sound()` method for the class.	`public void sound() {`
Calling `sound()` prints "Bark" to the console using the `System.out.println()` function.	`System.out.println("Bark");`
Close curly braces to end the Dog class definition.	`}` `}`

Description	Example
Create a `Cat` class that implements the `Animal` interface.	`class Cat implements Animal {`
Include a `sound()` method for the class.	`public void sound() {`
Calling `sound()` prints "Meow" to the console using the `System.out.println()` function.	`System.out.println("Meow");`
Close curly braces to end the `Cat` class definition.	`}` `}`

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create the `Dog` object and assign it to the variable `dog`.	`Animal dog = new Dog();`
Create the `Cat` object and assign it to the variable `cat`.	`Animal cat = new Cat();`
Call `sound()` on the `dog` object. This prints the message "Bark".	`dog.sound();`
Call `sound()` on the `cat` object. This prints the message "Meow".	`cat.sound();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: In this example, we define an interface `Animal` with a method `sound()`. The `Dog` and `Cat` classes implement the `Animal` interface and provide their own versions of the `sound()` method. In the `Main` class, we create instances of `Dog` and `Cat`, calling the `sound()` method on each to demonstrate polymorphism.

Creating an abstract class

Description	Example
Create an abstract class `Shape` that cannot be instantiated directly.	`abstract class Shape {`
Include an abstract method `draw()` that must be implemented by any subclass.	`abstract void draw();`
Include a concrete method `display()` that has a default implementation.	`void display() {`

Description	Example
Calling the `display()` method prints "This is a shape." to the console using the `System.out.println()` function.	`System.out.println("This is a shape.");`
Close curly braces to end the Dog class definition.	`}` `}`

Description	Example
Create a `Circle` class that extends the `Shape` class.	`class Circle extends Shape {`
Include a `draw()` method for the class.	`public void draw() {`
Calling the `draw()` method prints "Drawing Circle" to the console using the `System.out.println()` function.	`System.out.println("Drawing Circle");`
Close curly braces to end the Dog class definition.	`}` `}`

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
The `shape` object is instantiated from the `Shape` class but it refers to a `Circle` object.	`Shape shape = new Circle();`
Calling `draw()` on the `shape` object prints "Drawing Circle".	`shape.draw();`
Calling `display()` on the `shape` object prints "This is a shape."	`shape.display();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: In this example, we define an abstract class `Shape` with an abstract method `draw()` and a concrete method `display()`. The `Circle` class extends the `Shape` class and provides an implementation for the `draw()` method. In the `Main` class, we create an instance of `Circle` using the `Shape` reference type to show how it works. The `draw()` method executes the overridden version from `Circle`. The `display()` method is inherited from `Shape` and is called as is.

Inner classes in Java

Creating inner classes

Description	Example
Create an `OuterClass` that works as a container for the inner class.	`class OuterClass {`
Set the value of the `int outerVariable` to 10.	`int outerVariable = 10;`
Create a class `InnerClass` inside the `OuterClass`.	`class InnerClass {`
Include a method `display()` that accesses `OuterVariable` from the outer class. Inner classes have direct access to the outer class's members (including private ones).	`void display();`
Calling the `display()` method prints the `outerVariable` value to the console using the `System.out.println()` function. The `outerVariable` value is generated dynamically.	`System.out.println("Outer variable value: " + outerVariable);`
Close curly braces to end the `OuterClass` class definition.	`}` `}` `}`

Explanation: In this example, `OuterClass` contains a variable `outerVariable`. `InnerClass` is defined inside `OuterClass` and has a method `display()`. This method can access `outerVariable` directly.

Using inner classes

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`

Description	Example
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of the `OuterClass`. This is necessary because non-static inner classes require an instance of the outer class to be created first.	`OuterClass outer = new OuterClass();`
Create a class `InnerClass` inside the `OuterClass`. Since `InnerClass` is a non-static inner class, it must be created using an instance of `OuterClass`.	`OuterClass.InnerClass inner = outer.new InnerClass();`
Call the `display()` method inside `InnerClass`.	`inner.display();`
Close curly braces to end the `Main` class definition.	`    }` `}`

Explanation: In this example, `InnerClass` is nested inside `OuterClass` and has access to all outer class's members. The `display()` method will print the value of the `outerVariable`. The code demonstrates encapsulation in Java.

Creating a static nested classes

Description	Example
Create an `OuterClass` that works as a container for the inner class.	`class OuterClass {`
Set the value of the `int outerVariable` to 20.	`static int staticVariable = 20;`
Create a class `InnerClass` inside the `OuterClass`.	`static class StaticNestedClass {`
Include a method `show()` that accesses `OuterVariable` from the outer class. Inner classes have direct access to the outer class's members (including private ones).	`void show();`
Calling the `show()` method prints the `outerVariable` value to the console using the `System.out.println()` function. The `outerVariable` value is generated dynamically.	`System.out.println("Static variable value: " + staticVariable);`

Close curly braces to end the `OuterClass` class definition.	`    }`
	`}`
	`}`

Explanation: In this example, `OuterClass` contains a static variable named `staticVariable` with a value of 20. Since the variable is static, it belongs to the class itself rather than an instance. Static nested classes do not require an instance of the outer class. It can access `staticVariable` without an instance of `OuterClass`. The nested class keeps related logic inside `OuterClass`, improving organization.

Using a static nested classes

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of `StaticNestedClass` inside the `OuterClass`.	`OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();`
Include a method `nested.show()` that prints the value of the `staticVariable` from `OuterClass`.	`nested.show();`

Close curly braces to end the `OuterClass` class definition.	`    }`
	`}`
	`}`

Creating a method-local inner class

Description	Example
Create an `OuterClass` with a method `myMethod()` that will define and use a method-local inner class.	`class OuterClass {` `void myMethod() {`

Define a class `MethodLocalInner` inside `myMethod()`. `MethodLocalInner` is local to the method, meaning that it cannot be accessed outside of `myMethod()`. Calling `MethodLocalInner` prints the message "Inside Method Local Inner Class" to the console using the `System.out.println()` function.	`class MethodLocalInner {`
	`void display() {`
	`System.out.println("Inside Method Local Inner Class");`
	`}`
	`}`

The inner class is instantiated within the method where it is defined.	`MethodLocalInner inner = new MethodLocalInner();`

Description	Example
`inner.display()` calls the `display()` method, printing "Inside Method Local Inner Class".	`inner.display();`
Close curly braces to end the `OuterClass` class definition.	`}` `}`

Creating an anonymous inner class

Description	Example

The `Greeting` interface defines a single method `greet()`, which must be implemented by any class that uses this interface.	`interface Greeting {`
	`void greet();`
	`}`

This creates an anonymous inner class that implements the `Greeting` interface. The anonymous class provides an implementation for the `greet()` method at the moment of object creation.	`public class Main {`
	`public static void main(String[] args) {`
	`Greeting greeting = new Greeting() {`
	`public void greet() {`
	`System.out.println("Hello from Anonymous Inner Class!");`
	`}`
	`};`

This calls the overridden `greet()` method in the anonymous inner class, printing "Hello from Anonymous Inner Class!".	`greeting.greet();`
Close curly braces to end the `Main` class definition.	`}` `}`

Using inner classes in the real world

Description	Example

The `Library` class represents a library and has a private variable `libraryName` to store its name. A constructor initializes `libraryName`.	`class Library {`
	`private String libraryName;`
	`public Library(String name) {`
	`this.libraryName = name;`
	`}`

Nested inside `Library`, this class represents a book. It has two private attributes: `title` and `author`. The Book class has a constructor to initialize these attributes. The `displayBookInfo()` method prints the book's title and author. It also accesses `libraryName` from `Library`, demonstrating how inner classes can access private members of the outer class.	`class Book {`
	`private String title;`
	`private String author;`
	`public Book(String title, String author) {`
	`this.title = title;`
	`this.author = author;`
	`}`
	`public void displayBookInfo() {`
	`System.out.println("Library: " + libraryName);`
	`System.out.println("Book Title: " + title);`
	`System.out.println("Author: " + author);`
	`}`
	`}`

This creates a `Library` instance named "City Library" and creates a Book instance associated with that library. Since Book is a non-static inner class, it must be created using an instance of Library. The `displayBookInfo()` method in the Book inner class prints out the name of the library along with the book's title and author.	`public class Main {`
	`public static void main(String[] args) {`
	`Library myLibrary = new Library("City Library");`
	`Library.Book myBook = myLibrary.new Book("1984", "George Orwell");`
	`myBook.displayBookInfo();`

Close curly braces to end the `Main` class definition.	`}` `}`

Java Collections Framework (JCF)

Using an ArrayList array

Description	Example
Import `ArrayList` and `List` from the java.util package to use dynamic lists.	`import java.util.ArrayList;` `import java.util.List;`

Define a class `ListExample` that contains the Java `main` method. Create a `List` of type `String` using the `ArrayList` implementation. This list will store fruit names as string elements. Add elements "Apple", "Banana", and "Cherry" to the list. Print the entire list, showing its elements in the order they were added. Retrieve the first element `Apple` from the list using index 0. Print the retrieved element.	`public class ListExample {`
	`public static void main(String[] args) {`
	`List<String> fruits = new ArrayList<>();`
	`fruits.add("Apple");`
	`fruits.add("Banana");`
	`fruits.add("Cherry");`
	`System.out.println("Fruits: " + fruits);`
	`String firstFruit = fruits.get(0);`
	`System.out.println("First fruit: " + firstFruit);`

Close curly braces to end the `ListExample` class definition.	`}` `}`

Explanation: This Java program demonstrates how to use the `List` interface with an `ArrayList` implementation to store and manipulate a list of fruit names. `ArrayList` is a dynamic array-based implementation of `List`, allowing for flexible resizing. Elements are added in order and accessed using a zero-based index. The `get(index)` method retrieves elements at specific positions.

Using a LinkedList array

Description	Example
Import the `LinkedList` class from the java.util package to use a linked list.	`import java.util.LinkedList;`

Define a class `LinkedListExample` that contains the Java main method. Create a `LinkedList` of type `String` to store animal names. Add elements "Dog", "Cat", and "Elephant" to the list. Print the contents of the `LinkedList`, displaying all elements.	`public class LinkedListExample {`
	`public static void main(String[] args) {`
	`LinkedList<String> animals = new LinkedList<>();`
	`animals.add("Dog");`
	`animals.add("Cat");`
	`animals.add("Elephant");`
	`System.out.println("Animals: " + animals);`

Description	Example
Close curly braces to end the `LinkedListExample` class definition.	`}` `}`

Explanation: This Java program demonstrates how to use a `LinkedList` to store and manipulate a list of animal names. `LinkedList` is a doubly linked list implementation in Java, meaning that elements are linked using pointers. In `LinkedList`, insertions and deletions are faster compared to `ArrayList` (especially for large lists).

Using a HashSet collection

Description	Example
Import the `HashSet` class from the java.util package to store a collection of unique elements.	`import java.util.HashSet;`

Define a class `HashSetExample` that contains the Java `main` method. Create a `HashSet` of type `String` to store color names. Add elements "Red", "Green", and "Blue" to the HashSet. Add "Red" again to the HashSet. HashSet does not allow duplicate values. If a duplicate is added, it is ignored. Print the contents of the `HashSet`, displaying all elements.	`public class HashSetExample {`
	`public static void main(String[] args) {`
	`HashSet<String> colors = new HashSet<>();`
	`colors.add("Red");`
	`colors.add("Green");`
	`animals.add("Blue");`
	`colors.add("Red");`
	`System.out.println("Colors: " + colors);`

Close curly braces to end the `HashSetExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `HashSet`, which is a part of the Java Collections Framework and is used to store a collection of unique elements. `HashSet` does not maintain any specific order and ignores duplicates. It is useful when you need a collection of distinct elements with fast lookup times.

Using a HashMap collection

Description	Example
Import the `HashMap` class from the java.util package to store key-value pairs.	`import java.util.HashMap;`

Define a class `HashMapExample` that contains the Java `main` method. Create a `HashMap<String, Integer>` named `ageMap`. The keys are names (String), and the values are ages (Integer). Add key-value pair to the `HashMap` using the `put()` method. The `System.out.println()` statement prints the entire `HashMap` but does not maintain any order because `HashMap` does not maintain insertion order. The program retrieves Alice's age using `ageMap.get("Alice")` and stores it in `aliceAge`.	`public class HashMapExample {`
	`public static void main(String[] args) {`
	`HashMap<String, Integer> ageMap = new HashMap<>();`
	`ageMap.put("Alice", 30);`
	`ageMap.put("Bob", 25);`
	`ageMap.put("Charlie", 35);`
	`System.out.println("Age Map: " + ageMap);`
	`int aliceAge = ageMap.get("Alice");`
	`System.out.println("Alice's Age: " + aliceAge);`

Close curly braces to end the `HashMapExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `HashMap`, which is a part of the Java Collections Framework and is used to store key-value pairs. Keys are unique (if a duplicate key is added, it replaces the old value). Values can be duplicated. `HashMap` does not maintain any specific order. It provides fast access to values using keys.

Working with lists

Creating an ArrayList

Description	Example
Import `ArrayList` from the java.util package to use dynamic lists.	`import java.util.ArrayList;`

Define a class `ArrayListExample` that contains the Java `main` method. Create an `ArrayList<String>` named `fruits` to store a list of fruit names. This list will store fruit names as string elements. Add elements "Apple", "Banana", and "Cherry" to the list. Print the entire list, showing its elements in the order they were added. Retrieve the first element `Apple` from the list using index `0` and print the retrieved element. Call `fruits.remove("Banana")` to remove "Banana" from the list. Print the remaining elements of `ArrayList`.	`public class ArrayListExample {`
	`public static void main(String[] args) {`
	`ArrayList<String> fruits = new ArrayList<>();`
	`fruits.add("Apple");`
	`fruits.add("Banana");`
	`fruits.add("Cherry");`
	`System.out.println("First fruit: " + fruits.get(0));`
	`fruits.remove("Banana");`
	`System.out.println("Fruits List: " + fruits);`

Close the curly braces to end the `ArrayListExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of an `ArrayList`, which is a part of the Java Collections Framework and is used to store a resizable list of elements. `ArrayList` elements are added in order and accessed using a zero-based index. The `get(index)` method retrieves elements at specific positions. `ArrayList` allows duplicates and removing elements shifts subsequent elements left (affecting performance for large lists).

Creating a LinkedList

Description	Example
Import the `LinkedList` class from the java.util package to create a linked list.	`import java.util.LinkedList;`

Define a class `LinkedListExample` that contains the Java `main` method. Create a `LinkedList` of type `String` to store a list of color names. Add elements "Red", "Green", and "Blue" to the list using the `add()` method. Retrieve the first element of the list using `colors.get(0)`. Remove the first occurrence of "Green" from the list using `colors.remove("Green")`. Print the remaining elements of the `LinkedList`.	`public class LinkedListExample {`
	`public static void main(String[] args) {`
	`LinkedList<String> colors = new LinkedList<>();`
	`colors.add("Red");`
	`colors.add("Green");`
	`animals.add("Blue");`
	`System.out.println("First color: " + colors.get(0));`
	`colors.remove("Green");`
	`System.out.println("Colors List: " + colors);`

Close the curly braces to end the `LinkedListExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `LinkedList`, which is a part of the Java Collections Framework. `LinkedList` stores elements in nodes, where each node contains a reference to the next node. It allows efficient insertion and removal of elements from both ends: `addFirst()`, `addLast()`, `removeFirst()`, and `removeLast()`. Accessing elements by index `get(index)` is slower than in `ArrayList`, because it requires traversing the list from the beginning. Duplicates are allowed, and order is maintained. Unlike ArrayList, elements are not shifted after removal (only the references are updated), which can improve performance for certain operations.

HashSet and TreeSet

Creating a HashSet

Description	Example
Import the `HashSet` class from the java.util package to store a collection of unique elements.	`import java.util.HashSet;`

Define a class `HashSetExample` that contains the Java `main` method. Create a `HashSet` of type `String` to store fruit names. Add elements "Apple", "Banana", and "Cherry" to the HashSet. Add "Banana" again to the `HashSet`. Since `HashSet` does not allow duplicate values, it is ignored. Print the contents of the `HashSet`, displaying all elements. Checks if "Apple" is in the set by calling `fruits.contains("Apple")`. If found, the message "Apple" is present in the set is printed. The method `fruits.remove("Cherry")` removes "Cherry" from the set.	`public class HashSetExample {`
	`public static void main(String[] args) {`
	`HashSet<String> fruits = new HashSet<>();`
	`fruits.add("Apple");`
	`fruits.add("Banana");`
	`fruits.add("Cherry");`
	`fruits.add("Banana");`
	`System.out.println("Fruits in the HashSet: " + fruits);`
	`if (fruits.contains("Apple")) {`
	`System.out.println("Apple is present in the set.");`
	`}`
	`fruits.remove("Cherry");`

Description	Example
	`System.out.println("After removal: " + fruits);`
Close curly braces to end the `HashSetExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `HashSet`, which is a part of the Java Collections Framework and is used to store a collection of unique elements. The `contains()` method provides fast lookup to check if an element exists. The `remove()` method efficiently removes elements. `HashSet` does not maintain any specific order and ignores duplicates. It is useful when you need a collection of distinct elements with fast lookup times.

Creating a TreeSet

Description	Example
Import the `TreeSet` class from the java.util package to store a collection of unique elements.	`import java.util.TreeSet;`

Define a class `TreeSetExample` that contains the Java `main` method. Create a `TreeSet<Integer>` named `numbers` to store a set of integer values. Add the numbers 5, 3, 8, and 1 using the `add()` method. Add 3 again to the `TreeSet`. Since `TreeSet` does not allow duplicate values, it is ignored. Print the contents of the `TreeSet`, displaying all elements. Checks if 5 is in the set by calling `numbers.contains(5)`. If found, the message "5 is present in the set" is printed. The method `numbers.remove(8)` removes 8 from the set.	`public class TreeSetExample {`
	`public static void main(String[] args) {`
	`TreeSet<Integer> numbers = new TreeSet<>();`
	`numbers.add(5);`
	`numbers.add(3);`
	`numbers.add(8);`
	`numbers.add(1);`
	`numbers.add(3);`
	`System.out.println("Numbers in the TreeSet: " + numbers);`
	`if (numbers.contains(5)) {`
	`System.out.println("5 is present in the set.");`
	`}`
	`numbers.remove(8);`
	`System.out.println("After removal: " + numbers);`

Close curly braces to end the `HashSetExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the usage of a `TreeSet`, which is used to store a collection of unique elements. `TreeSet` elements are always sorted in ascending order. The `contains()` method provides fast lookup (uses a Balanced Tree structure). The `remove()` methoded efficiently deletes elements while maintaining order. `TreeSet` is useful when you need a sorted set with fast operations.

TreeSet versus HashSet: need for order

Description	Example

Use TreeSet: When you need the elements to be sorted in a specific order. For example: If you want to store a list of student grades and display them in ascending order, a TreeSet will automatically sort them.	`TreeSet<Integer> grades = new TreeSet<>();`
	`grades.add(85);`
	`grades.add(90);`
	`grades.add(78);`
	`// Output: [78, 85, 90]`
	`System.out.println(grades);`

Use HashSet: When the order of elements does not matter. For example: If you are storing unique user IDs and do not care about their order.	`HashSet<String> userIds = new HashSet<>();`
	`userIds.add("user1");`
	`userIds.add("user2");`
	`userIds.add("user3");`
	`// Output: Order may vary`
	`System.out.println(userIds);`

HashSet versus TreeSet: Need for performance

Description	Example

Use HashSet: For faster performance when adding, removing, or searching for elements. For Example: In a game, if you need to quickly check if a player has collected a unique item.	`HashSet<String> collectedItems = new HashSet<>();`
	`collectedItems.add("Sword");`
	`collectedItems.add("Shield");`
	`<boolean>boolean hasSword = collectedItems.contains("Sword"); // Fast check`

Use TreeSet: When you can afford slower operations but need the elements sorted. For Example: If you are maintaining a leaderboard that requires sorted scores, a TreeSet is suitable even if it's slightly slower.	`TreeSet<Integer> scores = new TreeSet<>();`
	`scores.add(300);`
	`scores.add(150);`
	`scores.add(200);`
	`System.out.println(scores); // [150, 200, 300]`

HashSet versus TreeSet: Avoidance of duplicates

Description	Example

Using HashSet to avoid duplicates. A `HashSet<String>` named `fruits` is created. "Apple" and "Banana" are added. A duplicate "Apple" is added but ignored because `HashSet` does not allow duplicates. The output may appear as [Banana, Apple] or [Apple, Banana], but the order is NOT guaranteed, since `HashSet` is unordered.	`HashSet<String> fruits = new HashSet<>();`
	`fruits.add("Apple");`
	`fruits.add("Banana");`
	`fruits.add("Apple"); // Duplicate will not be added`
	`System.out.println(fruits); // Output: [Banana, Apple] </String>`

Using TreeSet to avoid duplicates. A `TreeSet<String>` named `sortedFruits` is created. "Apple" and "Banana" are added. A duplicate "Apple" is added but ignored because `TreeSet` also does not allow duplicates. Unlike `HashSet`, `TreeSet` automatically sorts elements in ascending order. The output is always [Apple, Banana], since `TreeSet` maintains sorted order.	`TreeSet<String> sortedFruits = new TreeSet<>();`
	`sortedFruits.add("Apple");`
	`sortedFruits.add("Banana");`
	`sortedFruits.add("Apple"); // Duplicate will not be added`
	`System.out.println(sortedFruits); // Output: [Apple, Banana]`

Implementing queues in Java

Creating a simple queue using LinkedList

Description	Example
Import the java.util.LinkedList and java.util.Queue packages to use the `Queue` interface with a `LinkedList` implementation.	`import java.util.LinkedList;` `import java.util.Queue;`

Create an instance of `Queue<String>` named `queue` using new `LinkedList<>()`. Add three elements ("Apple", "Banana", "Cherry") to the queue using `offer()`, which inserts elements at the end of the queue. Print the queue to show its contents. The `poll()` method removes and returns the front element ("Apple") from the queue. Print the removed element ("Apple") and display the state of the queue again after removing the front element.	`public class QueueExample {`
	`public static void main(String[] args) {`
	`// Creating a Queue`
	`Queue<String> queue = new LinkedList<>();`
	`// Enqueue operation`
	`queue.offer("Apple");`
	`queue.offer("Banana");`
	`queue.offer("Cherry");`
	`// Displaying the Queue`
	`System.out.println("Queue: " + queue);`
	`// Dequeue operation`
	`String removedItem = queue.poll();`
	`System.out.println("Removed Item: " + removedItem);`
	`// Displaying the Queue after Dequeue`
	`System.out.println("Queue after dequeue: " + queue);`

Close curly braces to end the `QueueExample` class definition.	`}` `}`

Explanation: This Java program demonstrates the use of a Queue data structure using the `LinkedList` class. The method `offer()` adds an element to the queue (enqueue). `poll()` removes and returns the front element (dequeue). `LinkedList` as a queue implements FIFO (First-In-First-Out) behavior.

Creating a priority queue

Description	Example
Import the java.util.PriorityQueue package to use the PriorityQueue class.	`import java.util.PriorityQueue;`

Create an instance of `PriorityQueue<Integer>` named `priorityQueue` using new `PriorityQueue<>()`. Add three elements: 20, 15, and 30 using the `offer()` method. The `PriorityQueue` maintains a min-heap structure (smallest element has the highest priority). Print the queue; its order may not be in the exact insertion order due to the heap-based priority structure. Remove elements in priority order (ascending order for integers). A while loop continuously removes and prints the smallest element until the queue is empty.	`public class PriorityQueueExample {`
	`  public static void main(String[] args) {`
	`  PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();`
	`  // Adding elements`
	`  priorityQueue.offer(20);`
	`  priorityQueue.offer(15);`
	`  priorityQueue.offer(30);`
	`  // Displaying the Priority Queue`
	`  System.out.println("Priority Queue: " + priorityQueue);`
	`  // Removing elements in priority order`
	`  while (!priorityQueue.isEmpty()) {`
	`  System.out.println("Removed Item: " + priorityQueue.poll());`
	`  }`

Close curly braces to end the `PriorityQueueExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates the usage of a PriorityQueue, which is a type of queue where elements are processed based on their priority (natural order by default for numbers). The method `offer()` adds an element to the queue (enqueue), `poll()` removes and returns the element with the highest priority (smallest number in this case). Heap-based Implementation ensures efficient insertions and deletions.

Implementing a queue in the real world

Description	Example
Import the java.util.LinkedList and java.util.Queue packages to create and manage the queue.	`import java.util.LinkedList;` `import java.util.Queue;`

Create an instance of `Queue<String>` named `customerQueue` using new `LinkedList<>()` to store customers. Add "Customer 1", "Customer 2", and "Customer 3" are added to the queue using `offer()`. Prints the queue to show the customers waiting in order. The `poll()` method removes and returns the first customer ("Customer 1") from the queue. Display the remaining customers in the queue. Call `poll()` again to serve the next customer and print the final state of the queue.	`public class CustomerServiceQueue {`
	`  public static void main(String[] args) {`
	`  // Creating a queue to represent customers waiting for service`
	`  Queue<String> customerQueue = new LinkedList<>();`
	`  // Adding customers to the queue`
	`  customerQueue.offer("Customer 1");`
	`  customerQueue.offer("Customer 2");`
	`  customerQueue.offer("Customer 3");`
	`  // Displaying the current queue`
	`  System.out.println("Current Customer Queue: " + customerQueue);`
	`  // Serving the first customer in the queue`
	`  String servedCustomer = customerQueue.poll();`
	`  System.out.println("Serving: " + servedCustomer);`
	`  // Displaying the queue after serving one customer`
	`  System.out.println("Customer Queue after serving one: " + customerQueue);`
	`  // Serving another customer`
	`  servedCustomer = customerQueue.poll();`
	`  System.out.println("Serving: " + servedCustomer);`
	`  // final state of the queue`
	`  System.out.println("Final Customer Queue: " + customerQueue);`

Close curly braces to end the `CustomerServiceQueue` class definition.	`  }` `}`

Explanation: This Java program simulates a customer service queue using a Queue (FIFO - First In, First Out) implemented with a `LinkedList`. It models how customers arrive, wait, and are served in order. The method `offer()` adds customers to the queue and `poll()` removes customers in FIFO order. `LinkedList` as a queue mimics a real world waiting line. This approach can be extended to simulate bank queues, call centers, or ticket counters.

Using HashMap and TreeMap

Creating a HashMap

Description	Example
Import the `HashMap` class from the java.util package, which is a part of Java's Collection Framework.	`import java.util.HashMap;`

Initialize a `HashMap<String, Integer>` named map to represent `fruit names` as keys and their corresponding `numeric values` as values. Add key-value pairs using the put method. "Apple" is mapped to 1, "Banana" to 2, and "Cherry" to 3. Keys are unique: If the same key is added again, its value gets updated. The `map.get("Apple")` method fetches and prints the value associated with "Apple". The `keySet()` method returns all the keys in the `HashMap`, and the `for` loop prints each key-value pair. Order is NOT guaranteed in a `HashMap`. The `containsKey()` method checks whether "Banana" is present in the map and the `remove()` method deletes "Cherry" from the `HashMap`.	`public class HashMapExample {`
	`  public static void main(String[] args) {`
	`  // Creating a HashMap`
	`  HashMap<String, Integer> map = new HashMap<>();`
	`  // Adding key-value pairs to the HashMap`
	`  map.put("Apple", 1);`
	`  map.put("Banana", 2);`
	`  map.put("Cherry", 3);`
	`  // Accessing values`
	`  System.out.println("Value for key 'Apple': " + map.get("Apple")); // Output: 1`
	`  // Iterating through the HashMap`
	`  for (String key : map.keySet()) {`
	`  System.out.println(key + ": " + map.get(key));`
	`  }`
	`  // Checking if a key exists`
	`  if (map.containsKey("Banana")) {`
	`  System.out.println("Banana exists in the map.");`
	`  }`
	`  // Removing a key-value pair`
	`  map.remove("Cherry");`

Close curly braces to end the `TreeMapExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates the usage of a `HashMap`, a data structure that stores key-value pairs and allows fast access to values using keys. `put(K key, V value)` adds or updates a key-value pair. `get(K key)` retrieves the value for a key, `keySet()` returns all keys, `containsKey(K key)` checks if a key exists, and `remove(K key)` deletes a key-value pair.

Using a HashMap

Description	Example

Initialize a `HashMap<String, Integer>` named `wordCount`, where the keys are `words` (Strings) and the values are the `count of occurrences of each word` (Integers). Define the input text containing a string with multiple repeated words. The `split()` method splits the text string into a `words` array based on spaces. A `for` loop iterates over each word in the `words` array. The `wordCount.getOrDefault(word, 0)` method retrieves the current count of the word if it exists. If the word is not yet in the map, it defaults to 0. The `+1` increments the count for each occurrence and `put(word, newCount)` updates the count in the `HashMap`.	`HashMap<String, Integer> wordCount = new HashMap<>();`
	`String text = "apple banana apple orange banana apple";`
	`String[] words = text.split(" ");`
	
	`for (String word : words) {`
	`  wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);`
	`  }`

Explanation: This Java code snippet demonstrates how to use a HashMap to count the occurrences of words in a given text string. This approach is useful for word frequency analysis in text processing. The `split(" ")` function splits text into words. `HashMap` efficiently tracks word occurrences. `getOrDefault(key, defaultValue)` avoids null values.

Creating a TreeMap

Description	Example
Import the `TreeMap` class from the java.util package to store key-value pairs in sorted order.	`import java.util.TreeMap;`

Initialize a `TreeMap<String, Integer>` named `treeMap` to store fruit names (keys) and their corresponding values (integers). The `TreeMap` automatically sorts the keys in ascending order (Apple → Banana → Cherry). The `treeMap.get("Apple")` call fetches and prints the value associated with "Apple". The `for` loop calls the `keySet()` method to iterate over all keys (which are sorted) and print their associated values. The `containsKey()` method checks if "Cherry" is present and prints a message. The `treemap.remove()` method removes the "Banana" entry from the TreeMap.	`public class TreeMapExample {`
	`  public static void main(String[] args) {`
	`  // Creating a TreeMap`
	`  TreeMap<String, Integer> treeMap = new TreeMap<>();`
	`  // Adding key-value pairs to the TreeMap`
	`  treeMap.put("Banana", 2);`
	`  treeMap.put("Apple", 1);`
	`  treeMap.put("Cherry", 3);`
	`  // Accessing values`
	`  System.out.println("Value for key 'Apple': " + treeMap.get("Apple")); // Output: 1`
	
	`  // Iterating through the TreeMap`
	`  for (String key : treeMap.keySet()) {`
	`  System.out.println(key + ": " + treeMap.get(key));`
	`  }`
	`  // Checking if a key exists`
	`  if (treeMap.containsKey("Cherry")) {`
	`  System.out.println("Cherry exists in the TreeMap.");`
	`  }`
	`  // Removing a key-value pair`
	`  treeMap.remove("Banana");`

Close curly braces to end the `TreeMapExample` class definition.	`  }` `}`

Description	Example

Explanation: This Java program demonstrates the use of a TreeMap, a data structure that stores key-value pairs in sorted order based on keys. `TreeMap` maintains sorted order (ascending by default).

Using a TreeMap

Description	Example

Initialize a `TreeMap<String, Integer>` named `leaderboard` where Keys (String) represent `player names` and Values (Integer) represent `player scores`. `TreeMap` automatically sorts keys in ascending order. Add three players and their scores to the `TreeMap`. Since `TreeMap` maintains sorted order by key (name), the stored order will be: Alice → Bob → Charlie. Display the sorted leaderboard using the `keySet()` method.	`TreeMap<String, Integer> leaderboard = new TreeMap<>();`
	`leaderboard.put("Alice", 150);`
	`leaderboard.put("Bob", 200);`
	`leaderboard.put("Charlie", 100);`
	`// Displaying sorted leaderboard`
	`for (String player : leaderboard.keySet()) {`
	`System.out.println(player + ": " + leaderboard.get(player));`
	`}`

Explanation: This Java code snippet demonstrates the use of a `TreeMap` to store and display a sorted leaderboard of players and their scores. TreeMap stores entries in key-sorted order (ascending). `put(K key, V value)` adds key-value pairs, `get(K key)` retrieves the value for a given key, and keySet() returns keys in sorted order.

Using Java collections in the real world

Managing books in a library management system

Description	Example

Import the `ArrayList` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `Library` class to represent a collection of books. The `books` variable is a private ArrayList<String>, meaning it stores book titles as strings and it cannot be accessed directly from outside the class. The `Library()` constructor initializes the books list when a `Library` object is created. The `addBook()` method adds a new book to the books list. The `displayBooks()` method prints all books stored in the `books` list using a for-each loop. The `main` method creates a `Library` object named `myLibrary`, adds two books: "The Great Gatsby" and "To Kill a Mockingbird", and calls the displayBooks() method to print the book list.	`import java.util.ArrayList;`
	`public class Library {`
	`private ArrayList<String> books;`
	`public Library() {`
	`books = new ArrayList<>();`
	`}`
	`public void addBook(String book) {`
	`books.add(book);`
	`}`
	`public void displayBooks() {`
	`System.out.println("Books in the Library:");`
	`for (String book : books) {`
	`System.out.println(book);`
	`}`
	`}`
	`public static void main(String[] args) {`
	`Library myLibrary = new Library();`
	`myLibrary.addBook("The Great Gatsby");`
	`myLibrary.addBook("To Kill a Mockingbird");`
	`myLibrary.displayBooks();`

Close curly braces to end the `main` and `Library` class definition.	`}` `}`

Managing customer orders in an e-commerce application

Description	Example

Import the `HashMap` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `OrderManagement` class to manage orders. The `orders` variable is private, meaning it cannot be accessed directly from outside the class. It is encapsulated to ensure data integrity. The Java constructor `OrderManagement()` initializes the orders `HashMap` when an `OrderManagement` object is created. The `addOrder()` method adds a new order using the `.put(orderId, customerName)` method. If the same `orderId` is added again, it overwrites the previous entry. The `displayOrders()` method iterates over the `HashMap` using `keySet()` to get all order IDs, retrieves and prints the corresponding customer names. The `main` method creates an instance of `OrderManagement`, adds two orders: Order #101 for Alice and Order #102 for Bob, and calls the `displayOrders()` method to show all orders.	`import java.util.HashMap;`
	`public class OrderManagement {`
	`private HashMap<Integer, String> orders;`
	`public OrderManagement() {`
	`orders = new HashMap<>();`
	`}`
	`public void addOrder(int orderId, String customerName) {`
	`orders.put(orderId, customerName);`
	`}`
	`public void displayOrders() {`
	`System.out.println("Customer Orders:");`
	`for (int orderId : orders.keySet()) {`
	`System.out.println("Order ID: " + orderId + ", Customer Name: " + orders.get(orderId));`
	`}`
	`}`
	`public static void main(String[] args) {`
	`OrderManagement orderManagement = new OrderManagement();`
	`orderManagement.addOrder(101, "Alice");`
	`orderManagement.addOrder(102, "Bob");`
	`orderManagement.displayOrders();`

Close curly braces to end the `main` and `OrderManagement` class definition.	`}` `}`

Explanation: This Java program implements a basic Order Management system using a HashMap to store and manage customer orders. The program uses `HashMap<Integer, String>`, which stores Keys (Integer) to represent Order IDs and Values (String) to represent Customer Names.

Managing employee information in an employee management system

Description	Example

Import the `HashSet` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `EmployeeManager` class with a private variable named `employee` that stores employee names. Encapsulation ensures the set is only modified through class methods. The constructor `EmployeeManager()` initializes the `employees` set when an `EmployeeManager` object is created. The `addEmployee()` method adds an employee name to the `HashSet`. If the employee already exists, the `HashSet` prevents duplicate entries. The `displayEmployees()` method iterates over the `HashSet` to display all employees. The order is not guaranteed because `HashSet` does not maintain insertion order. The Java `main` method creates an instance of `EmployeeManager` and adds three employees: "John Doe", "Jane Smith", and "John Doe". Because "John Doe" is a duplicate, it is ignored by `HashSet`. Calling `displayEmployees()` shows all employees.	`import java.util.HashSet;`
	`public class EmployeeManager {`
	`private HashSet<String> employees;`
	`public EmployeeManager() {`
	`employees = new HashSet<>();`
	`}`
	`public void addEmployee(String employee) {`
	`employees.add(employee);`
	`}`
	`public void displayEmployees() {`
	`System.out.println("Employees in the Company:");`
	`for (String employee : employees) {`
	`System.out.println(employee);`
	`}`
	`}`
	`public static void main(String[] args) {`
	`EmployeeManager manager = new EmployeeManager();`
	`manager.addEmployee("John Doe");`
	`manager.addEmployee("Jane Smith");`
	`manager.addEmployee("John Doe"); // Duplicate will not be added`
	`manager.displayEmployees();`

Close curly braces to end the `main` and `EmployeeManager` class definition.	`}` `}`

Explanation: This Java program implements a basic Employee Management system using a HashSet to store and manage employee names. It uses `LinkedHashSet` to maintain insertion order and `TreeSet` to store employees in sorted order.

Managing tasks in a task management system

Description	Example

Import the `LinkedList` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `TaskManager` class with a private variable named `tasks` that stores tasks. Encapsulation ensures the set is only modified through class methods. The constructor `TaskManager()` initializes the `tasks` list when a `TaskManager` object is created. The `addTask()` method adds a task to the end of the list using `add()` and preserves the insertion order (LinkedList maintains order). The `completeTask()` method removes the first task using `removeFirst()`, prevents errors by checking `isEmpty()` before removal, and prints the completed task. The `displayTasks()` method iterates over the `LinkedList` and prints all tasks. Tasks remain ordered by insertion. The Java `main` method creates an instance of `TaskManager`, adds two tasks: "Finish report" and "Email client", displays tasks, completes the first task, and displays remaining tasks.	`import java.util.LinkedList;`
	`public class TaskManager {`
	`private LinkedList<String> tasks;`
	`public TaskManager() {`
	`tasks = new LinkedList<>();`
	`}`
	`public void addTask(String task) {`
	`tasks.add(task);`
	`}`
	`public void completeTask() {`
	`if (!tasks.isEmpty()) {`
	`String completedTask = tasks.removeFirst();`
	`System.out.println("Completed Task: " + completedTask);`
	`} else {`
	`System.out.println("No tasks to complete.");`
	`}`
	`}`
	`public void displayTasks() {`
	`System.out.println("Current Tasks:");`
	`for (String task : tasks) {`
	`System.out.println(task);`
	`}`
	`}`
	`public static void main(String[] args) {`
	`TaskManager manager = new TaskManager();`
	`manager.addTask("Finish report");`
	`manager.addTask("Email client");`
	`manager.displayTasks();`
	`manager.completeTask();`
	`manager.displayTasks();`

Close curly braces to end the `main` and `TaskManager` class definition.	`}` `}`

Explanation: This Java program implements a simple Task Manager using a LinkedList to store and manage tasks. It supports fast insertions/removals at both ends for addFirst() and removeFirst().

Managing followers in a social media application

Description	Example

Import the `HashSet` class from the `java.util` package, which is a part of Java's Collection Framework and is used to store a dynamic list. Create the `SocialMedia` class with a `HashMap` where Key (String) represents a user and Value (HashSet<String>) stores a set of followers (ensuring uniqueness). The constructor `SocialMedia()` initializes `userFollowers` as an empty `HashMap`. The `addFollower()` method ensures the user exists in the `HashMap` using the `putIfAbsent(user, new HashSet<>())` method and adds the follower to the user's `HashSet` (no duplicates allowed). The `displayFollowers()` method checks if the user exists, prints all followers of the user, and handles missing users by displaying "No followers found". The Java `main` method creates an instance of `SocialMedia` class, adds followers: "Bob" follows "Alice", "Charlie" follows "Alice", and displays Alice's followers.	`import java.util.HashSet;`
	
	`public class SocialMedia {`
	`  private HashMap<String, HashSet<String>> userFollowers;`
	
	`  public SocialMedia() {`
	`  userFollowers = new HashMap<>();`
	`  }`
	
	`  public void addFollower(String user, String follower) {`
	`  userFollowers.putIfAbsent(user, new HashSet<>());`
	`  userFollowers.get(user).add(follower);`
	`  }`
	
	`  public void displayFollowers(String user) {`
	`  System.out.println("Followers of " + user + ":");`
	`  HashSet<String> followers = userFollowers.get(user);`
	`  if (followers != null) {`
	`  for (String follower : followers) {`
	`  System.out.println(follower);`
	`  }`
	`  } else {`
	`  System.out.println("No followers found.");`
	`  }`
	`  }`
	
	`  public static void main(String[] args) {`
	`  SocialMedia socialMedia = new SocialMedia();`
	`  socialMedia.addFollower("Alice", "Bob");`
	`  socialMedia.addFollower("Alice", "Charlie");`
	`  socialMedia.displayFollowers("Alice");`

Close curly braces to end the `main` and `EmployeeManager` class definition.	`  }` `}`

Explanation: This Java program implements a basic social media follower system using `HashMap` and `HashSet`. `HashSet` ensures no user follows the same person twice. If a user has no followers, it prints "No followers found". HashMap provides average time complexity for lookups. Followers cannot be accessed directly, only through class methods.

Java File Handling / Working with File Input and Output Streams

Using the File class

Description	Example
Import the `File` class, which provides methods for file and directory operations.	`import java.io.File;`

Define a class `FileExample` that contains the Java `main` method. Create a `File` object representing a file named example.txt. This does not create the actual file, just a reference to it. Call the `exists()` method on the `File` object to check whether the file physically exists in the specified location. If the file exists, prints "File exists.", otherwise print "File does not exist.".	`public class FileExample {`
	`  public static void main(String[] args) {`
	`  File myFile = new File("example.txt");`
	
	`  // Check if the file exists`
	`  if (myFile.exists()) {`
	`  System.out.println("File exists.");`
	`  } else {`
	`  System.out.println("File does not exist.");`
	` }

Close curly braces to end the `FileExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates how to check whether a file exists in the filesystem using the `File` class from the java.io package.

Writing to Files

Description	Example

Import the `FileWriter` class for writing character data to a file, the `BufferedWriter` class that wraps `FileWriter` to provide efficient writing operations, and the `IOException` class to handle input/output exceptions.	`import java.io.BufferedWriter;`
	`import java.io.FileWriter;`
	`import java.io.IOException;`

Define a class `WriteToFile` that contains the Java `main` method. Create a `FileWriter` class to write to the file "output.txt". A `BufferedWriter` is wrapped around `FileWriter` for more efficient writing. Write text to the file using the `write()` method. The `newLine()` method inserts a newline character (\n). The `close()` method closes the writer to ensure all data is flushed to the file. A confirmation message is printed to the console. The `catch()` call catches `IOException` if any file operation fails (for example, permission issues, disk space) and prints an error message.	`public class WriteToFile {`
	`  public static void main(String[] args) {`
	`  try {`
	`  FileWriter writer = new FileWriter("output.txt");`
	`  BufferedWriter bufferedWriter = new BufferedWriter(writer);`
	
	`  bufferedWriter.write("Hello, World!");`
	`  bufferedWriter.newLine(); // Adds a new line`
	`  bufferedWriter.write("This is a Java file handling example.");`
	
	`  bufferedWriter.close(); // Always close the writer`
	`  System.out.println("Data written to file successfully.");`
	`  } catch (IOException e) {`
	`  System.out.println("An error occurred: " + e.getMessage());`
	` }

Close curly braces to end the `WriteToFile` class definition.	`  }` `}`

Explanation: This Java program demonstrates how to write text to a file using the `FileWriter` and `BufferedWriter` package. It writes multiple lines to the file, handles exceptions properly, and closes the file to prevent resource leaks.

Reading from Files

Description	Example

Import the `FileReader` class that reads character-based data from a file, the `BufferedReader` class that provides efficient reading capabilities by buffering input, and the `IOException` class to handle errors that may occur during file operations.	`import java.io.BufferedReader;`
	`import java.io.FileReader;`
	`import java.io.IOException;`

Define a class `ReadFromFile` that contains the Java `main` method. Create a `FileReader` class to read the file "output.txt". A `BufferedReader` is wrapped around `FileReader` for more efficient reading. Call `readLine()` reads one line at a time from the file. The loop continues until `readLine()` returns null (indicating the end of the file). Each line is printed to the console. The `bufferedReader.close()` method ensures the file resource is released after reading is complete. The `catch()` call catches `IOException` if any file operation fails (for example, permission issues, disk space) and prints an error message.	`public class ReadFromFile {`
	`  public static void main(String[] args) {`
	`  try {`
	`  FileReader reader = new FileReader("output.txt");`
	`  BufferedReader bufferedReader = new BufferedReader(reader);`
	
	`  String line;`
	`  while ((line = bufferedReader.readLine()) != null) {`
	`  System.out.println(line);`
	`  }`
	
	`  bufferedReader.close();`
	`  } catch (IOException e) {`
	`  System.out.println("An error occurred: " + e.getMessage());`
	` }

Close curly braces to end the `FileExample` class definition.	`  }` `}`

Explanation: This Java program reads a file line by line using `FileReader` and `BufferedReader` and prints its content to the console. It reads and prints lines the file line by line, handles exceptions properly, and closes the file to prevent resource leaks.

Using Java Byte Streams

Reading bytes

Description	Example
Import the `FileInputStream` class for reading raw byte data from a file and the `IOException` class to handle input/output exceptions.	`import java.io.FileInputStream;` `import java.io.IOException;`

Define a class `ReadBytes` that contains the Java `main` method. Declare a `fileInputStream` variable, but don't initialize it. Open "example.txt" for reading. Read one byte at a time until the end of the file is reached. The method `byteData()` converts the byte into a character and prints it. If an I/O error occurs, an error stack trace is printed. The `finally` block ensures the file stream is closed, preventing resource leaks. The method `fileInputStream.close()` closes the file to free system resources.	`public class ReadBytes {`
	`  public static void main(String[] args) {`
	`  FileInputStream fileInputStream = null;`
	`  try {`
	`  // Create a FileInputStream to read from a file`
	`  fileInputStream = new FileInputStream("example.txt");`
	
	`  // Variable to hold the byte data`
	`  int byteData;`
	`  // Read bytes until end of file`
	`  while ((byteData = fileInputStream.read()) != -1) {`

Description	Example

	``// Print the byte data as characters
	System.out.println((char) byteData);
	}
	} catch (IOException e) {
	e.printStackTrace();
	} finally {
	// Close the stream to free resources
	if (fileInputStream != null) {
	try {
	fileInputStream.close();
	} catch (IOException e) {
	e.printStackTrace();
	}
	}
	}```

Close curly braces to end the `FileExample` class definition.	```} }```

Explanation: This Java program reads a file byte by byte using `FileInputStream` and prints its contents to the console.

Writing bytes

Description	Example
Import the `FileOutputStream` class for writing raw byte data to a file and the `IOException` class to handle input/output exceptions.	```import java.io.FileOutputStream; import java.io.IOException;```

Define a class `WriteBytes` that contains the Java `main` method. Declare a `FileOutputStream` variable but don't initialize it. Open a `FileOutputStream` for the file "output.txt". If the file does not exist, create a new one. Define a String ("Hello, World!") to write to the file. Convert the string into a byte array using `.getBytes()`. Write the byte array to the file using `fileOutputStream.write(byteData)`. The `IOException` method catches and prints any exceptions during file writing. The `finally` block ensures that the `FileOutputStream` is properly closed to free system resources and uses a null check before calling `.close()`, preventing a `NullPointerException`. If closing the stream fails, it prints the exception.	```public class WriteBytes {
	public static void main(String[] args) {
	FileOutputStream fileOutputStream = null;
	try {
	// Create a FileOutputStream to write to a file
	fileOutputStream = new FileOutputStream("output.txt");
	
	// Data to write
	String data = "Hello, World!";
	// Convert the string to bytes
	byte[] byteData = data.getBytes();
	
	// Write bytes to the file
	fileOutputStream.write(byteData);
	} catch (IOException e) {
	e.printStackTrace();
	} finally {
	// Close the stream to free resources
	if (fileOutputStream != null) {
	try {
	fileOutputStream.close();
	} catch (IOException e) {
	e.printStackTrace();
	}
	}
	}```

Close curly braces to end the `FileExample` class definition.	```} }```

Explanation: This Java program, writes the string "Hello, World!" to a file named output.txt using a `FileOutputStream`. It uses exception handling to catch possible file operation errors and uses a `finally` block to ensure the file stream is always closed.

Byte streams example

Description	Example
Import the `FileInputStream` class for reading raw byte data from a file, `FileOutputStream` class for writing raw byte data to a file, and the `IOException` class to handle input/output exceptions.	```import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException;```

Declare `FileInputStream inputFile` to reads data from source.txt and `FileOutputStream outputFile` to write data to destination.txt. The try block initializes `inputFile` to read from source.txt, initializes `outputFile` to write to destination.txt, reads bytes from source.txt one byte at a time using `inputFile.read()`, writes each byte to destination.txt using `outputFile.write(byteData)`, continues until reaching the end of the file (`-1`), and prints "File copied successfully!" after completion. The catch block prints the stack trace if an `IOException` occurs (for example, file not found, read/write error). The `finally` bock ensures both `inputFile` and `outputFile` are closed to free system resources. It uses null checks to prevent `NullPointerException`.	```public class FileCopy {
	public static void main(String[] args) {
	FileInputStream inputFile = null;
	FileOutputStream outputFile = null;
	
	try {
	// Create FileInputStream to read from "source.txt"
	inputFile = new FileInputStream("source.txt");
	// Create FileOutputStream to write to "destination.txt"
	outputFile = new FileOutputStream("destination.txt");
	
	int byteData;
	// Read bytes from source and write them to destination
	while ((byteData = inputFile.read()) != -1) {
	outputFile.write(byteData);
	}
	
	System.out.println("File copied successfully!");
	} catch (IOException e) {
	e.printStackTrace();
	} finally {
	// Close both streams
	try {
	if (inputFile != null) inputFile.close();
	if (outputFile != null) outputFile.close();
	} catch (IOException e) {
	e.printStackTrace();
	}
	}```

Close curly braces to end the `FileCopy` class definition.	```} }```

Explanation: This Java program copies the contents of a file named source.txt into another file named destination.txt using `FileInputStream` and `FileOutputStream`. It reads and writes files one byte at a time and uses `finally` to always close file streams. The program catches `IOException` to prevent crashes.

Managing Directories in Java

Creating a directory

Description	Example
Import the `java.io.File` package to represent file and directory paths.	```import java.io.File;```

Define a class `CreateDirectory` that contains the Java `main` method. The `String directoryPath = "Projects/Java"` specifies the directory to be created. This means that the program will try to create a folder named "Java" inside a folder named "Projects". Create a `File` object for the directory by calling the `File(directoryPath)` method. The File object represents the directory but does not create it yet. The `if (!directory.exists())` method ensures the directory is created only if it does not already exist. The method `mkdirs()` ensures all parent directories are also created. If creation is successful, the message "Directory created successfully: Projects/Java" is printed to the console. If creation fails, the message "Failed to create directory" is printed to the console. If the directory already exists, the message "Directory aready exists: Projects/Java" is printed to the console.	```public class CreateDirectory {
	public static void main(String[] args) {
	// Define the directory path
	String directoryPath = "Projects/Java";
	
	// Create a File object
	File directory = new File(directoryPath);
	
	// Create the directory
	if (!directory.exists()) {
	boolean created = directory.mkdirs(); // Use mkdirs() to create nested directories
	if (created) {
	System.out.println("Directory created successfully: " + directoryPath);
	} else {
	System.out.println("Failed to create directory.");
	}
	} else {
	System.out.println("Directory already exists: " + directoryPath);
	}```

Close curly braces to end the `CreateDirectory` class definition.	```} }```

Explanation: This Java program creates a directory (including nested directories) if it does not already exist. It handles success and failure cases gracefully.

Listing directory contents

Description	Example
Import the `java.io.File` package to represent file and directory paths.	```import java.io.File;```

Description	Example

Define a class `ListDirectoryContents` that contains the Java `main` method. The `String directoryPath = "Projects/Java"` specifies the directory whose contents will be listed. Create a `File` object for the directory by calling the `File(directoryPath)` method. The File object represents the directory but does not perform any operations yet. The `directory.list()` method returns an array of filenames that exist in the directory. If the directory does not exist or is empty, list() returns null. The `if (contents != null)` method ensures the directory exists and is not empty before proceeding. If contents is null, it prints: "The directory is empty or does not exist." If the directory contains files/subdirectories, the program prints "Contents of Projects/Java", iterates through the contents array and prints each filename.	`public class ListDirectoryContents {`
	`  public static void main(String[] args) {`
	`  String directoryPath = "Projects/Java";`
	`  File directory = new File(directoryPath);`
	
	`  // List all files and directories in the specified directory`
	`  String[] contents = directory.list();`
	
	`  if (contents != null) {`
	`  System.out.println("Contents of " + directoryPath + ":");`
	`  for (String fileName : contents) {`
	`  System.out.println(fileName);`
	`  }`
	`  } else {`
	`  System.out.println("The directory is empty or does not exist.");`
	`  }`

Close curly braces to end the `ListDirectoryContents` class definition.	`  }` `}`

Explanation: This Java program lists all files and subdirectories inside a directory and handles cases where the directory is empty or does not exist
It uses the `File.list()` method to retrieve directory contents efficiently.

Deleting a directory

Description	Example
Import the `java.io.File` package to represent file and directory paths.	`import java.io.File;`

Define a class `ListDirectoryContents` that contains the Java `main` method. The `String directoryPath = "Projects/Java"` specifies the directory to be deleted. Create a `File` object for the directory by calling the `File(directoryPath)` method. The File object represents the directory but does not perform any operations yet. The `if (directory.exists())` method ensures the directory exists before attempting deletion. The `.delete()` method deletes the directory only if it is empty. If successful, it prints "Directory deleted successfully: Projects/Java". If it fails (for example, because it contains files/subdirectories), it prints "Failed to delete directory. It may not be empty.". If the directory is missing, it prints: "Directory does not exist: Projects/Java".	`public class ListDirectoryContents {`
	`  public static void main(String[] args) {`
	`  String directoryPath = "Projects/Java";`
	`  File directory = new File(directoryPath);`
	
	`  // List all files and directories in the specified directory`
	`  String[] contents = directory.list();`
	
	`  if (contents != null) {`
	`  System.out.println("Contents of " + directoryPath + ":");`
	`  for (String fileName : contents) {`
	`  System.out.println(fileName);`
	`  }`
	`  } else {`
	`  System.out.println("The directory is empty or does not exist.");`
	`  }`

Close curly braces to end the `ListDirectoryContents` class definition.	`  }` `}`

Explanation: This Java program uses the `File.delete()` method to delete a specified directory if it exists. The program handles success and failure cases gracefully.

Creating a directory with NIO

Description	Example

Import Java class `java.nio.file.Files` for file and directory operations. `java.nio.file.Path` to represent file and directory paths in a platform-independent way, `java.nio.file.Paths` to create Path instances, and `java.io.IOException` to handle potentila I/O errors.	`import java.nio.file.Files;`
	`import java.nio.file.Path;`
	`import java.nio.file.Paths;`
	`import java.io.IOException;`

Define a class `CreateDirectory` that contains the Java `main` method. The method `Paths.get("Projects/NiceExample")` creates a Path object representing the directory to be created. The `try` block uses `Files.createDirectories()` instead of `File.mkdirs()` to creates all necessary parent directories if they don't exist. It does not throw an error if the directory already exists and stores the created directory path in `createdDir`. The program prints "Directory created successfully: Projects/NiceExample" if it is successful. The `catch` block catches `IOException` if directory creation fails (for example, insufficient permissions) and prints an error message: "Failed to create directory: <error_messages>".	`public class CreateDirectory {`
	`  public static void main(String[] args) {`
	`  // Define the directory path`
	`  String directoryPath = "Projects/Java";`
	
	`  // Create a File object`
	`  File directory = new File(directoryPath);`
	
	`  // Create the directory`
	`  if (!directory.exists()) {`
	`  boolean created = directory.mkdirs(); // Use mkdirs() to create nested directories`
	`  if (created) {`
	`  System.out.println("Directory created successfully: " + directoryPath);`
	`  } else {`
	`  System.out.println("Failed to create directory.");`
	`  }`
	`  } else {`
	`  System.out.println("Directory already exists: " + directoryPath);`
	`  }`

Close curly braces to end the `CreateDirectory` class definition.	`  }` `}`

Explanation: This Java program creates a directory using Java NIO (New Input/Output) instead of the traditional File class. It handles success and failure cases gracefully and works cross-platform.

Real World example of Document Management System

Description	Example

Import Java class `java.nio.file.Files` for file and directory operations. `java.nio.file.Path` to represent file and directory paths in a platform-independent way, `java.nio.file.Paths` to create Path instances, `java.io.IOException` to handle potentila I/O errors, and `java.util.Scanner` for handling user input.	`import java.nio.file.Files;`
	`import java.nio.file.Path;`
	`import java.nio.file.Paths;`
	`import java.io.IOException;`
	`import java.util.Scanner;`

Define a class `DocumentManagementSystem` that contains the Java `main` method. All directory operations will occur within the "Documents" folder defined by the String `BASE_DIRECTORY`. The `main` method continuously prompts the user to choose an option and calls the corresponding method based on user input: "1" creates a new directory inside "Documents", "2" lists contents of a specified directory, "3" deletes a specified directory, and "4" exits the program.	`public class DocumentManagementSystem {`
	`  private static final String BASE_DIRECTORY = "Documents";`
	
	`  public static void main(String[] args) {`
	`  Scanner scanner = new Scanner(System.in);`
	`  String command;`
	
	`  while (true) {`
	`  System.out.println("1. Create directory\n2. List documents\n3. Delete directory\n4. Exit");`
	`  command = scanner.nextLine();`
	
	`  switch (command) {`
	`  case "1": createDirectory(scanner); break;`
	`  case "2": listDirectory(scanner); break;`
	`  case "3": deleteDirectory(scanner); break;`
	`  case "4": System.exit(0); return;`
	`  default: System.out.println("Invalid choice.");`
	`  }`
	`  }`
	`  }`

The `createDirectory()` method creates a new directory and reads directory name from user input. It uses `Files.createDirectories(path)` to create the directory (including missing parent directories). If successful, it prints "Created: path". If an error occurs, it prints "Error: message".	`  private static void createDirectory(Scanner scanner) {`
	`  System.out.print("New directory name: ");`
	`  Path path = Paths.get(BASE_DIRECTORY, scanner.nextLine());`
	`  try {`
	`  Files.createDirectories(path);`
	`  System.out.println("Created: " + Files.createDirectories(path));`
	`  } catch (IOException e) {`
	`  System.err.println("Error: " + e.getMessage());`
	`  }`
	`  }`

The `listDirectory()` method lists the contents of a directory and reads directory name from user input. It uses `Files.list(path)` to retrieve the directory and prints each file/subdirectory. If the directory doesn't exist or an error occurs, it prints "Error: message".	`  private static void listDirectory(Scanner scanner) {`
	`  System.out.print("Directory to list: ");`
	`  Path path = Paths.get(BASE_DIRECTORY, scanner.nextLine());`
	`  try {`
	
	`  Files.list(path).forEach(System.out::println);`
	`  } catch (IOException e) {`
	`  System.err.println("Error: " + e.getMessage());`
	`  }`
	`  }`

The `deleteDirectory()` method deletes a directory and reads directory name from user input. It uses `Files.delete(path)` to delete the specified directory. If successful, it prints "Deleted: path". The `Files.delete(path)` will fail if the directory is not empty. It only works on empty directories.	`  private static void deleteDirectory(Scanner scanner) {`
	`  System.out.print("Directory to delete: ");`
	`  Path path = Paths.get(BASE_DIRECTORY, scanner.nextLine());`
	`  try {`
	`  Files.delete(path);`
	`  System.out.println("Deleted: " + path);`
	`  } catch (IOException e) {`

Description	Example
	System.err.println("Error: " + e.getMessage()); }
Close curly braces to end the `main` class definition.	} }

Explanation: This Java program provides a simple command-line interface for managing directories inside a "Documents" folder. It allows users to create, list, and delete directories using Java NIO (New Input/Output).

Using Java Date and Time Classes

Using the LocalDate class

Description	Example
Import the `LocalDate` class, which is part of the Java Date and Time API.	`import java.time.LocalDate;`

Define a public class `LocalDateExample` that contains the Java `main` method. Use `LocalDate.now()` to retrieve the current date and print it in the "YYYY-MM-DD" format, which is the default format of `LocalDate.toString()`.	`public class LocalDateExample {`
	`public static void main(String[] args) {`
	`LocalDate today = LocalDate.now();`
	`System.out.println("Today's date: " + today);`

Close curly braces to end the `LocalDateExample` class definition.	} }

Explanation: This Java program demonstrates the use of the `LocalDate` class from the `java.time` package to get and display the current date.

Using the LocalTime class

Description	Example
Import the `LocalTime` class, which is part of the Java Date and Time API.	`import java.time.LocalTime;`

Define a public class `LocalTimeExample` that contains the Java `main` method. Use `LocalTime.now()` to retrieve the current system time and print it in the *"HH:mm:ss.SSS"* (hours, minutes, seconds, and milliseconds/nanoseconds) format, which is the default format of `LocalTime.toString()`.	`public class LocalTimeExample {`
	`public static void main(String[] args) {`
	`LocalTime currentTime = LocalTime.now();`
	`System.out.println("Current time: " + currentTime);`

Close curly braces to end the `LocalTimeExample` class definition.	} }

Explanation: This Java program demonstrates the use of the `LocalTime` class from the `java.time` package to get and display the current time.

Using the LocalDateTime class

Description	Example
Import the `LocalDateTime` class, which is part of the Java Date and Time API.	`import java.time.LocalDateTime;`

Define a public class `LocalDateTimeExample` that contains the Java `main` method. Use `LocalDateTime.now()` to retrieve the current system date and time. Print the current date and time in the default format "YYYY-MM-DDTHH:MM:SS.SSS" (year, month, day, hours, minutes, seconds, and milliseconds/nanoseconds), which is the default format of `LocalDateTime.toString()`.	`public class LocalDateTimeExample {`
	`public static void main(String[] args) {`
	`LocalDateTime now = LocalDateTime.now();`
	`System.out.println("Current date and time: " + now);`

Close curly braces to end the `LocalDateTimeExample` class definition.	} }

Explanation: This Java program demonstrates the use of the `LocalDateTime` class from the `java.time` package to get and display the current date and time. `LocalDateTime` is an immutable class that represents both date and time without a time zone.

Using the ZonedDateTime class

Description	Example
Import the `ZonedDateTime` class, which is part of the Java Date and Time API.	`import java.time.ZonedDateTime;`

Define a public class `ZonedDateTimeExample` that contains the Java `main` method. Use `ZonedDateTime.now()` to retrieve the current system date and time, including the time zone. Print the current date, time, and zone in the default ISO-8601 format.	`public class ZonedDateTimeExample {`
	`public static void main(String[] args) {`
	`ZonedDateTime zonedNow = ZonedDateTime.now();`
	`System.out.println("Current date and time with zone: " + zonedNow);`

Close curly braces to end the `ZonedDateTimeExample` class definition.	} }

Explanation: This Java program demonstrates how to use the `ZonedDateTime` class from the `java.time` package to retrieve and display the current date and time along with the time zone. It is useful when working with time zones in applications such as scheduling, logging, and internationalization.

Real World example of an Event Management System

Description	Example

Import the `LocalDate`, `LocalTime`, `LocalDateTime`, `ZoneId`, `ZonedDateTime`, and `Scanner` classes that are part of the Java Date and Time API.	`import java.time.LocalDate;`
	`import java.time.LocalDateTime;`
	`import java.time.LocalTime;`
	`import java.time.ZoneId;`
	`import java.time.ZonedDateTime;`
	`import java.util.Scanner;`

Define an `EventManagement` class to represent an event with `name`, `date`, `time`, and `timeZone`. The method `getEventDateTime()` converts `LocalDate` and `LocalTime` into `LocalDateTime`. Then converts `LocalDateTime` into `ZonedDateTime` using the specified time zone.	`public class EventManagement {`
	`static class Event {`
	`String name;`
	`LocalDate date;`
	`LocalTime time;`
	`ZoneId timeZone;`
	`public Event(String name, LocalDate date, LocalTime time, ZoneId timeZone) {`
	`this.name = name;`
	`this.date = date;`
	`this.time = time;`
	`this.timeZone = timeZone;`
	`}`
	`public ZonedDateTime getEventDateTime() {`
	`LocalDateTime localDateTime = LocalDateTime.of(date, time);`
	`return ZonedDateTime.of(localDateTime, timeZone);`

Description	Example
	} }

Define a public class with the Java `main` method and use it to accept user input for event details. This class captures `name`, `date`, `time`, and `timeZone` from user input. The method `Event(name, date, time, timeZone)` creates an event object through user input. The method `getEventDateTime()` displays the event date an time in the specified time zone. The method `ZonedDateTime` converts `eventDateTime` to the system's local time zone. The method `scanner.close()` closes the scanner to free up resources.	`public static void main(String[] args) {`
	`  Scanner scanner = new Scanner(System.in);`
	
	`  // Input event details`
	`  System.out.println("Enter event name:");`
	`  String name = scanner.nextLine();`
	
	`  System.out.println("Enter event date (YYYY-MM-DD):");`
	`  String dateInput = scanner.nextLine();`
	`  LocalDate date = LocalDate.parse(dateInput);`
	
	`  System.out.println("Enter event time (HH:MM):");`
	`  String timeInput = scanner.nextLine();`
	`  LocalTime time = LocalTime.parse(timeInput);`
	
	`  System.out.println("Enter time zone (e.g., America/New_York):");`
	`  String zoneInput = scanner.nextLine();`
	`  ZoneId timeZone = ZoneId.of(zoneInput);`
	
	`  // Create the event`
	`  Event event = new Event(name, date, time, timeZone);`
	
	`  // Display event details`
	`  System.out.println("Event created: " + event.name);`
	`  ZonedDateTime eventDateTime = event.getEventDateTime();`
	`  System.out.println("Event Date and Time: " + eventDateTime);`
	
	`  // Display in system's default time zone`
	`  ZonedDateTime defaultZonedDateTime = eventDateTime.withZoneSameInstant(ZoneId.systemDefault());`
	`  System.out.println("Event Date and Time in your local time zone: " + defaultZonedDateTime);`
	
	`  scanner.close();`

Close curly braces to end the `EventManagement` class definition.	} }

Explanation: This Java program is a simple event management system that allows users to enter an event's details, including its name, date, time, and time zone. It then converts and displays the event time in both the specified time zone and the system's default time zone.

Formatting Dates in Java

Formatting a date using LocalDate

Description	Example
Import the `LocalDate` class to represent a date (year, month, day) without time or a time zone and `DateTimeFormatter` class to define a custom format for displaying dates.	`import java.time.LocalDate;` `import java.time.format.DateTimeFormatter;`

Define a public class `DateFormattingExample` that contains the Java `main` method. Use `LocalDate.now()` to retrieve the current date in the "YYYY-MM-DD" format, which is the default format of `LocalDate()`. Define a date format using `DateTimeFormatter.ofPattern("dd/MM/yyyy")`. Format the date using `currentDate.format(formatter)` to convert the current date into the specified format and print the formatted date to the console.	`public class DateFormattingExample {`
	`  public static void main(String[] args) {`
	`  // Get the current date`
	`  LocalDate currentDate = LocalDate.now();`
	
	`  // Define the format`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");`
	
	`  // Format the date`
	`  String formattedDate = currentDate.format(formatter);`
	
	`  // Print the formatted date`
	`  System.out.println("Formatted Date: " + formattedDate);`

Close curly braces to end the `DateFormattingExample` class definition.	} }

Explanation: This Java program demonstrates how to format a date using DateTimeFormatter from the java.time package. It formats dates into a human-friendly format.

Real World example of formatting birthdates in a User Registration System

Description	Example

Import the `LocalDate` class to represent a date (year, month, day) without time or a time zone, the `DateTimeFormatter` class to define a custom format for displaying dates, and the `Scanner` class to get user input.	`import java.time.LocalDate;`
	`import java.time.format.DateTimeFormatter;`
	`import java.util.Scanner;`

Define a public class `UserRegistration` that contains the Java `main` method. Create a `Scanner` object to read user input. Get the user name and store it in the `name` variable. Get the user birthdate in the "YYYY-MM-DD" format. The input string `birthdateInput` is converted into a `LocalDate` object using `LocalDate.parse()`. Format the birthdate using the "EEEE, MMM dd, yyyy" pattern, where EEEE is the full weekday name, such as "Monday"; MMM is the abbreviated month name, such as Mar, dd is the two-digit day, such as 11, and "yyyy" is the four-digit year, such as 2025. Use the `birthdate.format(formatter)` method to convert the date into a readable format. Print a personalized message with the formatted birthdate and close the scanner.	`public class UserRegistration {`
	`  public static void main(String[] args) {`
	`  Scanner scanner = new Scanner(System.in);`
	
	`  // Get user's name`
	`  System.out.print("Enter your name: ");`
	`  String name = scanner.nextLine();`
	
	`  // Get user's birthdate`
	`  System.out.print("Enter your birthdate (yyyy-MM-dd): ");`
	`  String birthdateInput = scanner.nextLine();`
	
	`  // Parse the input string into a LocalDate object`
	`  LocalDate birthdate = LocalDate.parse(birthdateInput);`
	
	`  // Define the desired output format`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("EEEE, MMM dd, yyyy");`
	
	`  // Format the birthdate using the defined formatter`
	`  String formattedBirthdate = birthdate.format(formatter);`
	
	`  // Display the result`
	`  System.out.println("Hello " + name + "! Your birthdate is: " + formattedBirthdate);`
	
	`  // Close the scanner`
	`  scanner.close();`

Close curly braces to end the `UserRegistration` class definition.	} }

Explanation: This Java program prompts the user to enter their name and birthdate, then formats and displays the birthdate in a more readable format.

Using Timezones in Java

Creating a ZoneId

Description	Example
Import `ZoneId` which is part of the Java Date and Time API class to represent a time zone, such as "America/New_York", "Asia/Tokyo", and "Europe/London".	`import java.time.ZoneId;`

Define a public class `TimeZoneExample` that contains the Java `main` method. Use `ZoneId.of("America/New_York")` to create a `ZoneId` object for New York and display the Time Zone ID to the console.	`public class TimeZoneExample {`
	`  public static void main(String[] args) {`
	`  // Create a ZoneId for New York`
	`  ZoneId newYorkZone = ZoneId.of("America/New_York");`
	`  System.out.println("Time Zone ID: " + newYorkZone);`

Close curly braces to end the `TimeZoneExample` class definition.	} }

Explanation: This Java program demonstrates how to create and display a time zone ID using the `java.time` package.

Creating a ZoneDateTime

Description	Example
Import the `ZonedDateTime` and `ZoneId` classes which are part of the Java Date and Time API class to represent a date-time with a time zone.	`import java.time.ZonedDateTime;` `import java.time.ZoneId;`

Description	Example

Create a time zone object for New York by calling `ZoneId.of("America/New_York")` and retrieve the current date and time in that time zone. Display the current date and time in New York.	`public class ZonedDateTimeExample {`
	`  public static void main(String[] args) {`
	`  // Getting the current date and time in New York`
	`  ZonedDateTime newYorkTime = ZonedDateTime.now(ZoneId.of("America/New_York"));`
	`  System.out.println("Current Date and Time in New York: " + newYorkTime);`

Close curly braces to end the `ZonedDateTimeExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates how to create and display a time zone ID using the `java.time` package.

Real World example of Scheduling Meeting across Time Zones

Description	Example

Import the `ZonedDateTime`, `ZoneId`, and `DateTimeFormatter` classes which are part of the Java Date and Time API class to represent a date-time with a time zone and format the date-time in a custom pattern.	`import java.time.ZonedDateTime;`
	`import java.time.ZoneId;`
	`import java.time.format.DateTimeFormatter;`

Define the meeting time in UTC. `ZonedDateTime.parse("2024-12-30T15:00:00Z")` parses the fixed UTC time (2024-12-30 15:00:00 UTC) into a `ZonedDateTime` object. Create an array of time zones for participants in New York, London, Kolkata, and Sydney. These time zones are later used to convert the UTC time to each participant's local time. Create a custom formatter for displaying the date and time in the pattern: `DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss z")`. Print the meeting time in UTC using the defined format. For each time zone, use `meetingTimeUTC.withZoneSameInstant(ZoneId.ofTimeZone())` to convert the meeting time from UTC to the local time of that participant's time zone and print the meeting time in the participant's local time zone using the custom formatter.	`public class ConferenceScheduler {`
	`  public static void main(String[] args) {`
	`  // Define the meeting time in UTC`
	`  ZonedDateTime meetingTimeUTC = ZonedDateTime.parse("2024-12-30T15:00:00Z");`
	
	`  // Define participant time zones`
	`  String[] participantTimeZones = {`
	`  "America/New_York", // Eastern Standard Time (EST)`
	`  "Europe/London",  // Greenwich Mean Time (GMT)`
	`  "Asia/Kolkata",  // Indian Standard Time (IST)`
	`  "Australia/Sydney"  // Australian Eastern Daylight Time (AEDT)`
	`  };`
	
	`  // Format for displaying the date and time`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss z");`
	
	`  // Print the meeting time in each participant's local time zone`
	`  System.out.println("Meeting Time in UTC: " + meetingTimeUTC.format(formatter));`
	`  for (String timeZone : participantTimeZones) {`
	`  ZonedDateTime localTime = meetingTimeUTC.withZoneSameInstant(ZoneId.of(timeZone));`
	`  System.out.println("Meeting Time in " + timeZone + ": " + localTime.format(formatter));`
	`  }`

Close curly braces to end the `ConferenceScheduler` class definition.	`  }` `}`

Explanation: This Java program simulates scheduling a meeting across different time zones. It converts a fixed UTC meeting time to the local times of participants in various time zones and displays it in a formatted way.

Parsing Dates from Strings in Java

Parsing dates with DateTimeFormatter

Description	Example
Import the `LocalDate` and `DateTimeFormatter` classes, which are part of the Java Date and Time API class and used to represent dates without a time zone and define a pattern for parsing and formatting dates.	`import java.time.LocalDate;` `import java.time.format.DateTimeFormatter;`

Create a public class `DateParsingExample` that contains the Java `main` method and define a string variable `dateString` to represent date in the format "yyyy-MM-dd". Create a date formatter using the `DateTimeFormatter.ofPattern("yyyy-MM-dd")` method. Use `LocalDate.parse(dateString, formatter)` to convert the `dateString` into a `LocalDate` object and print the parsed date.	`public class DateParsingExample {`
	`  public static void main(String[] args) {`
	`  // Define a date string to parse`
	`  String dateString = "2025-01-27";`
	
	`  // Create a DateTimeFormatter to define the expected format`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");`
	
	`  // Parse the string into a LocalDate object`
	`  LocalDate date = LocalDate.parse(dateString, formatter);`
	
	`  // Output the parsed date`
	`  System.out.println("Parsed date: " + date);`

Close curly braces to end the `DateParsingExample` class definition.	`  }` `}`

Explanation: This Java program demonstrates how to parse a date string into a `LocalDate` object using the `DateTimeFormatter` class.

Using custom date formats

Description	Example
Import the `LocalDate` and `DateTimeFormatter` classes, which are part of the Java Date and Time API class and used to represent dates without a time zone and define a pattern for parsing and formatting dates.	`import java.time.LocalDate;` `import java.time.format.DateTimeFormatter;`

Create a public class `CustomDateParsing` that contains the Java `main` method and define a string variable `dateString` to represent date in the format "dd/MM/yyyy". Create a date formatter using the `DateTimeFormatter.ofPattern("dd/MM/yyyy")` method. Use `LocalDate.parse(dateString, formatter)` to convert the `dateString` into a `LocalDate` object and print the parsed date.	`public class CustomDateParsing {`
	`  public static void main(String[] args) {`
	`  String dateString = "15/01/2025";`
	
	`  // Define the pattern for parsing`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");`
	
	`  LocalDate date = LocalDate.parse(dateString, formatter);`
	
	`  System.out.println("Parsed date: " + date);`

Close curly braces to end the `CustomDateParsing` class definition.	`  }` `}`

Explanation: This Java program demonstrates how to parse a date string with a custom format into a `LocalDate` object using the `DateTimeFormatter` class.

Parsing LocalDateTime

Description	Example
Import the `LocalDateTime` and `DateTimeFormatter` classes, which are part of the Java Date and Time API class and used to represent dates without a time zone and define a pattern for parsing and formatting dates.	`import java.time.LocalDateTime;` `import java.time.format.DateTimeFormatter;`

Create a public class `DateTimeParsingExample` that contains the Java `main` method and define a string variable `dateString` to represent date in the "yyyy-MM-dd" format. Create a date formatter using the `DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")` method. Use `LocalDateTime.parse(dateTimeString, formatter)` to convert the `dateTimeString` into a `LocalDateTime` object using the formatter and print the parsed date.	`public class DateTimeParsingExample {`
	`  public static void main(String[] args) {`
	`  String dateTimeString = "2025-01-23 15:30";`
	
	`  // Define the pattern for date and time`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");`
	
	`  LocalDateTime dateTime = LocalDateTime.parse(dateTimeString, formatter);`
	
	`  System.out.println("Parsed date and time: " + dateTime);`

Close curly braces to end the `DateTimeParsingExample` class definition.	`  }` `}`

Description	Example

Explanation: This Java program demonstrates how to parse a date string with a custom format into a `LocalDateTime` object using the `DateTimeFormatter` class.

Example of extracting date from a simple sentence

Description	Example

Import the `LocalDate`, `DateTimeFormatter`, and `DateTimeParseException` classes, which are part of the Java Date and Time API class and used to represent dates without a time zone, define a pattern for parsing and formatting dates, and handle errors if the date format is incorrect.	`import java.time.LocalDate;`
	`import java.time.format.DateTimeFormatter;`
	`import java.time.format.DateTimeParseException;`

Create a public class `ExtractDateFromSentence` that contains the Java main method and define a sentence containing a date formatted as 'yyyy-MM-dd'. Extract the date substring using `sentence.substring(sentence.indexOf("on") + 3, sentence.indexOf("."))`. The `sentence.indexOf("on") + 3` method finds the position of "on" and moves three characters forward to skip "on " (with the space), `sentence.indexOf(".")` identifies the position of the period (".") at the end of the date, and `substring(...)` extracts the portion of the string that contains the date. Parse the extracted date using `LocalDate.parse(dateString, formatter)` and convert the extracted string into a `LocalDate` object. The `try-catch` block prints the extracted date if successful. If parsing fails due to an incorrect format, the block catches `DateTimeParseException` and displays an error message.	`public class ExtractDateFromSentence {`
	`  public static void main(String[] args) {`
	`  String sentence = "The event will take place on 2025-01-23.";`
	
	`  // Define the date pattern`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");`
	
	`  // Extract the date part from the string`
	`  String dateString = sentence.substring(sentence.indexOf("on") + 3, sentence.indexOf("."));`
	
	`  try {`
	`  LocalDate date = LocalDate.parse(dateString, formatter);`
	`  System.out.println("Extracted date: " + date);`
	`  } catch (DateTimeParseException e) {`
	`  System.out.println("Error parsing date: " + e.getMessage());`
	`  }`

Close curly braces to end the `ExtractDateFromSentence` class definition.	`  }` `}`

Explanation: This Java program extracts a date from a given sentence, parses it into a `LocalDate` object, and displays it in a structured format. It also gracefully handles potential parsing errors.

Example of extracting multiple dates from a text string

Description	Example

Import the `LocalDate`, `DateTimeFormatter`, and `DateTimeParseException` classes, which are part of the Java Date and Time API class and used to represent dates without a time zone, define a pattern for parsing and formatting dates, and handle errors if the date format is incorrect.	`import java.time.LocalDate;`
	`import java.time.format.DateTimeFormatter;`
	`import java.time.format.DateTimeParseException;`

Create a public class `ExtractMultipleDates` that contains the Java main method and define a text string containing three dates in the "yyyy-MM-dd" format. These dates are separated by commas and the word "and". Define the date format using `DateTimeFormatter.ofPattern("yyyy-MM-dd")`. Use regular expressions (`", | and "`) to split the string by comma followed by a space (", ") and the word "and" followed by a space ("and "). This extracts the date strings from the text. Iterate over the extracted parts and parse dates. For each extracted part, `trim()` removes any leading or trailing spaces and `LocalDate.parse(part.trim(), formatter)` converts the string into a `LocalDate` object. If parsing is successful, it prints the extracted date. If parsing fails, the catch block handles the error and prints an error message.	`public class ExtractMultipleDates {`
	`  public static void main(String[] args) {`
	`  String text = "Important dates: 2025-01-23, 2025-02-14, and 2025-03-01.";`
	
	`  // Define the date pattern`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");`
	
	`  // Split the string to find dates`
	`  String[] parts = text.split(", | and ");`
	
	`  for (String part : parts) {`
	`  try {`
	`  LocalDate date = LocalDate.parse(part.trim(), formatter);`
	`  System.out.println("Extracted date: " + date);`
	`  } catch (DateTimeParseException e) {`
	`  System.out.println("Error parsing date: " + part.trim());`
	`  }`
	`  }`

Close curly braces to end the `ExtractMultipleDates` class definition.	`  }` `}`

Explanation: This Java program extracts multiple dates from a given text, parses them into `LocalDate` objects, and prints them in a structured format. It also handles potential errors if any part of the text is not in the expected date format.

Example of extracting dates from mixed content

Description	Example

Import the `LocalDate`, `DateTimeFormatter`, and `DateTimeParseException` classes, which are part of the Java Date and Time API class and used to represent dates without a time zone, define a pattern for parsing and formatting dates, and handle errors if the date format is incorrect.	`import java.time.LocalDate;`
	`import java.time.format.DateTimeFormatter;`
	`import java.time.format.DateTimeParseException;`

Create a public class `ExtractDatesFromMixedContent` that contains the Java main method and define a string named `mixedContent` containing a mixture of text and two dates (2025-01-23 and 2025-02-28). The dates are in the "yyyy-MM-dd" format. These dates are separated by commas and the word "and". Define the date format using `DateTimeFormatter.ofPattern("yyyy-MM-dd")`. Splits the input string by spaces into individual words. The resulting `words[]` array contains both text and possible date strings. Iterate over each word using the regex `word.matches("\\d{4}-\\d{2}-\\d{2}")` and check if it matches the date pattern (yyyy-MM-dd). If a word matches the pattern, attempt to parse it into a LocalDate using the previously defined formatter. If parsing is successful, prints the extracted date. If there is a parsing error (invalid date), the `try-catch` block handles it and prints an error message.	`public class ExtractDatesFromMixedContent {`
	`  public static void main(String[] args) {`
	`  String mixedContent = "Please note that our deadlines are on 2025-01-23 and 2025-02-28.";`
	
	`  // Define the date pattern`
	`  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");`
	
	`  // Split based on spaces and check each part`
	`  String[] words = mixedContent.split(" ");`
	
	`  for (String word : words) {`
	`  if (word.matches("\\d{4}-\\d{2}-\\d{2}")) { // Check if it matches a date pattern`
	`  try {`
	`  LocalDate date = LocalDate.parse(word, formatter);`
	`  System.out.println("Extracted date: " + date);`
	`  } catch (DateTimeParseException e) {`
	`  System.out.println("Error parsing date: " + word);`
	`  }`
	`  }`
	`  }`

Close curly braces to end the `ExtractDatesFromMixedContent` class definition.	`  }` `}`

Explanation: This Java program extracts dates from a string containing mixed content (text and dates), parses them into `LocalDate` objects, and prints the valid dates. If any date format is invalid, the program gracefully handles the error.

Author(s)

Ramanujam Sriavvasn
Lavanya Thirevali Sundararajan