Inventive Media

Java Programming Essentials - Day 3

Jansen Ang

What are we learning today?

- Packages
- Class Inheritance
- Object Polymorphism
- Object Equality
- Interfaces
- Comparable Interface
- Exceptions
- Processing Files
- Generics
- Lambdas
- Streams
- Multidimensional data

Day 3

- Recap

- Object equality
- Class inheritance
- Interfaces
- Object polymorphism
- Streams

- Comparable Interface
- Packages
- Exceptions
- Processing files
- Type parameters

- Multidimensional data

Packages

What are Packages?

- Packages in Java are used to group related classes and interfaces together.
- They provide a unique namespace for the types they contain, helping to prevent naming conflicts.
- Packages can be seen as a way to organize your files into different directories.

Why are Packages Used?

- **Organization**: Packages help in organizing code files logically, making the code easier to manage and understand.
- **Avoiding Naming Conflicts**: By providing a unique namespace, they prevent conflicts that would occur if two classes in different parts of a program were given the same name.
- **Access Protection**: Packages can be used to control access; classes or members can be package-private, accessible only within their own package.
- **Reusability and Modularity:** Packages promote reusability and modularity of code. You can import and use classes from other packages, including the standard Java libraries.

Packages Example

```java
// animals/Dog.java
package animals;

public class Dog {
    public void bark() {
        System.out.println("Woof!");
    }
}


// main/Main.java
package main;

import animals.Dog; // Import the Dog class from the animals package

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.bark();
    }
}
```

Using the Packages:

- When these files are compiled and run, the **Main** class in the **main** package creates an instance of the **Dog** class from the **animals** package and calls its **bark** method.
- This demonstrates how packages are used to organize and use classes from different namespaces.
- Package names are typically lowercase to avoid conflict with class names.

Class Inheritance

Inheritance

Inheritance in Java is a mechanism where one class acquires the properties (fields) and behaviors (methods) of another class. With inheritance, we can create a new class based on an existing class.

Why Use Class Inheritance?

1. **Code Reusability**: Inheritance allows us to reuse methods and fields of the parent class, reducing code duplication.
2. **Method Overriding**: Allows a child class to provide a specific implementation of a method that is already defined in its parent class.
3. **Polymorphism**: Through inheritance, we can perform a single action in different ways, depending on the object's class.

Inheritance Example

```java
// Parent class
class Vehicle {
    public void honk() {
        System.out.println("Vehicle is honking");
    }
}

// Child class
class Car extends Vehicle {
    public void display() {
        System.out.println("This is a Car");
    }
}
```

Use the **extends** keyword to inherit from a parent/superclass.

```java
// Using the classes
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.honk(); // Inherited method
        myCar.display(); // Own method
    }
}
```

What are inherited from a Parent class?

- A subclass inherits all **public** and **protected** members of the superclass.
  - Members are fields (like variables) and methods (like functions) of a class.

- **Access to Private Members**: The subclass cannot directly access private members of the superclass. These are internal to the superclass and hidden from the subclass.

- **Using super Keyword:** The subclass can use the super keyword to call methods or constructors of the superclass.

- **Constructors Are Not Inherited**: Constructors are special methods used to initialize new objects, and they are not inherited. However, a subclass can call a constructor of the superclass using super.

Object Polymorphism

What is Polymorphism?

Polymorphism, in the context of object-oriented programming, is the ability of objects of different classes to respond to the same message (method call) in different ways.

Types: There are two main types of polymorphism:

- **Compile-Time Polymorphism (Method Overloading)**: Different methods in the same class have the same name but different parameters.
- **Run-Time Polymorphism (Method Overriding):** A subclass provides a specific implementation of a method that is already defined in its superclass.

Why is Polymorphism Used?

- **Flexibility and Reusability**: It allows for writing flexible and reusable code. Methods written for a superclass type can work with any subclass type.
- **Simplifies Code**: It helps to simplify code because one interface can be used to specify a general set of actions and specific implementations are decided at runtime.
- **Extensibility**: It makes it easier to extend code with new features without altering existing code.

Run-Time Polymorphism (Method Overriding)

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

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

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}
```

```java
public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Dog object
        myAnimal.sound(); // Outputs: Dog barks

        myAnimal = new Cat(); // Cat object
        myAnimal.sound(); // Outputs: Cat meows
    }
}
```

Object Equality

Object Equality

Object equality in Java is a fundamental concept that pertains to how objects are compared for equality.

Why Object Equality is Important

1. **Identifying** Unique Objects: In many applications, it's important to determine whether two objects represent the same data or state.
2. **Collections**: Java collections like Set rely on object equality to avoid duplicates.
3. **Logical Operations**: Object equality is used in logical operations where decisions are based on whether objects are considered equal.

Ways to Compare Objects for Equality

Using == Operator: This compares object references, not the actual content of the objects. Two object references are equal if they point to the same memory location.

```
String a = new String("Hello");
String b = new String("Hello");
System.out.println(a == b); // false, because a and b refer to different objects
```

Using equals() Method: This method is used to compare the content of the objects. The Object class provides a basic implementation of equals(), which you can override in your class to define what it means for two instances of your class to be considered equal.

```
String a = new String("Hello");
String b = new String("Hello");
System.out.println(a.equals(b)); // true, because a and b have the same content
```

Overriding **equals** Method

When overriding equals(), you should also override hashCode() to maintain the general contract for the hashCode() method, which states that equal objects must have equal hash codes.

```java
class Person {
    String name;
    int age;

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age && name.equals(person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

```java
Person person1 = new Person("Alice", 30);
Person person2 = new Person("Alice", 30);
System.out.println(person1.equals(person2)); // true
```

Two Person objects are considered equal if they have the same name and age. The equals() method is overridden to implement this logic, and hashCode() is also overridden to ensure that equal objects have the same hash code, which is a requirement especially when using hash-based collections like HashSet or HashMap

Interfaces

Interfaces

- **Interfaces** in Java are a way to define a **contract** or a **blueprint** of methods that classes can implement.
- They are similar to classes, but they can only contain **abstract** methods (methods without a body) and constant variables (**final** variables).
- An interface is like a set of rules or guidelines. If a class decides to use an interface, it must follow the rules and provide implementations for all the methods declared in the interface.

Why Use Interfaces?

- **To achieve abstraction**: Interfaces allow you to define what methods a class should have, but not how these methods are implemented.
- **For loose coupling**: Interfaces reduce dependencies between code modules, making it easier to change the implementation without affecting users of the interface.
- **To support multiple inheritance**: Java does not allow a class to inherit from multiple classes but allows a class to implement multiple interfaces.

Creating and Implementing an Interface

1. **Declaring an Interface**: An interface is declared using the interface keyword.
2. **Implementing an Interface**: A class implements an interface using the implements keyword and must provide concrete implementations of all its methods.
3. **Default Methods**: Since Java 8, interfaces can have default methods with a body, which are not mandatory for the implementing classes to override.

```java
interface Vehicle {
    void start();
    void stop();
}

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

    public void stop() {
        System.out.println("Car is stopping");
    }
}
```

```java
public class TestInterface {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start(); // Car is starting
        myCar.stop();  // Car is stopping
    }
}
```

Why not just use Inheritance?

- **Multiple Inheritance**: Java does not allow a class to inherit from more than one class. However, a class can implement multiple interfaces. This allows for more flexibility as a class can adhere to multiple contracts or behaviors defined by different interfaces.
- **Loose Coupling**: Using interfaces promotes loose coupling between code. Classes can be more independent and changes in one part of the code are less likely to affect other parts.
- **Clear Contracts**: Interfaces provide a clear contract of what a class should do, but not how to do it. This abstraction allows different classes to implement the same interface in different ways, as long as they adhere to the contract.

The "-able" Suffix in Interface Names

- **Indicates Capability or Behavior**: When an interface name ends with "-able," it suggests that the implementing class possesses a certain capability or behavior. For example, Serializable indicates that an object can be serialized.
- **Descriptive and Intuitive**: This naming convention makes the code more readable and self-descriptive. When you see a class implementing Cloneable, you immediately understand that objects of this class can be cloned.

Built-In Interfaces in Java that ends in -able:

- **Serializable Interface**
- **Cloneable Interface**
- **Comparable Interface**

Comparable Interface

Comparable Interface

- The **Comparable interface** is used to impose a natural ordering on the objects of a class.
- It has a single method, **compareTo(T obj)**, that must be implemented by any class that implements the interface.
- This method compares the current object with the specified object for order. It returns a negative integer, zero, or a positive integer as the current object is less than, equal to, or greater than the specified object.

Why is it Used?

- It allows for sorting collections of objects that implement this interface naturally (e.g., sorting in ascending order).
- It simplifies code by enabling the use of sorting functions provided by Java, such as **Collections.sort()** or **Arrays.sort()**, without the need for a separate comparator.

Comparable Interface

```java
class Person implements Comparable<Person> {
    String name;
    int age;

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

    @Override
    public int compareTo(Person other) {
        return this.age - other.age; // Sorts by age
    }

    @Override
    public String toString() {
        return name + ": " + age;
    }
}
```

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));

        Collections.sort(people);

        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```

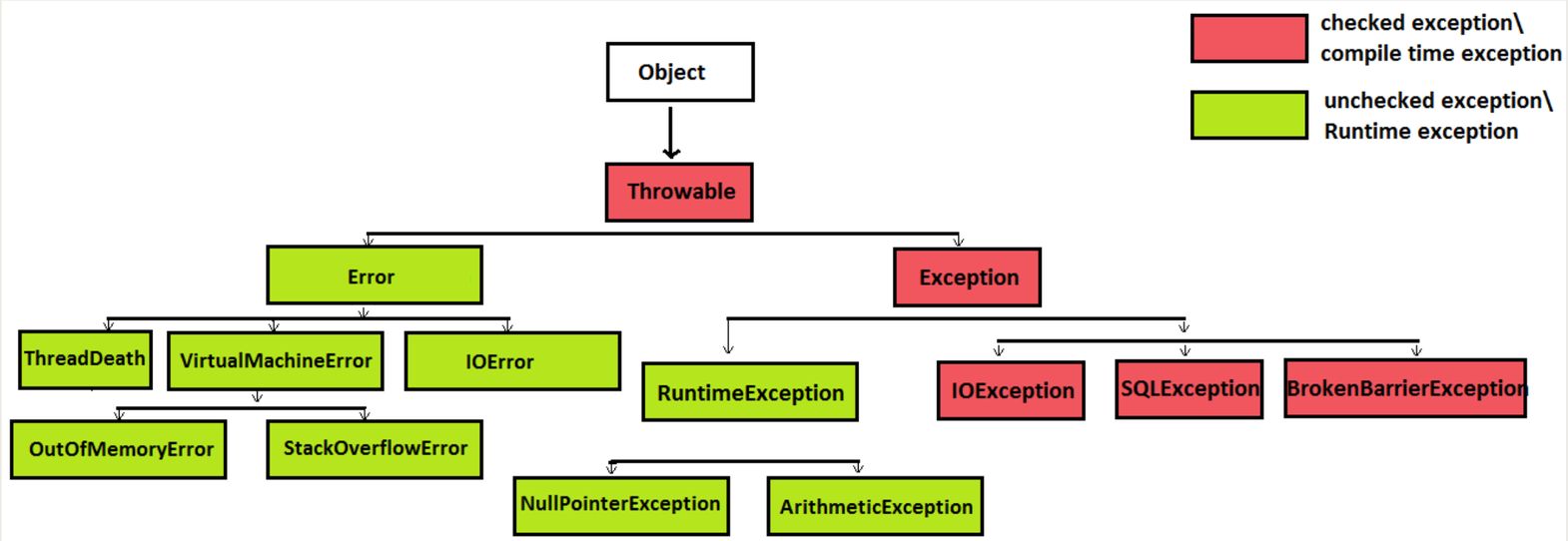
Exceptions

What are Exceptions?

- Exceptions are events that disrupt the normal flow of a program's execution.
- They are objects that represent an error or unexpected condition that occurs during the execution of a program.

Why are Exceptions Used?

- **Error Handling**: They provide a way to catch and handle errors gracefully without crashing the program.
- **Separation of Error Handling Code**: Allows the separation of regular code from error-handling code.
- **Propagating Errors**: Exceptions can be thrown up the call stack until they are caught and handled.



Checked Exceptions

Checked exceptions are exceptions that must be either caught or declared in a method's **throws** clause. They are checked at compile-time, meaning the compiler ensures that these exceptions are properly handled with a try-catch block or declared with a throws clause.

Examples: **IOException**, **SQLException**.

Handling:

- **Try-Catch Block**: You can handle the exception using a try-catch block.
- **Throws Clause**: Alternatively, you can declare the exception in the method signature using the **throws** keyword, indicating that the method may throw this exception.

Checked Exceptions Example

```java
import java.io.*;

public class Main {
    public static void main(String[] args) {
        try {
            FileInputStream file = new FileInputStream("nonexistent.txt");
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

```java
// throws keyword
public static void main(String[] args) {
    try {
        readFile();
    } catch (IOException e) {
        System.out.println("An error occurred while reading the file: " + e.getMessage());
    }
}

// Method declared with 'throws IOException'
public static void readFile() throws IOException {
    // Simulate an operation that might throw an IOException
    boolean fileNotFound = true; // This is just a simulated condition
    if (fileNotFound) {
        throw new IOException("File not found");
    }
}
```

Unchecked Exceptions

Unchecked exceptions include runtime exceptions (**RuntimeException**) and errors (**Error**). They are not checked at compile-time, meaning the compiler does not require them to be caught or declared.

Examples:
NullPointerException,
ArrayIndexOutOfBoundsException,
ArithmeticException

Handling:

- **Optional Handling**: Handling these exceptions is not enforced by the compiler.
- **Try-Catch Block:** You can still handle them using a try-catch block, especially when you anticipate a specific runtime issue.
- **Best Practices:** Often used to indicate programming errors and should be fixed in the code rather than caught.

Unchecked Exceptions Example

```java
public class Main {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero: " + e.getMessage());
        }
    }
}
```

finally keyword

The **finally** keyword in Java is used in exception handling to define a block of code that will always be executed after the try and catch blocks, regardless of whether an exception is thrown or caught. This is especially useful for performing cleanup activities, such as closing files, releasing resources, or other necessary final steps, ensuring that these operations are carried out even if an error occurs.

```java
public static void main(String[] args) {
    try {
        int[] numbers = {1, 2, 3};
        System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("An exception occurred: " + e.getMessage());
    } finally {
        System.out.println("This block is always executed.");
    }
}
```

Processing Files

File Processing

File processing in Java refers to reading from and writing to files. This is a fundamental aspect of many applications, as it allows programs to persist data beyond the runtime of the program, interact with external data, and communicate with other applications through files.

Why It's Used:

1. **Data Storage and Retrieval**: To store and retrieve data, like user settings, game scores, or application logs.
2. **Data Analysis**: To read and process large datasets for analysis or reporting.
3. **Interoperability**: To exchange data with other applications or systems through file formats like CSV, JSON, XML, etc.
4. **Configuration**: To read configuration settings for an application from a file.

Reading Files

One common way to read from files in Java is using a BufferedReader along with a FileReader.

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

public class ReadFileExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing Files

Writing to files in Java can be done using classes like FileWriter or PrintWriter.

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

public class WriteFileExample {
    public static void main(String[] args) {
        try (PrintWriter writer = new PrintWriter(new FileWriter("output.txt"))) {
            writer.println("Hello, World!");
            writer.println("This is an example of writing to a file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Generics

Generics

Generics in Java are a language feature that allows for the definition and use of generic types and methods. They enable classes, interfaces, and methods to be parameterized with types, making your code more flexible and type-safe.

Why Use Generics

1. **Type Safety**: Generics provide stronger type checks at compile time. This means you catch errors in your code earlier (during compilation rather than at runtime).
2. **Elimination** of Casts: With generics, you don't need to cast objects. You know what type of objects you're retrieving from a collection, for example.
3. **Reusability**: You can write a method/class/interface once and use it for any type you like.

Generic Class

```java
class Box<T> {
    private T t; // T stands for "Type"

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.set(10); // No need for casting

        Box<String> stringBox = new Box<>();
        stringBox.set("Hello World");

        System.out.println("Integer Value : " + integerBox.get());
        System.out.println("String Value : " + stringBox.get());
    }
}
```

Box is a generic class with a generic type T. It can be used to store any type of object. When creating an instance of Box, you specify the type of object it will hold, like Integer or String. This makes the code more flexible and safe, as you can't accidentally put a String into a Box intended for Integer

Generics in Collections

```java
List<String> listOfStrings = new ArrayList<>();
listOfStrings.add("Hello");
listOfStrings.add("World");

// No need for casting when retrieving elements
for (String s : listOfStrings) {
    System.out.println(s);
}
```

listOfStrings is a List that can only hold String objects. This ensures type safety, as you can't mistakenly add an Integer or any other type to this list.

Generics in Methods

```java
public static <E> void printArray(E[] inputArray) {
    for (E element : inputArray) {
        System.out.printf("%s ", element);
    }
    System.out.println();
}

public static void main(String args[]) {
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    String[] stringArray = { "Hello", "World" };

    System.out.println("Array integerArray contains:");
    printArray(intArray);   // prints an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // prints a Double array

    System.out.println("\nArray stringArray contains:");
    printArray(stringArray);   // prints a String array
}
```

printArray is a generic method that can print an array of any type, be it Integer, Double, or String. This demonstrates how generics can make your methods more flexible and reusable.

Lambdas

Lambdas

Lambdas, introduced in Java 8, are a powerful and concise way to represent a method interface using an expression. They are particularly useful for creating instances of functional interfaces (interfaces with a single abstract method) in a more concise way.

What are Lambdas?

1. **Simple Definition**: A lambda expression is like a concise representation of a function that can be passed around. It allows you to define an anonymous method (a method without a name) and treat it as an instance of a functional interface.
2. **Functional Interfaces**: These are interfaces with just one abstract method (like Runnable, Callable, Comparator, etc.). Lambda expressions are often used to create instances of these interfaces.

Where are Lambdas Used?

1. **Simplifying Code**: They make your code more readable and concise, especially when using functional interfaces.
2. **Collections**: Lambdas are commonly used with the Collections framework, particularly with methods that take predicates (conditions), comparators, or actions to perform on each element.
3. **Stream API:** In Java 8's Stream API, lambdas are used extensively for operations like filtering, mapping, or iterating over collections in a functional style.
4. **Event Listeners**: In GUI programming, like Swing, lambdas can simplify the creation of event listeners.

Lambdas Example

Using Threads without Lambdas

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running without lambda!");
    }
};

new Thread(runnable).start();
```

Using Threads with Lambdas

```
Runnable runnableLambda = () -> System.out.println("Running with lambda!");
new Thread(runnableLambda).start();
```

Remove names that starts with A

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.removeIf(name -> name.startsWith("A")); // Remove names
```

Sorting a list with and without Lambdas

```
List<String> names = Arrays.asList("Mike", "Anna", "Xenia");

// Without Lambda
Collections.sort(names, new Comparator<String>() {
    public int compare(String a, String b) {
        return a.compareTo(b);
    }
});

// With Lambda
Collections.sort(names, (String a, String b) -> {
    return a.compareTo(b);
});

// Even more concise
Collections.sort(names, (a, b) -> a.compareTo(b));
```

Stream API

Streams

Streams in Java are a key feature introduced in Java 8, providing a new way to work with collections of data in a declarative manner.

What are Streams?

1. **Definition**: A Stream in Java is a sequence of elements supporting sequential and parallel aggregate operations. It's important to note that streams don't store data; they operate on the source data structures (like collections) to perform computations.
2. **Functional Style:** Streams allow you to write code in a functional style, making operations on collections of data more concise and readable.

Where are Streams Used?

Streams are mainly used for processing collections of data, such as lists or sets. They are particularly useful for:

- **Filtering**: Selecting elements based on certain criteria.
- **Mapping**: Transforming elements into other forms.
- **Reducing**: Combining elements to produce a single result.
- **Collecting**: Accumulating elements into a collection or other data structure.

Stream Methods

filter() is used to select elements based on a condition.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave");
List<String> filteredNames = names.stream()
                                  .filter(name -> name.startsWith("A"))
                                  .collect(Collectors.toList());
// Result: ["Alice"]
```

forEach() performs an action for each element in the stream.

```
names.stream()
     .forEach(name -> System.out.println(name));
// Output: Alice Bob Charlie
```

map() applies a function to each element and transforms it into another form.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<Integer> nameLengths = names.stream()
                                 .map(name -> name.length())
                                 .collect(Collectors.toList());
// Result: [5, 3, 7]
```

collect() transforms the stream into a different form, typically a collection like a List or a Set.

```
List<String> namesList = names.stream()
                              .collect(Collectors.toList());
```

Multidimensional Data

Multidimensional Data

refers to arrays that have more than one dimension, such as two-dimensional (2D) arrays or even three-dimensional (3D) arrays. These arrays are essentially "arrays of arrays," where each element in the array itself is another array.

Why Use Multidimensional Arrays

1. **Data Representation**: They are used to represent complex data structures like matrices, grids, or any scenario where data is naturally organized in a multi-level structure.
2. **Games and Simulations**: In game development and simulations, 2D arrays can represent grids or maps, and 3D arrays can represent more complex spatial environments.
3. **Ease of Management**: They can make it easier to manage and work with large amounts of data that are related in a hierarchical or structured way.

Multidimensional Data Example

```java
public class Main {
    public static void main(String[] args) {
        // Declare and initialize a 2D array
        int[][] matrix = {
            {1, 2, 3}, // Row 0
            {4, 5, 6}, // Row 1
            {7, 8, 9}  // Row 2
        };

        // Accessing elements
        System.out.println("Element at matrix[1][1]: " + matrix[1][1]); // Outputs 5

        // Iterating over a 2D array
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Multidimensional Data Example

```java
public class Main {
    public static void main(String[] args) {
        // Declare and initialize a 3D array
        int[][][] threeDArray = {
            {
                {1, 2, 3}, {4, 5, 6}, {7, 8, 9}
            },
            {
                {10, 11, 12}, {13, 14, 15}, {16, 17, 18}
            },
            {
                {19, 20, 21}, {22, 23, 24}, {25, 26, 27}
            }
        };

        // Accessing elements
        System.out.println("Element at threeDArray[0][2][1]: " + threeDArray[0][2][1]); // Outputs 8

        // Iterating over a 3D array
        for (int i = 0; i < threeDArray.length; i++) {
            System.out.println("Matrix " + i + ": ");
            for (int j = 0; j < threeDArray[i].length; j++) {
                for (int k = 0; k < threeDArray[i][j].length; k++) {
                    System.out.print(threeDArray[i][j][k] + " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Any questions?

Jansen Ang