ELYSIUM
SPARK NOTES

ELYSIUM ACADEMY
Milestone of Cognizance

JAVA PROGRAMMING

Elysium Academy Spark Notes

VERSION 2.4

ELYSIUM SPARK NOTES

ELYSIUM ACADEMY Milestone of Cognizance

01. Java Basics

Java is a versatile and widely-used programming language that supports Object-Oriented Programming (OOP) principles. It is known for its portability across platforms (write once, run anywhere), robustness, and security.

Hello World Program

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- **Key Points:**
  - Class Declaration: Java programs are organized into classes, which contain the code. The keyword class defines a class.
  - main() Method: The entry point of any Java application. It has the signature public static void main(String[] args).
  - System.out.println(): A method to print output to the console.

02. Data Types

Java provides a wide range of data types, which are divided into primitive and non-primitive data types.

- **Primitive Data Types**

Data Type	Size	Default Value	Example Usage
byte	1 byte	0	byte b = 100;
short	2 bytes	0	short s = 5000;
int	4 bytes	0	int num = 12345;
long	8 bytes	0L	long l = 123456789L;
float	4 bytes	0.0f	float f = 5.75f;
double	8 bytes	0.0d	double d = 19.99;
char	2 bytes	\u0000	char letter = 'A';
boolean	1 bit	false	boolean flag = true;

- **Non-Primitive Data Types**

Type	Description	Example Usage
String	Sequence of characters (class)	String name = “Java”;
Array	A fixed-size collection of values	int[] nums = {1, 2};
Class	User-defined object	Defined by developers

Example :

```
int age = 30;                 // Primitive data type
String name = “John”;         // Non-primitive data type (String class)
final int DAYS_IN_WEEK = 7;   // Constant variable
```

03. Variables

Variables are containers for storing data. Java is a statically-typed language, meaning each variable must be declared with a data type.

- **Declaring Variables**

```
int age;             // Declaration
age = 25;            // Assignment
int height = 180;    // Declaration + Assignment
final double PI = 3.14159;   // Constant (unchangeable)
```

- **Variable Types:**
  - Local Variables: Declared inside methods, constructors, or blocks and used only within that scope.
  - Instance Variables: Declared inside a class but outside methods, representing the state of an object.
  - Static/Class Variables: Declared with the static keyword; shared among all objects of a class.

04. Operators

Java has various operators to perform operations on variables and values.

- **Arithmetic Operators:**

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remain-der)	a % b

- **Relational/Comparison Operators:**

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

- **Logical Operators:**

Operator	Description	Example
&&	Logical AND	a && b
`		`
!	Logical NOT	!a

- **Assignment Operators:**

Operator	Description	Example
=	Assign value	a = b;
+=	Add and assign	a += b;
-=	Subtract and as-sign	a -= b;
*=	Multiply and assign	a *= b;
/=	Divide and assign	a /= b;

- **Increment/Decrement Operators:**

Operator	Description	Example
++	Increment by 1	a++ or ++a
--	Decrement by 1	a-- or --a

05. Control Flow Statements

Control flow statements determine the flow of execution in a Java program. These include conditional statements, loops, and branching statements.

- **If-Else Statement:**

```
if (age > 18) {
    System.out.println("Adult");
} else if (age == 18) {
    System.out.println("Just turned 18");
} else {
    System.out.println("Minor");
}
```

- **Switch Case:**

```
int day = 3;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    default: System.out.println("Another day");
}
```

- **Loops**
  - For Loop:

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

  - While Loop:

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

- Do-While Loop:

```java
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 5);
```

- **Branching Statements:**

Statement	Description	Example
break	Exits a loop or switch statement	break;
continue	Skips the current iteration	continue;
return	Exits the current method and returns a value	return value;

06. Arrays

Control flow statements determine the flow of execution in a Java program. These include conditional statements, loops, and branching statements.

- **Array Declaration:**

```java
int[] numbers = {1, 2, 3, 4, 5};    // Array Initialization
int[] scores = new int[5];          // Declaring an empty array of size 5
```

- **Accessing Array Elements:**

```java
int firstNumber = numbers[0];    // Access first element
numbers[2] = 10;                  // Modify third element
```

- **Iterating through Arrays:**

```java
for (int num : numbers) {
    System.out.println(num);
}
```

- **Multidimensional Arrays:**

```java
int[][] matrix = { {1, 2, 3}, {4, 5, 6} };  // 2D Array
int value = matrix[0][1];  // Access element at row 0, column 1
```

07. Methods (Functions)

Methods allow reusability and modularity by grouping code into reusable blocks. Java supports both predefined methods (e.g., System.out.println()) and user-defined methods.

- **Defining Methods:**

```java
public static int add(int a, int b) {
    return a + b;
}
```

- **Calling Methods:**

```java
int sum = add(5, 3);     // Method call
System.out.println(sum); // Output: 8
```

- **Method Overloading:**

Method overloading allows methods to have the same name but different parameter lists (type, number, or order of parameters).

```java
public int add(int a, int b) { return a + b; }
public double add(double a, double b) { return a + b; }
```

08. Object-Oriented Programming (OOP)

Java is an object-oriented programming language, which revolves around the following key concepts:

8.1. Classes and Objects

A class is a blueprint for creating objects. An object is an instance of a class.

- Class Definition:

```java
class Dog {
    String name;
    int age;
    void bark() {
        System.out.println(“Woof!”);
    }
}
```

- Creating an Object:

```
Dog myDog = new Dog();    // Creating an object
myDog.name = "Buddy";     // Accessing fields
myDog.bark();             // Calling methods
```

8.2. Constructors

A constructor is a special method used to initialize objects. It has the same name as the class and no return type.

```
class Dog {
    String name;
    int age;

    // Constructor
    Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

- Creating an Object with Constructor:

```
Dog myDog = new Dog("Buddy", 3);
```

8.3. Encapsulation

Encapsulation is the concept of bundling data (fields) and methods that operate on that data into a single unit (class). The fields are made private, and access is provided through public getters and setters.

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

    public String getName() { return name; }
    public void setName(String newName) { this.name = newName; }
}
```

8.4. Inheritance

Inheritance allows a new class to inherit properties and behaviors (methods) from an existing class. The class being inherited from is called the superclass (parent), and the class inheriting is the subclass (child).

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

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

- Using Inheritance:

```
Dog myDog = new Dog();
myDog.eat();    // Inherited from Animal class
myDog.bark();   // Defined in Dog class
```

8.5. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It can occur through method overriding and method overloading.

- Method Overriding:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Woof!");
    }
}
Animal myAnimal = new Dog();
myAnimal.sound();  // Outputs: Woof!
```

8.6. Abstraction

Abstraction is the concept of hiding the internal details of an implementation and only exposing the essential features. In Java, abstraction is achieved using abstract classes or interfaces.

- Abstract Class:

```
abstract class Animal {
    abstract void sound();
    void eat() {
        System.out.println(“This animal eats food.”);
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println(“Woof!”);
    }
}
```

09. Exception Handling

Java uses exceptions to handle runtime errors. Exceptions are objects that represent an error. Java uses the try-catch-finally mechanism for handling exceptions.

- **Try-Catch Block:**

```
try {
    int result = 10 / 0;  // This will throw ArithmeticException
} catch (ArithmeticException e) {
    System.out.println(“Cannot divide by zero!”);
} finally {
    System.out.println(“This block always executes.”);
}
```

- **Throwing Exceptions:**

```
if (age < 18) {

    throw new IllegalArgumentException(“Age must be 18 or older.”);

}
```

- **Types of Exceptions:**
  - **Checked Exceptions:** Exceptions checked at compile-time (e.g., IOException, SQLException).
  - **Unchecked Exceptions:** Exceptions not checked at compile-time but at runtime (e.g., NullPointerException, ArrayIndexOutOfBounds-Exception).

10. File Handling

File handling in Java is done using classes from the java.io package, such as File, Scanner, FileWriter, BufferedReader, etc.

- **Reading from a File:**

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
File file = new File("filename.txt");
Scanner reader = new Scanner(file);
while (reader.hasNextLine()) {
    String data = reader.nextLine();
    System.out.println(data);
}
reader.close();
```

- **Writing to a File:**

```java
import java.io.FileWriter;
import java.io.IOException;
FileWriter writer = new FileWriter("filename.txt");
writer.write("Hello, File!");
writer.close();
```

- **Handling IOException:**

```java
try {
    FileWriter writer = new FileWriter("filename.txt");
    writer.write("Hello, File!");
    writer.close();
} catch (IOException e) {
    System.out.println("An error occurred.");
    e.printStackTrace();
}
```

11. Packages and Imports

A package in Java is a namespace that organizes a set of related classes and interfaces.

- **Using Built-In Packages:**

```java
import java.util.Scanner;
Scanner input = new Scanner(System.in);
int age = input.nextInt();
```

- **Creating a Custom Package:**

```java
package myPackage;
public class MyClass {
    public void displayMessage() {
        System.out.println("Hello from MyClass");
    }
```

- **Using the Custom Package:**

```java
import myPackage.MyClass;
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.displayMessage();
    }
}
```

12. Useful Java Libraries

Java comes with a vast set of libraries, also called APIs (Application Programming Interfaces), that provide a wide variety of functions.

12.1. String Manipulation:

- **length(): Returns the length of the string.**
- **charAt(index): Returns the character at the specified index.**
- **substring(start, end): Returns the substring.**
- **indexOf(char): Returns the index of the first occurrence of the character.**

```java
String str = "Hello, Java!";
System.out.println(str.length());         // Output: 12
System.out.println(str.charAt(0));        // Output: H
System.out.println(str.substring(0, 5));  // Output: Hello
System.out.println(str.indexOf('J'));     // Output: 7
```

12.2. Collections Framework:

Java's Collections Framework provides data structures like Lists, Sets, Maps, etc.

```java
import java.util.ArrayList;
import java.util.HashMap;
ArrayList<String> names = new ArrayList<>();
names.add(“Alice”);
names.add(“Bob”);
HashMap<String, Integer> ages = new HashMap<>();
ages.put(“Alice”, 30);
ages.put(“Bob”, 25);
```

12.3. Date and Time API:

Java 8 introduced the java.time package, which provides better date and time manipulation capabilities.

```java
import java.time.LocalDate;
import java.time.LocalTime;
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.now();
System.out.println(“Current Date: “ + date);
System.out.println(“Current Time: “ + time);
```

13. Java Annotations

Annotations provide metadata about the code and do not change the program's actual logic.

- **Common Annotations:**
  - **@Override:** Indicates that a method is overriding a superclass method.
  - **@Deprecated:** Marks a method as deprecated (i.e., no longer recommended for use).
  - **@SuppressWarnings:** Suppresses compiler warnings.

14. Java Threads and Concurrency

Java provides built-in support for multithreading and concurrent programming through the Thread class and the Runnable interface.

- **Creating a Thread:**
  - **By Extending the Thread Class:**

```java
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}
MyThread thread = new MyThread();
thread.start();
```

  - **By Implementing Runnable Interface:**

```java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable is running");
    }
}

Thread thread = new Thread(new MyRunnable());
thread.start();
```

- **Concurrency Utilities:**

Java provides higher-level concurrency APIs like ExecutorService, Callable, Future, etc., for handling multiple threads efficiently.

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.submit(() -> {
    System.out.println("Task 1");
});
executor.submit(() -> {
    System.out.println("Task 2");
});
executor.shutdown();
```

15. Java Memory Management

Java provides automatic memory management with the help of Garbage Collection. The garbage collector reclaims memory used by objects that are no longer in use.

- **Memory Areas in Java:**
    - **Heap:** The runtime data area where objects are allocated.
    - **Stack:** Stores method calls and local variables.
    - **Method Area:** Contains class structure like metadata, methods, constants, and static variables.
    - **PC Registers:** Program counter register, storing addresses of current instructions.

16. Java Design Patterns

Design patterns are reusable solutions to commonly occurring problems in software design. Some popular patterns in Java include:

- **Creational Patterns:**
    - **Singleton Pattern:** Ensures that a class has only one instance.

```java
public class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Factory Pattern:** Provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

```java
interface Shape {
    void draw();
}
class Circle implements Shape {
    public void draw() {
}
}
class Square implements Shape {
    public void draw() {
        System.out.println("Drawing a Square");
    }
}
class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType == null) return null;
        if (shapeType.equalsIgnoreCase("CIRCLE")) return new Circle();
        if (shapeType.equalsIgnoreCase("SQUARE")) return new Square();
        return null;
    }
}
```

- **Structural Patterns:**
  - **Adapter Pattern:** Allows incompatible interfaces to work together.
  - **Decorator Pattern:** Allows behavior to be added to an individual object, dynamically.
- **Behavioral Patterns:**
  - **Observer Pattern:** Defines a one-to-many dependency between objects.
  - **Strategy Pattern:** Allows algorithms to be selected at runtime.

17. Java Advanced Topics

17.1. Java Streams (Java 8):

Streams are used to process collections of objects. They support operations like filtering, mapping, and reducing in a declarative way.

```java
import java.util.Arrays;
import java.util.List;
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream().filter(s -> s.startsWith("A")).forEach(System.out::println);  // Output: Alice
```

17.2. Lambda Expressions (Java 8):

Lambdas are anonymous functions that provide a concise way to represent instances of functional interfaces.

```java
interface MyFuncInterface {
    void sayHello();
}
MyFuncInterface greet = () -> System.out.println("Hello, World!");
greet.sayHello();  // Output: Hello, World!
```

17.3. Functional Interfaces (Java 8):

A functional interface has only one abstract method but can have multiple default or static methods. Some commonly used functional interfaces are Runnable, Callable, Supplier, Consumer, etc.

```java
@FunctionalInterface
interface MyFuncInterface {
    int add(int a, int b);
}
MyFuncInterface addition = (a, b) -> a + b;
System.out.println(addition.add(5, 3));  // Output: 8
```

18. Java Best Practices

- **Follow Naming Conventions:**
  - Use camelCase for variables and methods (e.g., myVariable).
  - Use PascalCase for class names (e.g., MyClass).
  - Use ALL_CAPS for constants (e.g., MAX_VALUE).
- **DRY Principle (Don't Repeat Yourself):**
  - Write reusable code by avoiding redundancy.
- **Code Documentation:**
  - Use comments and Javadoc for better readability and maintainability.
- **Error Handling:**
  - Catch specific exceptions and avoid using broad exception classes (e.g., avoid catch(Exception e) unless necessary).
- **Immutable Classes:**
  - Make classes immutable to improve thread safety.

- **Avoid Memory Leaks:**
  - **Be cautious about object references, particularly in long-lived objects such as collections.**

18. Conclusion

Java is a powerful, high-level programming language that has withstood the test of time due to its robustness, security, portability, and performance. This comprehensive has covered the fundamental and advanced aspects of Java, offering a valuable reference to both beginners and seasoned developers alike. From core language features like data types, operators, and control flow statements, to advanced topics like OOP principles, file handling, concurrency, and design patterns, Java continues to evolve, supporting modern development paradigms like functional programming and streams.

As you continue learning Java, explore the vast ecosystem of libraries, frameworks, and tools that extend Java's capabilities. Whether you're building enterprise-level applications, Android apps, or working with big data technologies like Hadoop and Spark, Java remains a top choice for developers worldwide. Happy coding!

Click Here To Find Out More

ELYSIUM SPARK NOTES

ELYSIUM ACADEMY® Milestone of Cognizance

Thank you

For Your Learning Today

elysiumacademy.org | info@elysiumacademy.org

Scan Here for More
Spark Notes