Java Basics Notes for ProgSD Exam

Contents

Chapter 1 Java Basics

1.1 Java Setup

- Setting up a Java project involves organizing files, configuring the environment, and understanding the folder structure.
- A typical Java project follows this folder structure:
  - `src/`: Contains source code files (`.java`).
  - `bin/`: Contains compiled `.class` files (output of the Java compiler).
  - `lib/`: Contains external libraries (`.jar` files).
- The entry point of a Java application is the `main()` method inside a `public class`.
- The `main()` method signature:

  ```java
  // Entry point for a Java application
  public static void main(String[] args) {
      // Code to execute
  }
  ```

  - `public`: Accessible to all.
  - `static`: No need to create an instance of the class to call `main()`.
  - `void`: Does not return any value.
  - `String[] args`: Accepts command-line arguments as an array of strings.

Example: Basic Java Project Structure

```
MyJavaProject/
         src/
               Main.java
         bin/
               Main.class
         lib/
```

Example: Creating and Running a Simple Java Program

```java
// File: Main.java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

- Compile the program:

  ```
  javac -d bin src/Main.java
  ```

- Run the program:

  ```
  java -cp bin Main
  ```

- Output:

  ```
  Hello, Java!
  ```

Note: Many IDEs like IntelliJ IDEA or Eclipse automate the compilation and execution process and offer project templates to get started quickly.

Command-Line Arguments Example

```java
// File: Main.java
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("Command-line argument: " +
                args[0]);
        } else {
            System.out.println("No command-line arguments
                provided.");
        }
    }
}
```

- Run with arguments:

  ```
  java -cp bin Main "Hello from the command line!"
  ```

- Output:

  ```
  Command-line argument: Hello from the command line!
  ```

1.2 Java Syntax

- Java uses braces {} to define blocks of code.
- Comments can be single-line (starting with `//`) or multi-line (enclosed in `/* */`).
- Variable assignment requires explicit declaration of the data type.
- Common data types include:
  - `int`: Integer values.
  - `double`: Decimal numbers.
  - `String`: Text or string data.
  - `boolean`: Boolean values (`true`, `false`).
  - `Array`: Ordered, fixed-size collection of items.
  - `List`: Ordered, dynamic collection of items (requires `import java.util.List;` and `import java.util.Arrays;`).
  - `HashMap`: Key-value pairs (requires `import java.util.HashMap;`).
  - `HashSet`: Unordered collection of unique items (requires `import java.util.HashSet;` and `import java.util.Arrays;`).
  - `null`: Represents the absence of a value.

Example:

```java
// Integer: 10
int a = 10;

// Double: 3.14
double b = 3.14;

// String: "Hello"
String c = "Hello";

// Boolean: true
boolean d = true;

// Array: [1, 2, 3]
int[] e = {1, 2, 3};

// List: [1, 2, 3]
// Required imports:
// import java.util.List;
// import java.util.Arrays;
List<Integer> f = Arrays.asList(1, 2, 3);

// HashMap: {’key’: ’value’}
// Required import:
// import java.util.HashMap;
HashMap<String, String> g = new HashMap<>();
g.put("key", "value");

// HashSet: {1, 2, 3}
// Required imports:
// import java.util.HashSet;
// import java.util.Arrays;
HashSet<Integer> h = new HashSet<>(Arrays.asList(1, 2, 3));

// Null value
String i = null;
```

1.3 Variables in Java

- Variables store data that can be manipulated and reused.
- Java requires explicit declaration of variable types.
- Common data types include:
  - `int`: Integer values.
  - `float`: Decimal numbers.
  - `double`: Double-precision floating-point numbers.
  - `char`: Single characters.
  - `boolean`: `true` or `false`.
  - `String`: Sequence of characters (not a primitive type).
- Java provides strong typing, which ensures variables cannot store incompatible data types.

1.3.1 Type Conversions

- Java supports two types of conversions:
  - **Implicit (Widening)**: Converts smaller types to larger types automatically.
  - **Explicit (Narrowing)**: Requires casting to convert larger types to smaller types.

Example: Implicit Conversion

```
// Implicit conversion
int num = 100;
double convertedNum = num; // int to double
System.out.println(convertedNum); // Outputs: 100.0
```

Example: Explicit Conversion

```
// Explicit conversion
double num = 100.5;
int convertedNum = (int) num; // double to int
System.out.println(convertedNum); // Outputs: 100
```

1.3.2 Integer Division

- Dividing integers results in integer division, truncating the decimal part.
- Use explicit conversion for floating-point division.

Example: Integer Division

```java
// Integer division
int a = 7;
int b = 2;
int result = a / b; // Result is 3
System.out.println(result);

// Explicit conversion for floating-point division
double floatResult = (double) a / b; // Result is 3.5
System.out.println(floatResult);
```

1.4 Storing and Manipulating Values

- Variables in Java must be declared with a type, such as `int`, `double`, `String`, `ArrayList`, and more.
- Operations on variables depend on their data type and include arithmetic, concatenation, indexing, and more.
- Imports are needed for data structures like `ArrayList` and `HashMap`.

Example: Integer Operations

```
// Integer variable
int a = 10;

// Addition
int b = a + 5; // 15

// Multiplication
int c = a * 2; // 20

// Exponentiation
double d = Math.pow(a, 3); // Requires import
    java.lang.Math; result: 1000.0
```

Example: Float (Double) Operations

```
// Float variable (in Java, use double for precision)
double x = 3.14;

// Division
double y = x / 2; // 1.57

// Multiplication
double z = x * 3; // 9.42
```

Example: String Operations

```
// String variable
String s = "Hello";

// Concatenation
String greeting = s + " World"; // "Hello World"

// Repetition (Java doesn't support direct repetition like
    Python)
// Use a loop or repeat explicitly
String repeated = s.repeat(3); // Requires Java 11+;
    "HelloHelloHello"

// Substring
String substring = s.substring(1, 4); // "ell"
```

Example: ArrayList Operations (Equivalent to Python Lists)

```
import java.util.ArrayList; // Required for ArrayList

// ArrayList variable
ArrayList<Integer> numbers = new ArrayList<>();

// Add elements
numbers.add(1); // [1]
numbers.add(2); // [1, 2]
numbers.add(3); // [1, 2, 3]

// Indexing
int firstElement = numbers.get(0); // 1

// Slicing (Java doesn't support slicing directly; use
    subList)
ArrayList<Integer> subset = new
    ArrayList<>(numbers.subList(1, 3)); // [2, 3]

// Concatenation
ArrayList<Integer> combined = new ArrayList<>(numbers);
combined.addAll(new ArrayList<>(List.of(4, 5))); //
    Requires import java.util.List; [1, 2, 3, 4, 5]
```

Example: Tuple-Like Structures (Using Arrays or Custom Classes)

```
// Using an Array for tuples
int[] coordinates = {10, 20, 30};

// Indexing
int xCoord = coordinates[0]; // 10

// Slicing (not supported directly; create a new array)
int[] subset = Arrays.copyOfRange(coordinates, 1, 3); //
    Requires import java.util.Arrays; {20, 30}

// Concatenation (manually combine arrays)
int[] newCoords = Arrays.copyOf(coordinates,
    coordinates.length + 2); // {10, 20, 30, 0, 0}
System.arraycopy(new int[]{40, 50}, 0, newCoords,
    coordinates.length, 2); // {10, 20, 30, 40, 50}
```

Example: HashMap Operations (Equivalent to Python Dictionaries)

```
import java.util.HashMap; // Required for HashMap

// HashMap variable
HashMap<String, Object> student = new HashMap<>();
student.put("name", "Alice");
student.put("age", 25);
student.put("major", "Math");

// Access value by key
String name = (String) student.get("name"); // "Alice"

// Add or modify a key-value pair
student.put("grade", "A"); // {"name": "Alice", "age": 25,
    "major": "Math", "grade": "A"}
student.put("age", 26); // {"name": "Alice", "age": 26,
    "major": "Math", "grade": "A"}
```

Example: HashSet Operations (Equivalent to Python Sets)

```java
import java.util.HashSet; // Required for HashSet

// HashSet variable
HashSet<Integer> uniqueNumbers = new HashSet<>();
uniqueNumbers.add(1);
uniqueNumbers.add(2);
uniqueNumbers.add(3);

// Add an element
uniqueNumbers.add(4); // {1, 2, 3, 4}

// Union (combine sets)
HashSet<Integer> otherNumbers = new HashSet<>(Set.of(3, 4,
    5)); // Requires import java.util.Set
uniqueNumbers.addAll(otherNumbers); // {1, 2, 3, 4, 5}

// Intersection (common elements)
uniqueNumbers.retainAll(otherNumbers); // {3, 4}
```

Example: Boolean Operations

```java
// Boolean variable
boolean flag = true;

// Logical NOT
boolean notFlag = !flag; // false

// Logical AND
boolean resultAnd = flag && false; // false

// Logical OR
boolean resultOr = flag || false; // true
```

Example: Null Values (Equivalent to Python's None)

```java
// Null variable
String x = null;

// Checking if value is null
boolean isNull = (x == null); // true
```

1.5 Enum and Constants

1.5.1 Enumerated Types (Enums)

- Enums in Java are a special data type used to define a collection of constants.
- Useful for representing fixed sets of values, like days of the week or directions.
- Enums are type-safe, ensuring only valid values are used.

Example: Defining and Using an Enum

```
// Define an enum
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
        SUNDAY
}

public class EnumExample {
    public static void main(String[] args) {
        Day today = Day.FRIDAY;
        System.out.println("Today is: " + today); //
            Output: Today is: FRIDAY
    }
}
```

Example: Iterating Over Enum Values

```
public class EnumIteration {
    public static void main(String[] args) {
        for (Day day : Day.values()) {
            System.out.println(day);
        }
    }
}
// Output: MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY
    SATURDAY SUNDAY
```

1.5.2 Constants

- Constants are variables whose values cannot be changed after initialization.
- Use the `final` keyword to define constants.
- Combine `static` with `final` for constants that belong to the class rather than an instance.

Example: Defining and Using Constants

```
public class ConstantsExample {
    // Define constants
    public static final double PI = 3.14159;
    public static final String APP_NAME = "Java Basics";

    public static void main(String[] args) {
        System.out.println("Value of PI: " + PI); //
            Output: Value of PI: 3.14159
        System.out.println("App Name: " + APP_NAME); //
            Output: App Name: Java Basics
    }
}
```

1.6 Type Conversions

- Java supports two types of type conversions:
  - **Implicit Conversion (Widening Conversion)**:
    - Automatic conversion from a smaller to a larger data type.
    - Example: `int` to `long`, `float` to `double`.
  - **Explicit Conversion (Narrowing Conversion)**:
    - Requires explicit casting to convert a larger type to a smaller type.
    - Example: `double` to `int`.
- Use `parse` methods from wrapper classes to convert strings to primitives, such as `Integer.parseInt()`.

Example: Implicit Conversion (Widening)

```java
int num = 100;
long longNum = num; // Implicit conversion
double doubleNum = longNum; // Implicit conversion
System.out.println(doubleNum); // Outputs: 100.0
```

Example: Explicit Conversion (Narrowing)

```java
double doubleValue = 99.99;
int intValue = (int) doubleValue; // Explicit conversion
System.out.println(intValue); // Outputs: 99
```

1.6.1 Converting Between Strings and Primitives

- Use wrapper classes to convert strings to primitives:
  - `Integer.parseInt()` for `int`.
  - `Double.parseDouble()` for `double`.
- Use `String.valueOf()` to convert primitives to strings.

Example: String to Primitive

```java
String str = "123";
int num = Integer.parseInt(str); // Convert string to int
double decimal = Double.parseDouble("45.67"); // Convert
    string to double
System.out.println(num + decimal); // Outputs: 168.67
```

Example: Primitive to String

```java
int num = 123;
String str = String.valueOf(num); // Convert int to string
System.out.println("Number: " + str); // Outputs: Number:
    123
```

1.7 Strings

- Strings in Java are objects that represent sequences of characters.
- Strings are immutable, meaning they cannot be changed after they are created.
- Commonly used methods in the `String` class:
  - `length()`: Returns the length of the string.
  - `charAt(index)`: Returns the character at a specific index.
  - `substring(start, end)`: Extracts a substring.
  - `toUpperCase()` and `toLowerCase()`: Converts the string to uppercase or lowercase.
  - `trim()`: Removes whitespace from the beginning and end.
  - `equals()` and `equalsIgnoreCase()`: Compares two strings for equality.
  - `compareTo()`: Compares two strings lexicographically.
  - `split(delimiter)`: Splits the string based on a delimiter and returns an array.

Example: Basic String Operations

```java
String str = "Hello, Java!";

// Length of the string
System.out.println(str.length()); // Outputs: 12

// Character at index 7
System.out.println(str.charAt(7)); // Outputs: J

// Substring
System.out.println(str.substring(7, 11)); // Outputs: Java

// Convert to uppercase
System.out.println(str.toUpperCase()); // Outputs: HELLO,
    JAVA!

// Remove leading and trailing spaces
String spacedStr = " Hello ";
System.out.println(spacedStr.trim()); // Outputs: Hello
```

1.7.1 String Concatenation

- Use the `+` operator to concatenate strings.
- For efficient concatenation in loops, use `StringBuilder`.

Example: Concatenation Using +

```java
String greeting = "Hello";
String name = "Alice";
System.out.println(greeting + ", " + name + "!"); //
    Outputs: Hello, Alice!
```

Example: Efficient Concatenation With `StringBuilder`

```java
StringBuilder builder = new StringBuilder("Hello");
builder.append(", ").append("World!");
System.out.println(builder.toString()); // Outputs: Hello,
    World!
```

1.7.2 String Comparison

- Use `equals()` to compare content.
- Use `==` to check reference equality (memory location).

Example: String Comparison

```java
String str1 = "Java";
String str2 = "Java";
String str3 = new String("Java");

// Content comparison
System.out.println(str1.equals(str2)); // Outputs: true
System.out.println(str1.equals(str3)); // Outputs: true

// Reference comparison
System.out.println(str1 == str2); // Outputs: true (same
    memory)
System.out.println(str1 == str3); // Outputs: false
    (different memory)
```

1.8 Decision Making

- Java supports conditional branching with `if`, `else if`, and `else`.
- Logical operators like `&&` (and), `||` (or), and `!` (not) can combine conditions for more complex decisions.

Example: Basic `if` Statement

```java
// Basic if statement
int x = 10;
if (x > 5) {
    String result = "x is greater than 5"; // "x is greater
        than 5"
}
```

Example: `if-else` Statement

```java
// If-else statement
int x = 3;
String result;
if (x > 5) {
    result = "x is greater than 5";
} else {
    result = "x is not greater than 5"; // "x is not
        greater than 5"
}
```

Example: `if-else if-else` Statement

```java
// If-else if-else statement
int x = 5;
String result;
if (x > 5) {
    result = "x is greater than 5";
} else if (x == 5) {
    result = "x is equal to 5"; // "x is equal to 5"
} else {
    result = "x is less than 5";
}
```

Example: Logical Operators

```java
// Using logical operators
int x = 10;
int y = 5;

// Logical AND
if (x > 5 && y < 10) {
    String result = "Both conditions are True"; // "Both conditions are True"
}

// Logical OR
if (x < 5 || y < 10) {
    String result = "At least one condition is True"; // "At least one condition is True"
}

// Logical NOT
if (!(x < 5)) {
    String result = "x is not less than 5"; // "x is not less than 5"
}
```

Example: Combining Logical Operators with `if-else if-else`

```java
// Combining logical operators with if-else if-else
int x = 8;
int y = 3;
String result;

if (x > 5 && y > 5) {
    result = "Both x and y are greater than 5";
} else if (x > 5 || y > 5) {
    result = "At least one of x or y is greater than 5"; // "At least one of x or y is greater than 5"
} else {
    result = "Neither x nor y is greater than 5";
}
```

1.9 Repetition

- Java supports loops for repeating code:
  - `for` loops are used to iterate over a range or an array.
  - Enhanced `for` loops (also known as ”for-each”) iterate over collections like arrays and lists.
  - `while` loops execute as long as a condition is `true`.
  - Nested loops allow iteration over multiple levels, including arrays or multidimensional arrays.

Example: Basic `for` Loop

```java
// Iterating over a range
for (int i = 0; i < 5; i++) {
    int result = i; // 0, 1, 2, 3, 4
}
```

Example: Enhanced `for` Loop with an Array

```java
// Iterating over an array
String[] fruits = {"apple", "banana", "cherry"};
for (String fruit : fruits) {
    String result = fruit; // "apple", "banana", "cherry"
}
```

Example: `while` Loop

```java
// While loop with a condition
int x = 0;
while (x < 5) {
    x++; // 1, 2, 3, 4, 5
}
```

Example: Nested Loops

```java
// Nested for loops
for (int i = 0; i < 2; i++) { // Outer loop
    for (int j = 0; j < 3; j++) { // Inner loop
        int[] result = {i, j};
        // (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)
    }
}
```

Example: Nested `for` Loop in a 2D Array

```java
// Example of iterating over a 2D array using nested loops
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

for (int i = 0; i < matrix.length; i++) { // Outer loop for
    rows
    for (int j = 0; j < matrix[i].length; j++) { // Inner
        loop for columns
        System.out.println("Element at [" + i + "][" + j +
            "]: " + matrix[i][j]);
        // Example output:
        // Element at [0][0]: 1
        // Element at [0][1]: 2
        // Element at [0][2]: 3
        // ...
    }
}
```

1.10 Functions

- Functions in Java encapsulate reusable blocks of code, often referred to as methods.
- Methods can:
  - Be defined using the `public`, `private`, or `protected` access modifiers.
  - Accept parameters for input.
  - Return values or perform actions without returning.
  - Be overloaded to allow multiple methods with the same name but different parameters (overloading enables handling of varied inputs).
- **Method Declaration Syntax:**
  - The basic syntax to declare a method in Java:

    ```
    // Access Modifier ReturnType
        MethodName(ParameterList) {
        // Method body
        // return statement (if needed)
    // }
    ```

  - Example:

    ```
    // A simple method to add two numbers
    public int addNumbers(int a, int b) {
        return a + b;
    }
    ```

- **Difference Between Functions and Methods:**
  - Functions are standalone blocks of code, as in some programming languages like Python or C.
  - Methods are functions that are associated with objects or classes (e.g., in Java or Python classes).
  - In Java, all functions are methods because they belong to a class.

Example: Defining a Function Without Parameters

```
// Function without parameters
public static void greet() {
    String result = "Hello, World!";
    System.out.println(result); // Output: "Hello, World!"
}
```

Example: Defining a Function With Parameters

```java
// Function with parameters
public static int add(int a, int b) {
    return a + b; // Sum of a and b
}
```

Example: Function With Return Value

```java
// Function that returns a value
public static int multiply(int a, int b) {
    return a * b;
}

// Usage
int result = multiply(2, 3); // 6
System.out.println("Result: " + result);
```

Example: Function Without Return Value

```java
// Function without a return statement
public static void printMessage(String message) {
    System.out.println("Message: " + message);
}

// Usage
printMessage("Hello, Java!"); // Output: "Message: Hello, Java!"
```

Example: Function With Default Arguments (Using Overloading)

```java
// Function with default parameters (achieved via overloading)
public static String greet(String name) {
    return "Hello, " + name + "!";
}

// Overloaded method to handle default case
public static String greet() {
    return greet("Guest"); // Default name is "Guest"
}

// Usage
String result1 = greet(); // "Hello, Guest!"
String result2 = greet("Alice"); // "Hello, Alice!"
System.out.println(result1);
System.out.println(result2);
```

1.11 Managing Variables and Side Effects

- Java variables have different scopes:
  - **Local Scope**: Variables declared inside a method or block are accessible only within that scope.
  - **Class-Level Scope**: Variables declared as instance variables (outside methods) are accessible throughout the class.
  - **Global-like Scope**: Static variables (class variables) are shared across all instances of a class.
- Improper scoping can lead to unexpected side effects, such as unintended modifications to class or instance variables.

Example: Local Scope

```java
// Local variable inside a method
public class ScopeExample {
    public void localScope() {
        int localVar = 10; // Local variable
        System.out.println("Local variable: " + localVar);
            // Output: 10
    }
}
```

Example: Class-Level Scope

```java
// Class-level variable (instance variable)
public class ScopeExample {
    private int instanceVar = 20;

    public void accessInstanceVar() {
        System.out.println("Instance variable: " +
            instanceVar); // Output: 20
    }
}
```

Example: Static Variables (Global-like Scope)

```java
// Static variable (shared across instances)
public class ScopeExample {
    private static int sharedVar = 50;

    public void modifySharedVar() {
        sharedVar += 10; // Modify the shared variable
        System.out.println("Shared variable: " +
            sharedVar); // Output varies
    }
}
```

Example: Preventing Side Effects by Proper Encapsulation

```java
// Use encapsulation to control access
public class ScopeExample {
    private int sensitiveData = 100; // Private variable

    public int getSensitiveData() {
        return sensitiveData; // Provide read access
    }

    public void setSensitiveData(int newValue) {
        sensitiveData = newValue; // Provide controlled
            write access
    }
}

// Usage
ScopeExample obj = new ScopeExample();
System.out.println(obj.getSensitiveData()); // Output: 100
obj.setSensitiveData(200);
System.out.println(obj.getSensitiveData()); // Output: 200
```

Example: Unintended Side Effects

```
public class SideEffectExample {
    private static int value = 50;

    public void unintendedModification() {
        int value = 100; // This creates a local variable,
            not modifying the static one
    }

    public static void main(String[] args) {
        SideEffectExample obj = new SideEffectExample();
        obj.unintendedModification();
        System.out.println(value); // Output: 50 (static
            variable remains unchanged)
    }
}
```

Example: Managing Side Effects Properly

```
public class SideEffectExample {
    private static int value = 50;

    public void modifyStaticValue() {
        value = 100; // Explicitly modifies the static
            variable
    }

    public static void main(String[] args) {
        SideEffectExample obj = new SideEffectExample();
        obj.modifyStaticValue();
        System.out.println(value); // Output: 100
    }
}
```

1.12 Object-Oriented Programming (OOP)

- Java supports Object-Oriented Programming (OOP) using classes and objects.
- **Classes** define objects with attributes (fields) and methods.
- Constructors (methods with the same name as the class) are used to initialize attributes when creating objects.
- Visibility modifiers (`public`, `private`, `protected`, `default`) control access to class members:
  - `public`: Accessible from anywhere.
  - `private`: Accessible only within the same class.
  - `protected`: Accessible within the same package and subclasses.
  - `default` (no modifier): Accessible only within the same package.
- Use **Getters and Setters** to control access to private fields.
- **Static Members:** Shared among all instances of the class. Accessed using the class name.
- **Inheritance** allows child classes to extend or override the functionality of parent classes.
- **Polymorphism:** Enables using a parent reference to call overridden methods in a child class.

Example: Defining a Class With Attributes and Methods

```java
// Class definition
public class Animal {
    private String name;

    // Constructor to initialize attributes
    public Animal(String name) {
        this.name = name;
    }

    // Getter
    public String getName() {
        return name;
    }

    // Setter
    public void setName(String name) {
        this.name = name;
    }

    // Method
    public String speak() {
        return name + " makes a sound.";
    }
}

// Creating an object
public class Main {
    public static void main(String[] args) {
        Animal dog = new Animal("Dog");
        dog.setName("Buddy"); // Using the setter
        String result = dog.speak(); // "Buddy makes a
            sound."
        System.out.println(result);
    }
}
```

Example: Static Members

```java
public class Counter {
    private static int count = 0; // Static field

    // Static method
    public static int getCount() {
        return count;
    }

    // Constructor increments count
    public Counter() {
        count++;
    }
}

// Accessing static members
public class Main {
    public static void main(String[] args) {
        new Counter(); // Increment count
        new Counter(); // Increment count
        System.out.println("Count: " + Counter.getCount());
            // Output: Count: 2
    }
}
```

Example: Inheritance and Method Overriding

```java
// Parent class
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public String speak() {
        return name + " makes a sound.";
    }
}

// Child class inheriting from Animal
public class Dog extends Animal {
    public Dog(String name) {
        super(name); // Call the parent class constructor
    }

    @Override
    public String speak() { // Overriding the parent method
        return name + " barks.";
    }
}

// Creating objects
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog("Buddy"); // Polymorphism
        System.out.println(dog.speak()); // "Buddy barks."
    }
}
```

Example: Accessing Parent Methods in Child Classes

```java
// Parent class
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public String speak() {
        return name + " makes a sound.";
    }
}

// Child class
public class Bird extends Animal {
    public Bird(String name) {
        super(name); // Call the parent class constructor
    }

    @Override
    public String speak() {
        // Call the parent method
        String parentSpeak = super.speak();
        return parentSpeak + " and sings.";
    }
}

// Creating an object
public class Main {
    public static void main(String[] args) {
        Bird bird = new Bird("Robin");
        System.out.println(bird.speak()); // "Robin makes a
            sound and sings."
    }
}
```

1.13 Advanced Object-Oriented Programming (OOP)

1.13.1 Polymorphism

- Polymorphism allows methods to perform different tasks based on the object that invokes them.
- Two main types:
  - **Compile-Time Polymorphism (Method Overloading)**: Methods with the same name but different parameter lists.
  - **Run-Time Polymorphism (Method Overriding)**: Methods in a subclass override methods in the parent class.

Example: Method Overloading (Compile-Time Polymorphism)

```
public class OverloadingExample {
    // Overloaded methods
    public int add(int a, int b) {
        return a + b;
    }

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

    public static void main(String[] args) {
        OverloadingExample example = new
            OverloadingExample();
        System.out.println(example.add(5, 3));   // Output: 8
        System.out.println(example.add(2.5, 3.5)); //
            Output: 6.0
    }
}
```

Example: Method Overriding (Run-Time Polymorphism)

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

// Child class
public class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Dog barks.");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Polymorphism
        myDog.speak(); // Output: Dog barks.
    }
}
```

1.13.2 Abstract Classes and Methods

- Abstract classes cannot be instantiated and may include both abstract and concrete methods.
- Abstract methods are declared without implementation; subclasses must override them.

Example: Abstract Class and Methods

```java
public abstract class Shape {
    protected String color;
    public Shape(String color) {
        this.color = color;
    }
    // Abstract method
    public abstract double area();

    // Concrete method
    public String getColor() {
        return color;
    }
}

// Subclass
public class Circle extends Shape {
    private double radius;
    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

public class AbstractExample {
    public static void main(String[] args) {
        Shape circle = new Circle("Red", 5);
        System.out.println("Color: " + circle.getColor());
            // Output: Color: Red
        System.out.println("Area: " + circle.area()); //
            Output: Area: 78.53981633974483
    }
}
```

1.13.3 Interfaces

- Interfaces define a contract that implementing classes must follow.
- All methods in interfaces are implicitly public and abstract.
- A class can implement multiple interfaces, overcoming the limitation of single inheritance.

Example: Defining and Implementing an Interface

```
// Define an interface
public interface Vehicle {
    void start();
    void stop();
}

// Implement the interface
public class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car starts.");
    }

    @Override
    public void stop() {
        System.out.println("Car stops.");
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start(); // Output: Car starts.
        car.stop(); // Output: Car stops.
    }
}
```

1.13.4 Difference Between Abstract Classes and Interfaces

- Abstract classes can have both abstract and concrete methods; interfaces can only have abstract methods (prior to Java 8).
- A class can inherit from one abstract class but implement multiple interfaces.
- Interfaces are ideal for defining behavior contracts, while abstract classes provide a base for similar types.

Comparison Table:

Feature	Abstract Class	Interface
Methods	Abstract and concrete	Abstract only (pre-Java 8)
Inheritance	Single	Multiple
Fields	Can have fields	Constants only
Use Case	Base class for related objects	Define behavior contracts

1.14 Core Object Methods and java.lang.Object

- `java.lang.Object` is the root class of all Java classes. Every class implicitly inherits from `Object`.
- Commonly used methods inherited from `Object`:
  - `equals(Object obj)`: Compares the current object to another for equality.
  - `hashCode()`: Returns a hash code value for the object.
  - `toString()`: Returns a string representation of the object.
  - `clone()`: Creates a copy of the object (must implement `Cloneable`).
  - `getClass()`: Returns the runtime class of the object.

1.14.1 Using `equals()` and `hashCode()`

- The `equals()` method determines if two objects are logically equivalent.
- The `hashCode()` method generates an integer used in hash-based collections (e.g., `HashMap`).
- Both should be overridden together to maintain consistency in custom classes.

Example: Overriding `equals()` and `hashCode()`:

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

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Reference equality
        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); // Uses
            java.util.Objects for consistency
    }
}
```

1.14.2 Using `toString()`

- The `toString()` method returns a string representation of an object.
- By default, it returns the class name and hash code.
- Override it for meaningful output.

Example: Overriding `toString()`:

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

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

    @Override
    public String toString() {
        return "Person{name=’" + name + "’, age=" + age +
            "}";
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 25);
        System.out.println(person); // Output:
            Person{name=’Alice’, age=25}
    }
}
```

1.14.3 Default Implementation of `equals()` and `hashCode()`

- The default `equals()` in `Object` checks reference equality using `==`.
- The default `hashCode()` generates distinct integers for different objects (not consistent for logically equivalent objects unless overridden).

1.14.4 Using `clone()`

- `clone()` creates a shallow copy of the object.
- To use `clone()`, a class must implement the `Cloneable` interface, or `CloneNotSupportedException` is thrown.

Example: Using `clone()`:

```java
public class Person implements Cloneable {
    private String name;
    private int age;

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

    @Override
    protected Object clone() throws
        CloneNotSupportedException {
        return super.clone(); // Calls Object's clone method
    }

    public static void main(String[] args) {
        try {
            Person original = new Person("Alice", 25);
            Person copy = (Person) original.clone();
            System.out.println("Original: " + original);
            System.out.println("Copy: " + copy);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

1.15 Data Structures and Collections Framework

1.15.1 Lists

- Lists in Java are implemented via `ArrayList` or `LinkedList`, both part of the `java.util` package.
- Lists are ordered, allow duplicates, and are dynamically resizable.
- Common operations:
  - `add(x)`: Add an item to the list.
  - `remove(index)`: Remove the item at the specified index.
  - `get(index)`: Retrieve an item at a specified index.
  - `size()`: Get the number of elements.
  - `contains(x)`: Check if an element exists.
  - `clear()`: Remove all elements from the list.

Example: ArrayList Operations

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple"); // ["Apple"]
        fruits.add("Banana"); // ["Apple", "Banana"]
        fruits.add("Cherry"); // ["Apple", "Banana",
            "Cherry"]

        fruits.remove(1); // ["Apple", "Cherry"]
        System.out.println(fruits.get(0)); // "Apple"
        System.out.println(fruits.contains("Banana")); //
            false
        fruits.clear(); // []
    }
}
```

1.15.2 Maps

- Maps in Java store key-value pairs and are part of the `java.util` package.
- Keys are unique, and values can be of any type.
- Common implementations include `HashMap`, `TreeMap`, and `LinkedHashMap`.
- Common operations:
  - `put(key, value)`: Add or update a key-value pair.
  - `get(key)`: Retrieve the value for a key.
  - `remove(key)`: Remove a key-value pair.
  - `containsKey(key)`: Check if a key exists.
  - `keySet()`: Retrieve all keys.

Example: HashMap Operations

```java
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> scores = new HashMap<>();
        scores.put("Alice", 90); // {"Alice": 90}
        scores.put("Bob", 85); // {"Alice": 90, "Bob": 85}

        System.out.println(scores.get("Alice")); // 90
        System.out.println(scores.containsKey("Charlie"));
            // false

        scores.remove("Alice"); // {"Bob": 85}
        System.out.println(scores.keySet()); // ["Bob"]
    }
}
```

1.15.3 Sets

- Sets are unordered collections of unique elements.
- Common implementations include `HashSet` and `TreeSet`.
- Common operations:
  - `add(x)`: Add an element.
  - `remove(x)`: Remove an element.
  - `contains(x)`: Check if an element exists.

Example: HashSet Operations

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<String> colors = new HashSet<>();
        colors.add("Red"); // ["Red"]
        colors.add("Blue"); // ["Red", "Blue"]
        colors.add("Red"); // ["Red", "Blue"] (no
            duplicates)

        System.out.println(colors.contains("Blue")); // true
        colors.remove("Red"); // ["Blue"]
    }
}
```

1.15.4 Queues

- Queues are part of the `java.util` package and follow the First-In-First-Out (FIFO) principle.
- Deques allow fast insertion and deletion from both ends.
- Common operations:
  - `add(x)`: Add an item to the end.
  - `remove()`: Remove the first item.
  - `peek()`: View the first item without removing it.

Example: ArrayDeque Operations

```java
import java.util.ArrayDeque;

public class Main {
    public static void main(String[] args) {
        ArrayDeque<Integer> queue = new ArrayDeque<>();
        queue.add(1); // [1]
        queue.add(2); // [1, 2]
        queue.add(3); // [1, 2, 3]

        System.out.println(queue.peek()); // 1
        queue.remove(); // [2, 3]
    }
}
```

1.15.5 Iteration and Enhanced For Loop

- Iteration over collections is simplified using the enhanced `for-each` loop.
- Applicable to arrays, lists, sets, and other iterable collections.

Example: Iterating Over a List

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        for (String fruit : fruits) {
            System.out.println(fruit); // "Apple", "Banana",
                "Cherry"
        }
    }
}
```

Example: Iterating Over a Map

```java
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> scores = new HashMap<>();
        scores.put("Alice", 90);
        scores.put("Bob", 85);

        for (String key : scores.keySet()) {
            System.out.println(key + ": " + scores.get(key));
            // Output: "Alice: 90", "Bob: 85"
        }
    }
}
```

1.15.6 Arrays

- Arrays in Java are fixed-size collections of elements of the same type.
- Common operations:
  - Access elements using indices.
  - Use `Arrays.sort(array)` to sort an array.
  - Use `array.length` to get the number of elements.

Example: Array Operations

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] numbers = {4, 2, 3, 1};
        Arrays.sort(numbers); // [1, 2, 3, 4]

        for (int num : numbers) {
            System.out.println(num); // 1, 2, 3, 4
        }
    }
}
```

1.16 Packages and Libraries

- Java uses **packages** to group related classes and interfaces, making code modular, reusable, and easier to manage.
- **Built-in Packages:** Java provides many built-in packages, such as:
  - `java.util`: Contains utility classes like `ArrayList`, `HashMap`, `Date`, etc.
  - `java.io`: Includes classes for input and output operations (e.g., `File`, `BufferedReader`).
  - `java.lang`: Contains fundamental classes like `String`, `Math`, and `Object`.
- **Creating Custom Packages:**
  - Custom packages help organize user-defined classes and interfaces.
  - Use the `package` keyword at the top of the file.
  - To use a custom package, import it using the `import` keyword.

1.16.1 Example: Creating and Using a Custom Package

```
// File: MyPackage/MyClass.java
package MyPackage; // Declare package

public class MyClass {
    public void displayMessage() {
        System.out.println("Hello from MyClass in
            MyPackage!");
    }
}

// File: Main.java
import MyPackage.MyClass; // Import custom package

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Create an object of
            MyClass
        obj.displayMessage(); // Output: Hello from MyClass
            in MyPackage!
    }
}
```

1.16.2 Library Management

- Libraries are collections of pre-written classes and methods for specific functionalities.
- Add external libraries to your project via your IDE or using tools like `Maven` or `Gradle`.
- Popular external libraries:
  - `Apache Commons`: General-purpose utility classes.
  - `Google Gson`: For converting Java objects to JSON and vice versa.
  - `JUnit`: For unit testing in Java.

Example: Adding and Using an External Library

```java
// Example using Gson library (add Gson dependency to your project)
import com.google.gson.Gson;

public class Main {
    public static void main(String[] args) {
        // Create a sample object
        Person person = new Person("Alice", 30);

        // Convert object to JSON
        Gson gson = new Gson();
        String json = gson.toJson(person);
        System.out.println(json); // Output: {"name":"Alice","age":30}

        // Convert JSON back to object
        Person deserialized = gson.fromJson(json, Person.class);
        System.out.println(deserialized.getName()); // Output: Alice
    }
}

class Person {
    private String name;
    private int age;

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

    // Getters
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

1.17 File Reading and Writing

1.17.1 Text File Operations

- Different access modes for text files:
  - "r": Read-only mode (file must exist).
  - "w": Write mode (creates file or truncates existing file).
  - "a": Append mode (adds content to the end of the file).
- Use `try-catch` blocks to handle file exceptions.

Example: Creating and Writing to a Text File:

```java
import java.io.*;

public class FileExample {
    public static void main(String[] args) {
        // Writing to a file
        try (BufferedWriter writer = new BufferedWriter(new
            FileWriter("example.txt"))) {
            writer.write("Hello, World!\n");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Appending to a file
        try (BufferedWriter writer = new BufferedWriter(new
            FileWriter("example.txt", true))) {
            writer.write("Appending new content.\n");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reading from a file
        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();
        }
    }
}
```

1.17.2 CSV File Operations

- Use `java.io` and `java.util` packages for CSV operations.
- For simpler handling, use libraries like `OpenCSV`.
- Common access modes for CSV files:
    - Writing rows: Create or overwrite existing file.
    - Reading rows: Process each line as an array of values.

Example: Writing to and Reading from a CSV File:

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

public class CSVExample {
    public static void main(String[] args) {
        String fileName = "example.csv";

        // Writing to a CSV file
        try (BufferedWriter writer = new BufferedWriter(new
            FileWriter(fileName))) {
            writer.write("Name,Age,City\n"); // Header row
            writer.write("Alice,25,New York\n");
            writer.write("Bob,30,Los Angeles\n");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reading from a CSV file
        try (BufferedReader reader = new BufferedReader(new
            FileReader(fileName))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] values = line.split(","); // Split
                    each line into values
                System.out.println(Arrays.toString(values));
                    // Print each row
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1.18 Exception Handling

1.18.1 Handling Exceptions in Java

- Exceptions are events that disrupt the normal flow of program execution.
- Java uses `try-catch-finally` blocks to handle exceptions gracefully.
- Exceptions are categorized into:
  - **Checked Exceptions:** Must be declared in the `throws` clause or handled using `try-catch`. Example: `IOException`.
  - **Unchecked Exceptions:** Occur at runtime and do not need to be declared or explicitly handled. Example: `ArithmeticException`, `NullPointerException`.
- Common exceptions include:
  - `FileNotFoundException`: File does not exist.
  - `ArithmeticException`: Division by zero or other arithmetic errors.
  - `NumberFormatException`: Invalid conversion of a string to a number.
  - `NullPointerException`: Accessing an object reference that is `null`.

Example: Handling `FileNotFoundException`

```
import java.io.*;

public class ExceptionExample {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new
               FileReader("nonexistent.txt"));
            String content = reader.readLine();
            reader.close();
        } catch (FileNotFoundException e) {
            // File does not exist
            System.out.println("File not found. Using
               default content.");
        } catch (IOException e) {
            System.out.println("Error reading the file.");
        }
    }
}
```

Example: Handling Division by Zero

```
public class DivisionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.
                Defaulting result to 0.");
            int result = 0;
        }
    }
}
```

1.18.2 Throwing Exceptions

- Use the `throw` keyword to explicitly raise an exception.
- Custom exceptions can be created by extending the `Exception` or `RuntimeException` classes.

Example: Throwing an Exception

```
public class ThrowExample {
    public static void validateAge(int age) throws
        IllegalArgumentException {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be
                18 or above.");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(16); // Throws exception
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Example: Custom Exceptions

```
class InvalidInputException extends Exception {
    public InvalidInputException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new InvalidInputException("Invalid input
                provided.");
        } catch (InvalidInputException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

1.18.3 Using `finally`

- The `finally` block always executes after `try-catch`, regardless of whether an exception occurs.
- Use it to release resources like file handles or database connections.

Example: Using `finally`

```java
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int num = Integer.parseInt("42"); // Conversion
                succeeds
            System.out.println("Number: " + num);
        } catch (NumberFormatException e) {
            System.out.println("Invalid format.");
        } finally {
            // Always executes
            System.out.println("Execution complete.");
        }
    }
}
```