Java Programming Language Documentation: Core Java 8

Introduction

Java, developed by Sun Microsystems (now owned by Oracle), is one of the most widely used programming languages in the world. It is a high-level, object-oriented, and platform-independent language known for its robustness, security, and scalability. This documentation aims to provide a comprehensive overview of Java, focusing specifically on **Core Java 8** concepts, including its history, fundamental language constructs, Object-Oriented Programming (OOP) principles, the revolutionary Stream API, robust Exception Handling, and powerful Multithreading capabilities.

I. A Brief History of Java

Java's journey began in the early 1990s as a project called "Oak" by James Gosling and his team at Sun Microsystems. Its initial goal was to create a language for consumer electronic devices.

- **1991:** Project "Oak" (later renamed Java) initiated by James Gosling.
- **1995:** Java 1.0 released. Its slogan "Write Once, Run Anywhere" (WORA) quickly gained popularity, signifying its platform independence due to the Java Virtual Machine (JVM). It leveraged applets for web interaction.
- **1997:** JDK 1.1 introduced Inner Classes, JavaBeans, JDBC (Java Database Connectivity), and RMI (Remote Method Invocation).
- **1998:** J2SE 1.2 (Java 2 Platform, Standard Edition) released, a major overhaul introducing the Collections Framework, Swing GUI toolkit, and the Java Plug-in for browsers.
- **2000:** J2SE 1.3 (Kestrel) – Minor enhancements, including HotSpot JVM.
- **2002:** J2SE 1.4 (Merlin) – Major additions like `assert` keyword, NIO (New I/O), regular expressions, and XML processing.
- **2004:** J2SE 5.0 (Tiger) – A significant release introducing Generics, Annotations, Autoboxing/Unboxing, Enums, enhanced `for` loop (foreach), and Varargs.
- **2006:** Java SE 6 (Mustang) – Scripting language support (JavaScript integration), improved web services, and JDBC 4.0.
- **2011:** Java SE 7 (Dolphin) – Minor language changes (Project Coin): `switch` on Strings, `try-with-resources`, diamond operator (`<>`), multi-catch, numeric literals with underscores.
- **2014: Java SE 8 (Pikachu)** – A monumental release that brought functional programming paradigms to Java with **Lambda Expressions**, the **Stream API**, default methods in interfaces, and the new Date and Time API (`java.time`). This version significantly changed how developers write concurrent and collection-processing code.
- **Post-Java 8:** Subsequent releases (Java 9, 10, 11, etc.) adopted a faster release cadence, introducing module system (Jigsaw), local-variable type inference (`var`), enhanced `switch` expressions, records, and sealed classes, among others.

This documentation focuses on **Core Java 8** as it represents a pivotal shift in the language's evolution and is still widely used in many production environments.

II. Core Java Language Concepts

At its heart, Java relies on a few fundamental concepts that enable its "Write Once, Run Anywhere" philosophy and facilitate robust application development.

A. JVM, JRE, and JDK

Understanding the distinctions between these three components is crucial for any Java developer.

1. **JVM (Java Virtual Machine):**
    - **Purpose:** The JVM is an abstract machine that provides a runtime environment in which Java bytecode can be executed. It's the core component that enables platform independence.
    - **How it Works:** When you compile a Java source file (`.java`), it's translated into bytecode (`.class`) by the Java compiler. The JVM then reads and executes this bytecode on any hardware platform for which a JVM implementation exists.
    - **Key Features:**
        - **Bytecode Interpreter:** Executes the bytecode instructions.
        - **Just-In-Time (JIT) Compiler:** Optimizes performance by compiling frequently executed bytecode into native machine code during runtime.
        - **Garbage Collector:** Automatically manages memory by reclaiming memory occupied by objects that are no longer referenced.
        - **Class Loader:** Loads `.class` files into memory.
2. **JRE (Java Runtime Environment):**
    - **Purpose:** The JRE is a software package that provides the minimum requirements for executing a Java application. If you only want to *run* Java applications, you only need the JRE.
    - **Components:** It includes the JVM, core libraries, and other supporting files necessary for running Java programs. It does **not** include development tools like compilers or debuggers.
3. **JDK (Java Development Kit):**
    - **Purpose:** The JDK is a complete software development kit for Java. It's designed for developers who want to *write* and *compile* Java applications.
    - **Components:** It includes everything in the JRE, plus development tools such as:
        - `javac` **(Java Compiler):** Compiles `.java` source files into `.class` bytecode files.
        - `java` **(Java Launcher):** Executes Java applications (invokes the JVM).
        - `jar` **(Archiver):** Creates and manages JAR (Java Archive) files.
        - `javadoc` **(Documentation Generator):** Generates HTML documentation from Javadoc comments.
        - **Debugging Tools:** (`jdb` for debugger, `jconsole` for monitoring, etc.).
    - **Analogy:** If you think of a car:
        - **JVM is the engine.**
        - **JRE is the car itself (with the engine, wheels, etc.) – ready to drive.**
        - **JDK is the entire factory with all the tools, machinery, and blueprints to build cars.**

B. Basic Syntax and Structure

1. **Comments:** Used to explain code, ignored by the compiler.

   - **Single-line:** `// This is a single-line comment`
   - **Multi-line:**

     ```
     /*
      * This is a multi-line comment.
      * It can span across several lines.
      */
     ```

   - **Documentation (Javadoc):**

     ```
     /**
      * This is a Javadoc comment.
      * Used to generate API documentation.
      * @param args Command-line arguments.
      */
     ```

2. **Identifiers:** Names given to classes, methods, variables, packages, etc.

   - Must begin with a letter, dollar sign (`$`), or underscore (`_`).
   - Subsequent characters can be letters, digits, `$` or `_`.
   - Case-sensitive (`myVar` is different from `myvar`).
   - Cannot be a Java keyword.

3. **Keywords (Reserved Words):** Words with predefined meanings in Java (e.g., `public`, `static`, `void`, `class`, `int`, `if`, `else`). Cannot be used as identifiers.

4. **Data Types:** Define the type of values a variable can hold.

   - **Primitive Data Types:** Store simple values directly in memory.
     - `byte` (1 byte, -128 to 127)
     - `short` (2 bytes)
     - `int` (4 bytes, default for integer numbers)
     - `long` (8 bytes)
     - `float` (4 bytes, single-precision floating-point)
     - `double` (8 bytes, double-precision floating-point, default for decimal numbers)
     - `boolean` (1 bit, `true` or `false`)
     - `char` (2 bytes, Unicode character)
   - **Non-Primitive (Reference) Data Types:** Store references (memory addresses) to objects.
     - `String`
     - Arrays
     - Classes
     - Interfaces

5. **Variables:** Named memory locations that store data.
   - **Declaration:** `dataType variableName;` (e.g., `int age;`)
   - **Initialization:** `dataType variableName = value;` (e.g., `int age = 30;`)
   - **Scope:**
     - **Instance variables:** Declared inside a class, outside any method/constructor/block. Belong to an object.
     - **Class (static) variables:** Declared with `static` keyword. Belong to the class, shared by all objects.
     - **Local variables:** Declared inside a method, constructor, or block. Only accessible within that scope.

6. **Operators:** Symbols that perform operations on variables and values.
   - **Arithmetic:** `+`, `-`, `*`, `/`, `%` (modulus)
   - **Relational:** `==` (equal to), `!=` (not equal to), `>`, `<`, `>=`, `<=`
   - **Logical:** `&&` (AND), `||` (OR), `!` (NOT)
   - **Assignment:** `=`, `+=`, `-=`, `*=` etc.
   - **Unary:** `++` (increment), `--` (decrement), `+` (unary plus), `-` (unary minus)
   - **Ternary:** `condition ? expr1 : expr2;`

7. **Control Flow Statements:** Control the order in which statements are executed.
   - **Conditional Statements:**
     - `if-else if-else`:

```
if (condition1) {
    // code if condition1 is true
} else if (condition2) {
    // code if condition2 is true
} else {
    // code if no condition is true
}
```

     - `switch`:

```
int day = 3;
String dayName;
switch (day) {
    case 1: dayName = "Monday"; break;
    case 2: dayName = "Tuesday"; break;
    default: dayName = "Invalid Day";
}
```

   - **Looping Statements:**
     - `for` **loop:**

```java
for (int i = 0; i < 5; i++) {
    System.out.println(i); // Prints 0 to 4
}
```

- **Enhanced `for` loop (for-each):**

```java
int[] numbers = {1, 2, 3};
for (int num : numbers) {
    System.out.println(num); // Prints 1, 2, 3
}
```

- **`while` loop:**

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

- **`do-while` loop:** (Executes at least once)

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

- **Branching Statements:**
  - **`break`:** Exits the current loop or `switch` statement.
  - **`continue`:** Skips the rest of the current iteration of a loop and proceeds to the next iteration.
  - **`return`:** Exits the current method and returns a value (if the method has a non-`void` return type).

III. Object-Oriented Programming (OOP) in Java

Java is fundamentally an object-oriented language. OOP is a programming paradigm based on the concept of "objects," which can contain data (fields/attributes) and code (methods/procedures). The main goal of OOP is to increase flexibility and maintainability.

A. Core OOP Concepts (Pillars)

1. **Encapsulation:**

- **Definition:** The bundling of data (attributes) and methods that operate on the data into a single unit (class). It also involves restricting direct access to some of an object's components, typically achieved by making fields `private` and providing `public` getter and setter methods to access and modify them.
- **Benefit:** Data hiding and protection. It allows control over how data is accessed and modified, preventing unintended side effects. Changes to internal implementation don't affect external code as long as the public interface remains the same.
- **Example:**

```java
class BankAccount {
    private double balance; // Encapsulated data

    public BankAccount(double initialBalance) {
        if (initialBalance >= 0) {
            this.balance = initialBalance;
        } else {
            this.balance = 0;
        }
    }

    public double getBalance() { // Public getter
        return balance;
    }

    public void deposit(double amount) { // Public setter/modifier
        if (amount > 0) {
            balance += amount;
        }
    }
    // ... withdraw method
}
```

2. **Inheritance:**
   - **Definition:** A mechanism where one class (subclass/child class) acquires the properties (fields and methods) of another class (superclass/parent class). It promotes code reusability and establishes an "is-a" relationship.
   - **Keyword:** `extends`
   - **Rules:**
     - A class can only inherit from one direct superclass (no multiple inheritance of classes in Java).
     - Constructors are not inherited.
     - `private` members are not directly accessible but can be accessed through `public`/`protected` methods of the superclass.

- **Example:**

  ```
  class Vehicle { // Superclass
      String brand;
      public void honk() {
          System.out.println("Tuut, tuut!");
      }
  }

  class Car extends Vehicle { // Subclass
      String model;
      public Car(String brand, String model) {
          this.brand = brand; // Inherited field
          this.model = model;
      }
      public void drive() {
          System.out.println(brand + " " + model + " is driving.");
      }
  }
  ```

3. **Polymorphism:**

   - **Definition:** Meaning "many forms." It allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent multiple underlying forms.

   - **Types:**

     - **Compile-time Polymorphism (Method Overloading):** Multiple methods in the *same class* have the same name but different parameters (different number or types of arguments). The compiler decides which method to call based on the arguments.

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

     - **Runtime Polymorphism (Method Overriding):** A subclass provides a specific implementation for a method that is already defined in its superclass. The decision of which method to call is made at runtime based on the actual object type.

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

       class Dog extends Animal {
           @Override // Annotation indicating override
       ```

```java
    public void makeSound() { System.out.println("Dog barks"); }
}

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

// Usage:
Animal myAnimal = new Dog();
myAnimal.makeSound(); // Outputs: Dog barks (Runtime decision)
myAnimal = new Cat();
myAnimal.makeSound(); // Outputs: Cat meows
```

4. **Abstraction:**

   - **Definition:** The process of hiding the implementation details and showing only the essential features of an object. It focuses on "what" an object does rather than "how" it does it. Achieved using abstract classes and interfaces.
   - **Abstract Class:**
     - Can have `abstract` methods (methods without a body, must be implemented by subclasses) and concrete methods.
     - Cannot be instantiated directly.
     - Declared using the `abstract` keyword.
     - A subclass of an abstract class must implement all its abstract methods or be declared `abstract` itself.
     - **Example:**

```java
abstract class Shape {
    String color;
    abstract double getArea(); // Abstract method
    public void displayColor() {
        System.out.println("Color: " + color);
    }
}

class Circle extends Shape {
    double radius;
    public Circle(String color, double radius) {
        this.color = color;
        this.radius = radius;
    }
    @Override
    double getArea() { return Math.PI * radius * radius; }
}
```

- **Interface:**
  - A blueprint of a class. It can contain method signatures (abstract methods) and constants.
  - Prior to Java 8, all methods in an interface were implicitly `public abstract`.
  - **Java 8 onwards:** Interfaces can now have `default` methods (with implementation) and `static` methods. This was a significant addition for the Stream API and Lambdas.
  - A class can `implement` multiple interfaces (achieving a form of multiple inheritance of behavior).
  - **Example:**

```java
interface Playable {
    void play(); // Abstract method

    // Java 8: default method
    default void stop() {
        System.out.println("Stopped playing.");
    }

    // Java 8: static method
    static void greet() {
        System.out.println("Hello from Playable interface!");
    }
}

class MusicPlayer implements Playable {
    @Override
    public void play() {
        System.out.println("Playing music...");
    }
}
```

B. Classes and Objects

- **Class:** A blueprint or a template for creating objects. It defines the structure (fields) and behavior (methods) that objects of that class will have.

```java
class Dog {
    // Fields (attributes)
    String name;
    String breed;
    int age;

    // Methods (behaviors)
    public void bark() {
        System.out.println(name + " barks!");
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }
}
```

- **Object:** An instance of a class. When a class is defined, no memory is allocated. Memory is allocated only when an object is created.

```
// Creating an object (instance) of the Dog class
Dog myDog = new Dog(); // 'new Dog()' allocates memory and calls the constructor

// Accessing fields and methods
myDog.name = "Buddy";
myDog.breed = "Golden Retriever";
myDog.age = 5;

myDog.bark(); // Output: Buddy barks!
```

C. Constructors

- **Definition:** A special method used to initialize objects. It has the same name as the class and does not have a return type (not even `void`).
- **Types:**
    - **Default Constructor:** If you don't define any constructor, Java provides a no-argument default constructor.
    - **No-argument Constructor:**

    ```
    class Person {
        String name;
        public Person() { // No-argument constructor
            this.name = "Unknown";
        }
    }
    ```

    - **Parameterized Constructor:**

    ```
    class Person {
        String name;
        int age;
        public Person(String name, int age) { // Parameterized constructor
            this.name = name;
            this.age = age;
        }
    ```

```
}
// Usage: Person p1 = new Person("Alice", 30);
```

- **Constructor Overloading:** A class can have multiple constructors with different parameter lists.

D. `this` Keyword

- `this` is a reference to the current object.
- **Usage:**
  - To differentiate between instance variables and local variables (especially in constructors and setters).
  - To call another constructor from the current constructor (constructor chaining: `this(...)`).
  - To return the current class instance.

```java
class Box {
    int width;
    int height;

    public Box(int width, int height) {
        this.width = width;  // 'this.width' refers to instance variable
        this.height = height; // 'height' refers to local parameter
    }

    public Box(int side) {
        this(side, side); // Calls the two-argument constructor (constructor chaining)
    }

    public Box setWidth(int width) {
        this.width = width;
        return this; // Returns the current object for method chaining
    }
}
```

E. `static` Keyword

- The `static` keyword can be applied to fields, methods, nested classes, and initializer blocks.
- **Static Members (Class Members):** Belong to the class itself, not to any specific object instance.
- **Static Fields (Class Variables):**
  - Only one copy exists, shared by all objects of the class.
  - Accessed using the class name (`ClassName.staticField`).
  - **Example:** `public static final double PI = 3.14159;`
- **Static Methods (Class Methods):**
  - Can be called directly on the class without creating an object.
  - Cannot access non-static (instance) members directly, as they don't operate on a specific object instance.

  - Can only call other static methods.
  - **Example:** `public static int max(int a, int b) { return a > b ? a : b; }`
- **`main` method:** `public static void main(String[] args)` is static so that the JVM can call it without creating an object of the class.

F. Packages

- **Definition:** A mechanism to organize classes, interfaces, and sub-packages into a logical hierarchy. It helps in preventing naming conflicts and provides better management of large projects.
- **Declaration:** `package com.example.myapp;` (must be the first statement in a Java source file).
- **Naming Convention:** Lowercase, reverse domain name (e.g., `com.mycompany.project.module`).
- **Importing:** Use the `import` keyword to use classes from other packages.
  - `import java.util.ArrayList;` (imports a specific class)
  - `import java.util.*;` (imports all classes in the `java.util` package)
- **Default Package:** If no `package` statement is present, classes are considered part of the "default package."

G. Access Modifiers

Control the visibility (accessibility) of classes, fields, methods, and constructors.

Modifier	Class	Package	Subclass	World	Description
`private`	Yes	No	No	No	Accessible only within the declaring class.
`default`	Yes	Yes	No	No	Accessible only within the same package. (No keyword, just omit others)
`protected`	Yes	Yes	Yes	No	Accessible within the same package and by subclasses (even in different packages).
`public`	Yes	Yes	Yes	Yes	Accessible from anywhere.

IV. Advanced Core Java Concepts

A. Exception Handling

Exceptions are events that disrupt the normal flow of a program's instructions. They are objects that wrap information about an error condition. Exception handling is a mechanism to gracefully manage these errors, preventing the program from crashing.

1. **What is an Exception?**
   - An unexpected event that occurs during the execution of a program, leading to its termination if not handled.
   - Examples: `ArithmeticException` (division by zero), `NullPointerException` (accessing a method/field on a `null` object), `ArrayIndexOutOfBoundsException`, `FileNotFoundException`, `IOException`.

2. **`Throwable` Class Hierarchy:**

- All exceptions and errors in Java are subclasses of the `java.lang.Throwable` class.
- **`Error`:** Represents serious problems that applications should not try to catch. Examples: `OutOfMemoryError`, `StackOverflowError`. These usually indicate unrecoverable conditions.
- **`Exception`:** Represents conditions that a reasonable application might want to catch.
  - **`Checked Exceptions`:** Subclasses of `Exception` (but not `RuntimeException`). The compiler forces you to handle them (either by `try-catch` or by `throws` declaration). Examples: `IOException`, `SQLException`, `ClassNotFoundException`.
  - **`Unchecked Exceptions` (Runtime Exceptions):** Subclasses of `RuntimeException`. The compiler does *not* force you to handle them. They typically indicate programming errors. Examples: `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`.

3. **Keywords for Exception Handling:**

   - **`try`:**
     - A block of code that is monitored for exceptions.
     - If an exception occurs within the `try` block, it is "thrown."
   - **`catch`:**
     - Follows a `try` block. It defines the type of exception it can handle.
     - If the `try` block throws an exception of the specified type, the `catch` block is executed.
     - A `try` block can have multiple `catch` blocks for different exception types.
     - **Multi-catch (Java 7+):**

       ```
       try {
           // ... code that might throw IOException or SQLException
       } catch (IOException | SQLException e) {
           System.err.println("An I/O or SQL error occurred: " +
       e.getMessage());
       }
       ```
   - **`finally`:**
     - An optional block that always executes, regardless of whether an exception occurred in the `try` block or was caught by a `catch` block.
     - Useful for cleanup code (e.g., closing resources like files or database connections).
   - **Example (`try-catch-finally`):**

     ```
     public class ExceptionDemo {
         public static void main(String[] args) {
             try {
                 int result = 10 / 0; // Throws ArithmeticException
                 System.out.println("Result: " + result); // This line is
     skipped
             } catch (ArithmeticException e) {
                 System.err.println("Error: Cannot divide by zero.");
     ```

```
            e.printStackTrace(); // Prints the stack trace for
debugging
        } catch (Exception e) { // Generic catch for any other
exception
            System.err.println("An unexpected error occurred: " +
e.getMessage());
        } finally {
            System.out.println("This finally block always executes.");
        }
        System.out.println("Program continues after exception
handling.");
    }
}
```

- `throw`:
  - Used to explicitly throw an exception object.
  - Can throw either a newly created exception (`throw new IllegalArgumentException("Invalid input");`) or a caught exception.
- `throws`:
  - Used in a method signature to declare that a method *might* throw one or more specified checked exceptions.
  - It doesn't handle the exception; it delegates the responsibility of handling to the caller of the method.
- **Example (`throw` and `throws`):**

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

public class FileProcessor {

    public void readFile(String filePath) throws IOException { //
Declares it might throw IOException
        FileReader reader = null;
        try {
            reader = new FileReader(filePath);
            int data = reader.read();
            // Simulate an error condition
            if (data == -1) {
                throw new IOException("File is empty or could not be
read."); // Throws a new exception
            }
            System.out.println("First char: " + (char)data);
        } finally {
            if (reader != null) {
                reader.close(); // Important: Close the resource
            }
        }
    }

    public static void main(String[] args) {
        FileProcessor processor = new FileProcessor();
        try {
            processor.readFile("nonexistent.txt");
        } catch (IOException e) {
            System.err.println("Caught an IO Exception: " +
e.getMessage());
        }
    }
}
```

4. `try-with-resources` **(Java 7+):**

    - A special `try` statement that automatically closes resources (like files, network connections, etc.) that implement the `AutoCloseable` interface (or `Closeable`).
    - Eliminates the need for explicit `finally` blocks for resource closing, making code cleaner and less error-prone.

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

public class AutoCloseableDemo {
    public static void main(String[] args) {
        String filePath = "data.txt"; // Ensure this file exists for demo
        try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
        // reader is automatically closed here, no need for finally
    }
}
```

5. **Custom Exceptions:**

    - You can create your own exception classes by extending `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).
    - Useful for representing specific error conditions within your application's domain.

```
class InsufficientFundsException extends Exception { // Checked exception
    public InsufficientFundsException(String message) {
        super(message);
```

```java
    }
}

class BankAccount {
    private double balance;
    public BankAccount(double balance) { this.balance = balance; }

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient funds! 
Available: " + balance + ", Requested: " + amount);
        }
        balance -= amount;
        System.out.println("Withdrew " + amount + ". New balance: " + 
balance);
    }
}

public class CustomExceptionDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(100.0);
        try {
            account.withdraw(150.0);
        } catch (InsufficientFundsException e) {
            System.err.println("Transaction failed: " + e.getMessage());
        }
    }
}
```

B. Multithreading & Concurrency

Multithreading allows a program to execute multiple parts of its code concurrently, improving resource utilization and responsiveness.

1. **Process vs. Thread:**
    - **Process:** An independent execution unit that has its own memory space, open files, and other resources. Running multiple applications on your computer means running multiple processes.
    - **Thread:** A lightweight sub-process within a process. Threads within the same process share the same memory space and resources, making inter-thread communication more efficient than inter-process communication.
2. **Creating Threads:** In Java, you can create threads in two primary ways:
    - **a. Extending the `Thread` Class:**
        - Create a class that `extends java.lang.Thread`.
        - Override the `run()` method with the code that the thread will execute.
        - Create an instance of your class and call its `start()` method.

```java
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + ": "
+ i);
            try {
                Thread.sleep(100); // Pause for 100 milliseconds
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + "
interrupted.");
            }
        }
    }
}

public class ThreadExtensionDemo {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        thread1.setName("Thread-A");
        MyThread thread2 = new MyThread();
        thread2.setName("Thread-B");

        thread1.start(); // Starts execution of run() in a new thread
        thread2.start();
    }
}
```

- **b. Implementing the `Runnable` Interface:**
  - Create a class that `implements java.lang.Runnable`.
  - Override the `run()` method.
  - Create an instance of your `Runnable` implementation.
  - Pass this instance to a `Thread` constructor and then call `start()` on the `Thread` object.
  - **Advantage:** This is generally preferred as Java does not support multiple inheritance of classes, but it supports multiple interface implementations. This separates the task (Runnable) from the thread execution mechanism (Thread).

```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + ": "
+ i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + "
interrupted.");
            }
        }
    }
}

public class RunnableImplementationDemo {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable(), "Thread-X");
        Thread thread2 = new Thread(new MyRunnable(), "Thread-Y");

        thread1.start();
        thread2.start();
    }
}
```

3. **Thread Lifecycle:** A thread goes through various states:
    - **New:** Thread has been created but not yet started.
    - **Runnable:** Thread is ready to run (eligible to be picked up by the scheduler).
    - **Running:** Thread is currently executing.
    - **Blocked/Waiting:** Thread is temporarily inactive, waiting for a resource or condition (e.g., waiting for I/O, `sleep()`, `wait()`).
    - **Terminated:** Thread has finished its execution (`run()` method completed).

4. **Synchronization:**
    - **Problem (Race Condition):** When multiple threads try to access and modify the same shared resource concurrently, the final outcome can be unpredictable and incorrect due to interleaving of operations.

    - **Solution:** Synchronization ensures that only one thread can access a shared resource at a time, preventing race conditions.

    - `synchronized` **Keyword:** Can be used on methods or blocks.
        - **Synchronized Method:** When a thread invokes a synchronized method, it acquires the monitor (lock) for that object. Other threads trying to invoke any synchronized method on the *same object* will be blocked until the first thread releases the monitor.

```
class Counter {
    private int count = 0;
    public synchronized void increment() { // Synchronized method
        count++;
    }
    public int getCount() { return count; }
}
```

        - **Synchronized Block:** Provides finer-grained control. It acquires the lock on the `object` specified in parentheses.

```
class Counter {
    private int count = 0;
    private final Object lock = new Object(); // Object to lock on

    public void increment() {
        synchronized (lock) { // Synchronized block
            count++;
        }
    }
    public int getCount() { return count; }
}
```

- **Static Synchronization:** If a `static` method is synchronized, it acquires the lock on the `Class` object itself, not an instance. This affects all threads trying to access *any* synchronized static method of that class.

5. **Inter-thread Communication (`wait()`, `notify()`, `notifyAll()`):**
    - Methods of the `Object` class, used to coordinate activities between threads.
    - **Important:** Must be called from within a synchronized block or method, and the calling thread must own the monitor of the object on which `wait()` or `notify()` is called.
    - **`wait()`:** Causes the current thread to wait until another thread invokes `notify()` or `notifyAll()` for this object. The thread releases its lock on the object and goes into a waiting state.
    - **`notify()`:** Wakes up a single thread that is waiting on this object's monitor.
    - **`notifyAll()`:** Wakes up all threads that are waiting on this object's monitor.
    - **Example (Producer-Consumer):**

```
class MessageQueue {
    private String message;
    private boolean isEmpty = true;

    public synchronized void put(String message) {
        while (!isEmpty) {
            try { wait(); } catch (InterruptedException e) {}
        }
        this.message = message;
        isEmpty = false;
        notifyAll(); // Notify consumer
    }

    public synchronized String take() {
        while (isEmpty) {
            try { wait(); } catch (InterruptedException e) {}
        }
        isEmpty = true;
```

```
            notifyAll(); // Notify producer
            return message;
        }
    }

    class Producer implements Runnable { /* ... uses put() ... */ }
    class Consumer implements Runnable { /* ... uses take() ... */ }
```

6. **Deadlock (Brief Mention):**

    - A situation where two or more threads are blocked indefinitely, waiting for each other to release the resources that they need.
    - Occurs when threads hold locks on resources and simultaneously attempt to acquire locks on resources held by other threads.

V. Java 8 Specific Features: Stream API & Lambda Expressions

Java 8 introduced significant features that changed how we write code, especially for collection processing and concurrency. These features embrace a functional programming style.

A. Lambda Expressions

1. **What are Lambdas?**

    - Lambda expressions provide a concise way to represent an anonymous function (a function without a name).
    - They are primarily used to implement **functional interfaces** (interfaces with a single abstract method).
    - They enable functional programming in Java by allowing functions to be treated as arguments to methods or to be stored in variables.

2. **Syntax:** `(parameters) -> expression` or `(parameters) -> { statements; }`

    - **No Parameters:** `() -> System.out.println("Hello World")`
    - **Single Parameter (no type needed, no parentheses if just one):** `s -> s.length()`
    - **Multiple Parameters:** `(a, b) -> a + b`
    - **Block of Code:** `(x, y) -> { System.out.println("Adding"); return x + y; }`

3. **Functional Interfaces:**

    - An interface with exactly one abstract method.

    - Can be annotated with `@FunctionalInterface` (optional, but good practice for clarity and compiler checks).

    - Java 8 introduced several built-in functional interfaces in `java.util.function`:

        - `Predicate<T>`: `boolean test(T t)` (takes T, returns boolean)
        - `Consumer<T>`: `void accept(T t)` (takes T, returns nothing)
        - `Function<T, R>`: `R apply(T t)` (takes T, returns R)

        - `Supplier<T>`: `T get()` (takes nothing, returns T)
        - `UnaryOperator<T>`: `T apply(T t)` (takes T, returns T, extends `Function`)
        - `BinaryOperator<T>`: `T apply(T t1, T t2)` (takes two Ts, returns T, extends `BiFunction`)

    - **Example (Pre-Java 8 Anonymous Class vs. Lambda):**

```java
// Old way (Anonymous inner class)
Runnable oldRunnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running from anonymous class.");
    }
};

// New way (Lambda Expression)
Runnable newRunnable = () -> System.out.println("Running from lambda.");

new Thread(oldRunnable).start();
new Thread(newRunnable).start();

// Using a functional interface directly
GreetingService greet = message -> System.out.println("Hello " + message);
greet.sayMessage("World");

@FunctionalInterface
interface GreetingService {
    void sayMessage(String message);
}
```

4. **Method References:**

    - A concise way to refer to methods or constructors.
    - Syntax: `ClassName::methodName`, `objectName::methodName`, `ClassName::new` (for constructors).
    - Can be used when a lambda expression just calls an existing method.
    - **Example:**

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Lambda:
names.forEach(name -> System.out.println(name));

// Method reference (equivalent):
names.forEach(System.out::println); // Static method reference
```

```
// Another example:
List<Integer> numbers = Arrays.asList(5, 1, 3, 2, 4);
// Lambda:
Collections.sort(numbers, (a, b) -> Integer.compare(a, b));
// Method reference:
Collections.sort(numbers, Integer::compare); // Static method reference

// Instance method reference
MyPrinter printer = new MyPrinter();
names.forEach(printer::print); // printer.print(name) for each name

// Constructor reference
Supplier<List<String>> listSupplier = ArrayList::new; // Calls new
ArrayList<>()
List<String> newList = listSupplier.get();
```

B. Stream API

The Stream API is a powerful feature in Java 8 for processing collections of objects. It provides a declarative, functional approach to data processing, allowing you to express what you want to do with the data rather than how to do it.

1. **What is a Stream?**
    - A sequence of elements supporting sequential and parallel aggregate operations.
    - **Not a Data Structure:** It doesn't store data. It's a view or a pipeline for processing data from a source.
    - **Functional in Nature:** Operations on streams are designed to be non-interfering and stateless. They don't modify the source.
    - **Lazy Evaluation:** Intermediate operations are not executed until a terminal operation is invoked.
    - **Can be traversed only once:** Once a terminal operation is performed, the stream is "consumed" and cannot be reused.
2. **Stream Pipeline Structure:** A typical stream pipeline consists of three parts:
    - **a. Source:**
        - Can be a `Collection` (`list.stream()`, `set.stream()`), an `Array` (`Arrays.stream(array)`), `I/O channels`, `generate()`, `iterate()`, etc.

    ```
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
    Stream<String> nameStream = names.stream(); // From a List
    ```

    - **b. Zero or More Intermediate Operations:**
        - Transform a stream into another stream. They are *lazy*; they don't perform any actual processing until a terminal operation is invoked.
        - Examples: `filter()`, `map()`, `distinct()`, `sorted()`, `peek()`, `limit()`, `skip()`.

    - They return a `Stream` object, allowing for method chaining.

    ```
    // Example chain of intermediate operations
    names.stream()
         .filter(name -> name.startsWith("A")) // Keep names starting with
    'A'
         .map(String::toUpperCase)           // Convert to uppercase
         .distinct();                        // Ensure unique elements
    ```

  - **c. One Terminal Operation:**

    - Produces a result or a side-effect. It triggers the execution of all preceding intermediate operations.
    - After a terminal operation, the stream cannot be reused.
    - Examples: `forEach()`, `collect()`, `reduce()`, `count()`, `min()`, `max()`, `findFirst()`, `findAny()`, `allMatch()`, `anyMatch()`, `noneMatch()`.

    ```
    // Example of a terminal operation (forEach)
    names.stream()
         .filter(name -> name.startsWith("A"))
         .map(String::toUpperCase)
         .forEach(System.out::println); // ALICE
    ```

3. **Common Stream Operations in Detail:**

  - **`filter(Predicate<T> predicate)`:**

    - Selects elements that match a given condition.
    - Returns a new stream containing only the elements for which the predicate returns `true`.

    ```
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    numbers.stream()
           .filter(n -> n % 2 == 0) // Keep only even numbers
           .forEach(System.out::println); // 2, 4, 6, 8, 10
    ```

  - **`map(Function<T, R> mapper)`:**

    - Transforms each element of the stream into a new element (of potentially a different type).
    - Applies the function to each element and returns a stream of the results.

    ```
    List<String> words = Arrays.asList("hello", "world");
    words.stream()
         .map(String::toUpperCase) // Converts to HELLO, WORLD
         .forEach(System.out::println);

    words.stream()
    ```

```
        .map(String::length) // Converts to lengths: 5, 5
        .forEach(System.out::println);
```

- `flatMap(Function<T, Stream<R>> mapper)`:
    - Transforms each element into a stream of zero or more elements, and then flattens the resulting streams into a single stream.
    - Useful for processing nested collections.

```
List<List<String>> listOfLists = Arrays.asList(
    Arrays.asList("a", "b"),
    Arrays.asList("c", "d", "e")
);
listOfLists.stream()
           .flatMap(Collection::stream) // Flattens to a single stream of strings
           .forEach(System.out::println); // a, b, c, d, e
```

- `distinct()`:
    - Returns a stream consisting of the distinct elements (based on `equals()`) of this stream.

```
List<Integer> numsWithDuplicates = Arrays.asList(1, 2, 2, 3, 1, 4);
numsWithDuplicates.stream()
                  .distinct()
                  .forEach(System.out::println); // 1, 2, 3, 4
```

- `sorted()` / `sorted(Comparator<T> comparator)`:
    - Returns a stream consisting of the elements of this stream, sorted according to natural order or a custom comparator.

```
List<String> unsorted = Arrays.asList("zebra", "apple", "cat");
unsorted.stream().sorted().forEach(System.out::println); // apple, cat, zebra
unsorted.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println); // zebra, cat, apple
```

- `limit(long maxSize)`:
    - Truncates the stream to be no longer than `maxSize` in length.

```
numbers.stream().limit(5).forEach(System.out::println); // 1, 2, 3, 4, 5
```

- `skip(long n)`:
  - Discards the first `n` elements of the stream.

  ```
  numbers.stream().skip(5).forEach(System.out::println); // 6, 7, 8, 9,
  10
  ```

- `forEach(Consumer<T> action)`:
  - Performs an action for each element in the stream. A terminal operation.

  ```
  names.stream().forEach(name -> System.out.println("Hello " + name));
  ```

- `collect(Collector<T, A, R> collector)`:
  - Performs a mutable reduction operation on the elements of this stream, accumulating them into a mutable result container.
  - `Collectors` utility class provides many common collectors.
  - **Common uses:**
    - `Collectors.toList()`: Collects elements into a `List`.
    - `Collectors.toSet()`: Collects elements into a `Set`.
    - `Collectors.toMap()`: Collects elements into a `Map`.
    - `Collectors.joining()`: Concatenates elements of a `CharSequence` stream.
    - `Collectors.groupingBy()`: Groups elements based on a classification function.
    - `Collectors.counting()`, `summingInt()`, `averagingDouble()`, etc.

  ```
  List<String> upperCaseNames = names.stream()
                                     .map(String::toUpperCase)
                                     .collect(Collectors.toList());
  System.out.println(upperCaseNames); // [ALICE, BOB, CHARLIE, DAVID]

  Map<Integer, List<String>> namesByLength = names.stream()
      .collect(Collectors.groupingBy(String::length));
  System.out.println(namesByLength); // {5=[Alice, David], 3=[Bob], 7=
  [Charlie]}
  ```

- `reduce(initialValue, BinaryOperator<T> accumulator)` / `reduce(BinaryOperator<T> accumulator)`:
  - Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an `Optional` describing the reduced value.
  - Combines elements into a single result.

  ```
  List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
  int sum = intList.stream().reduce(0, (a, b) -> a + b); // initialValue
  ```

```
0
System.out.println(sum); // 15

Optional<Integer> product = intList.stream().reduce((a, b) -> a * b);
// No initial value, returns Optional
product.ifPresent(System.out::println); // 120

String combined = names.stream().reduce("", (s1, s2) -> s1 + "-" + s2);
// ""-Alice-Bob-Charlie-David
String combinedCorrect = names.stream().collect(Collectors.joining("-
")); // Alice-Bob-Charlie-David
```

- `count()`:
  - Returns the count of elements in the stream.

```
long count = names.stream().filter(name -> name.length() > 3).count();
// 4
```

- `min(Comparator<T> comparator)` / `max(Comparator<T> comparator)`:
  - Returns an `Optional` describing the minimum/maximum element of this stream according to the provided `Comparator`.

```
Optional<String> longestName = names.stream()

.max(Comparator.comparingInt(String::length));
longestName.ifPresent(System.out::println); // Charlie
```

- `findFirst()` / `findAny()`:
  - Returns an `Optional` describing the first element of this stream, or any element. Useful for short-circuiting.
  - `findAny()` is more performant in parallel streams.

```
Optional<String> first = names.stream().findFirst(); // Alice
Optional<String> any = names.stream().filter(n ->
n.contains("o")).findAny(); // Bob (could be any name with 'o' in
parallel stream)
```

- `anyMatch(Predicate<T> predicate)` / `allMatch(Predicate<T> predicate)` / `noneMatch(Predicate<T> predicate)`:
  - Return a `boolean` indicating if any, all, or none of the elements match the predicate. These are short-circuiting terminal operations.

```java
boolean hasBob = names.stream().anyMatch(name -> name.equals("Bob"));
// true
boolean allLong = names.stream().allMatch(name -> name.length() > 2);
// true
boolean noZ = names.stream().noneMatch(name -> name.contains("Z")); //
true
```

4. **Parallel Streams:**
    - To process data in parallel, simply call `parallelStream()` on a collection instead of `stream()`.
    - Java's Fork/Join framework handles the partitioning of data and parallel execution automatically.
    - **Caution:** Not all operations benefit from parallel streams, and some might even perform worse due to overhead of managing threads and combining results. Side-effects should be avoided in parallel streams.

```java
List<Integer> bigList = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); //
Imagine a much larger list
long sumParallel = bigList.parallelStream()
                          .filter(n -> n % 2 == 0)
                          .mapToInt(Integer::intValue)
                          .sum();
System.out.println("Parallel Sum of Evens: " + sumParallel);
```

Conclusion

This documentation has provided a detailed journey through Core Java 8, covering its historical evolution, fundamental language constructs, the pillars of Object-Oriented Programming, essential Exception Handling mechanisms, the power of Multithreading, and the transformative features introduced in Java 8: Lambda Expressions and the Stream API.

Java 8 marked a significant shift towards a more functional programming style, making code more concise, readable, and efficient for data processing, especially with large collections. Understanding these core concepts thoroughly is the bedrock for building robust, scalable, and maintainable Java applications.