WEEK-2

A First Taste of Java

Why is it so complicated?

Java is designed to be platform-independent, object-oriented, and secure, which can introduce some complexity. The syntax is strict, and it enforces strong typing, making it robust but sometimes verbose.

Compiling and running Java code

To compile and run Java code:

1. Save your Java program in a file with a `.java` extension.
2. Use the `javac` command to compile the file:
   `javac MyProgram.java`
3. This will generate a `MyProgram.class` file containing bytecode.
4. Use the `java` command to run the compiled bytecode:
   `java MyProgram`

Basic Data Types in Java

Scalar Types

Java has several scalar types including:

- `int` (integer)
- `double` (double-precision floating point)
- `char` (character)
- `boolean` (true or false)

Example:

```
int number = 10;
```

```
double price = 9.99;
char letter = 'A';
boolean isJavaFun = true;
```

Declarations, assigning values

Declarations specify the type and name of a variable, while assigning values initializes the variable.

Example:

```
int age; // Declaration
age = 25; // Assignment
```

Initialization, constants

Initialization assigns a value at the time of declaration. Constants use the `final` keyword and cannot be changed once initialized.

Example:

```
int year = 2024; // Initialization
final double PI = 3.14159; // Constant
```

Operators, shortcuts, type casting

Java supports various operators such as arithmetic (`+`, `-`, `*`, `/`), assignment (`=`, `+=`, `-=`, etc.), and comparison (`==`, `!=`, `<`, `>`).

Shortcuts:

```
int count = 10;
count += 5; // Equivalent to count = count + 5;
count++; // Increment by 1
```

Type casting:

```
double a = 9.78;
```

```
int b = (int) a; // Explicit type casting from double to int
```

Strings

Strings in Java are objects that represent sequences of characters. They are immutable.

Example:

```
String greeting = "Hello, World!";
```

Arrays

Arrays store multiple values of the same type in a single variable.

Example:

```
int[] numbers = {1, 2, 3, 4, 5};
String[] names = {"Alice", "Bob", "Charlie"};
```

Basic Input and Output in Java

Reading input

Using Console

Using the `Console` class to read input:

```
Console console = System.console();
if (console != null) {
    String name = console.readLine("Enter your name: ");
    System.out.println("Hello, " + name);
}
```

Using Scanner

The `Scanner` class is more commonly used for reading input:

```
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = scanner.nextInt();
System.out.println("You are " + age + " years old.");
```

Generating Output

Output is typically generated using `System.out.print` or `System.out.println`:

```
System.out.println("Hello, World!"); // Prints with a newline
System.out.print("Hello, "); // Prints without a newline
System.out.print("World!"); // Continues on the same line
```

Control Flow in Java

Control flow

Control flow statements determine the order in which instructions are executed. This includes conditionals and loops.

Conditional Execution

Conditional execution uses `if`, `else if`, and `else` statements.

```
int score = 85;

if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else {
    System.out.println("Grade: C");
}
```

Conditional Loops

Conditional loops use `while` and `do-while`.

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

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

Iteration

Iteration typically uses `for` loops.

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

Iterating over elements directly

Enhanced for loop for arrays and collections:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int number : numbers) {
    System.out.println(number);
}
```

Multiway branching

Using `switch` for multiway branching:

```
int day = 3;
switch (day) {
```

```
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Other day");
        break;
}
```

Defining Classes and Objects in Java

Defining a class

A class is a blueprint for objects. It defines attributes and methods.

```
public class Dog {
    // Attributes
    String name;
    int age;

    // Methods
    void bark() {
        System.out.println("Woof!");
    }
}
```

Creating Object

Creating an object from a class:

```
Dog myDog = new Dog();
```

```
myDog.name = "Buddy";
myDog.age = 3;
myDog.bark(); // Outputs: Woof!
```

Constructors

Constructors initialize new objects. They have the same name as the class.

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

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

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

// Creating an object with a constructor
Dog myDog = new Dog("Buddy", 3);
```

Copy constructors

A copy constructor creates a new object as a copy of an existing object.

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

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
```

```
    }

    // Copy constructor
    public Dog(Dog other) {
        this.name = other.name;
        this.age = other.age;
    }
}

// Using the copy constructor
Dog originalDog = new Dog("Buddy", 3);
Dog copiedDog = new Dog(originalDog);
```

WEEK-3

Dynamic Dispatch and Polymorphism

Code

Dynamic dispatch and polymorphism are central to object-oriented programming in Java. Here's a basic example illustrating these concepts:

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

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark");
```

```
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow");
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Bark

        myAnimal = new Cat(); // Upcasting
        myAnimal.makeSound(); // Meow
    }
}
```

Dynamic Dispatch

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at runtime. The above example demonstrates dynamic dispatch: the call to `makeSound` is determined at runtime based on the actual object type.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It is mainly achieved through inheritance and interfaces.

Functions, Signatures, and Overloading

Function overloading in Java allows multiple methods with the same name but different parameter lists.

```
class MathUtils {
    static int add(int a, int b) {
        return a + b;
    }
```

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

public class TestOverloading {
    public static void main(String[] args) {
        System.out.println(MathUtils.add(5, 10)); // Calls int version
        System.out.println(MathUtils.add(5.5, 10.5)); // Calls double
version
    }
}
```

Type Casting

Type casting is converting an object of one type to another. It can be implicit or explicit.

```
Animal myAnimal = new Dog(); // Upcasting (implicit)
Dog myDog = (Dog) myAnimal; // Downcasting (explicit)
```

Java Modifiers

`public` vs `private`

- `public`: The member is accessible from any other class.
- `private`: The member is accessible only within its own class.

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
```

```
    }
}
```

`static` Components

`static` components belong to the class rather than any instance of the class.

```
class Counter {
    public static int count = 0;

    public Counter() {
        count++;
    }
}

public class TestStatic {
    public static void main(String[] args) {
        new Counter();
        new Counter();
        System.out.println(Counter.count); // Output: 2
    }
}
```

`final` Components

`final` keyword can be used with variables, methods, and classes to restrict modification.

- `final` variable: cannot be reassigned.
- `final` method: cannot be overridden.
- `final` class: cannot be subclassed.

```
final class ImmutableClass {
    final int MAX_VALUE = 100;

    final void display() {
        System.out.println("This is a final method.");
    }
}
```

Subclasses and Inheritance

A Java Class

A Java class serves as a blueprint for objects.

```
public class Animal {
    String name;
    int age;

    void eat() {
        System.out.println("Eating...");
    }
}
```

Subclasses

Subclasses inherit fields and methods from a superclass and can add their own.

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

Inheritance

Inheritance allows a class to inherit fields and methods from another class.

```
public class Cat extends Animal {
    void meow() {
        System.out.println("Meow");
    }
}
```

Summary

Inheritance promotes code reusability and establishes a relationship between parent and child classes.

Subtyping vs Inheritance

Subclasses, Subtyping, and Inheritance

- Subclassing: Creating a new class based on an existing class.
- Subtyping: A form of polymorphism where a subclass is considered to be a subtype of its superclass.

Subtyping vs Inheritance

Inheritance is about code reuse, while subtyping is about type compatibility. A subclass inherits methods and fields, while subtyping ensures that an instance of the subclass can be used wherever an instance of the superclass is expected.

The Java Class Hierarchy

Multiple Inheritance

Java does not support multiple inheritance with classes but allows it through interfaces.

```
interface CanFly {
    void fly();
}

interface CanSwim {
    void swim();
}

class Duck implements CanFly, CanSwim {
    public void fly() {
        System.out.println("Flying");
    }
```

```
    public void swim() {
        System.out.println("Swimming");
    }
}
```

Java Class Hierarchy

All classes in Java inherit from the `Object` class, the root of the class hierarchy.

```
public class TestObject {
    public static void main(String[] args) {
        Object obj = new String("Hello");
        System.out.println(obj.toString());
    }
}
```

Writing Generic Functions — Using `Object` Class

Generic functions can work with any type of objects by using the `Object` class.

```
public class Utils {
    public static void printObject(Object obj) {
        System.out.println(obj.toString());
    }
}
```

Overriding Functions

Method overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.

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

class Dog extends Animal {
```

```
    void makeSound() {
        System.out.println("Bark");
    }
}
```

Overriding Looks for “Closest” Match

When a method is overridden in a subclass, the overridden method is called based on the object's runtime type.

```
Animal myAnimal = new Dog();
myAnimal.makeSound(); // Outputs: Bark
```

The Philosophy of OO Programming

Object-Oriented Design

Object-oriented design involves planning a system of interacting objects for the purpose of solving a software problem.

Object-Oriented Design: Example

Consider designing a simple banking system with classes such as `Bank`, `Account`, and `Customer`.

Designing Objects

Objects are designed to encapsulate data and behavior. They should represent entities relevant to the problem domain.

Relationship Between Classes

Relationships between classes include:

- Inheritance (is-a)
- Association (has-a)
- Dependency (uses-a)

Example:

```
class Customer {
    private String name;
    private Account account;

    public Customer(String name, Account account) {
        this.name = name;
        this.account = account;
    }
}

class Account {
    private double balance;

    public Account(double balance) {
        this.balance = balance;
    }
}
```

WEEK-4

Abstract Classes and Interfaces

Abstract Classes

Abstract classes cannot be instantiated and are meant to be subclassed. They can have abstract methods (without a body) that must be implemented by subclasses.

```
abstract class Animal {
    abstract void makeSound();
```

```
    void sleep() {
        System.out.println("Sleeping");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark");
    }
}
```

Multiple Inheritance

Java does not support multiple inheritance with classes but allows it through interfaces.

```
interface CanFly {
    void fly();
}

interface CanSwim {
    void swim();
}

class Duck implements CanFly, CanSwim {
    public void fly() {
        System.out.println("Flying");
    }

    public void swim() {
        System.out.println("Swimming");
    }
}
```

Callbacks

Implementing a Callback Facility

A callback is a method that gets called in response to an event. Implementing callbacks usually involves using interfaces.

```
interface EventListener {
    void onEvent();
}

class EventNotifier {
    private EventListener listener;

    void setListener(EventListener listener) {
        this.listener = listener;
    }

    void notifyEvent() {
        if (listener != null) {
            listener.onEvent();
        }
    }
}
```

Implementing Callbacks

Implementing a callback by defining an interface and passing an implementation to a method.

```
class MyEventListener implements EventListener {
    public void onEvent() {
        System.out.println("Event occurred!");
    }
}

public class TestCallback {
    public static void main(String[] args) {
        EventNotifier notifier = new EventNotifier();
        notifier.setListener(new MyEventListener());
        notifier.notifyEvent();
    }
```

```
}
```

A Generic Timer

A generic timer that accepts a callback interface.

```
interface TimerListener {
    void onTimeout();
}

class Timer {
    private TimerListener listener;

    void setListener(TimerListener listener) {
        this.listener = listener;
    }

    void start(int milliseconds) {
        try {
            Thread.sleep(milliseconds);
            if (listener != null) {
                listener.onTimeout();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class TestTimer {
    public static void main(String[] args) {
        Timer timer = new Timer();
        timer.setListener(new TimerListener() {
            public void onTimeout() {
                System.out.println("Time's up!");
            }
        });
        timer.start(2000);
```

```
    }
}
```

Use Interfaces

Interfaces define methods that must be implemented by classes. They are used to provide a contract for what a class can do, without dictating how it should do it.

Summary

Callbacks, abstract classes, and interfaces are powerful features in Java that help in creating flexible and reusable code.

Controlled Interaction with Objects

Querying a Database

Controlled interaction involves restricting access to certain methods or fields of a class, often using encapsulation principles.

```
public class Database {
    private String[] data = {"Alice", "Bob", "Charlie"};

    public String getData(int index) {
        if (index >= 0 && index < data.length) {
            return data[index];
        }
        return null;
    }
}
```

Interfaces

Interfaces

Interfaces are abstract types used to specify a set of methods that a class must implement.

```
interface Animal {
    void makeSound();
    void sleep();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }

    public void sleep() {
        System.out.println("Sleeping");
    }
}
```

Exposing Limited Capabilities

Interfaces allow exposing only specific capabilities of a class.

```
interface Readable {
    String read();
}

class Book implements Readable {
    private String content = "This is a book";

    public String read() {
        return content;
    }
}
```

Adding Methods to Interfaces

In Java 8 and later, you can add default and static methods to interfaces.

```
interface Printable {
    void print();
```

```
    default void printWithPrefix(String prefix) {
        System.out.println(prefix + ": " + toString());
    }

    static void printClassName() {
        System.out.println("Printable");
    }
}
```

Dealing with Conflicts

When a class implements multiple interfaces that define the same default method, it must override the method to resolve the conflict.

```
interface InterfaceA {
    default void foo() {
        System.out.println("InterfaceA foo");
    }
}

interface InterfaceB {
    default void foo() {
        System.out.println("InterfaceB foo");
    }
}

class MyClass implements InterfaceA, InterfaceB {
    public void foo() {
        System.out.println("MyClass foo");
    }
}
```

Iterators

Linear List

A linear list is a sequence of elements arranged in a linear order.

```
public class LinearList {
    private int[] data;
    private int size;

    public LinearList(int capacity) {
        data = new int[capacity];
        size = 0;
    }

    public void add(int value) {
        if (size < data.length) {
            data[size++] = value;
        }
    }

    public int get(int index) {
        if (index >= 0 && index < size) {
            return data[index];
        }
        throw new IndexOutOfBoundsException();
    }

    public int size() {
        return size;
    }
}
```

Iteration

Iteration involves traversing through the elements of a collection.

```
public class TestIteration {
    public static void main(String[] args) {
        LinearList list = new LinearList(10);
        list.add(1);
        list.add(2);
        list.add(3);
```

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

Iterator

An iterator is an object that allows you to traverse a collection, one element at a time.

```
import java.util.Iterator;

public class LinearList implements Iterable<Integer> {
    private int[] data;
    private int size;

    public LinearList(int capacity) {
        data = new int[capacity];
        size = 0;
    }

    public void add(int value) {
        if (size < data.length) {
            data[size++] = value;
        }
    }

    public int get(int index) {
        if (index >= 0 && index < size) {
            return data[index];
        }
        throw new IndexOutOfBoundsException();
    }

    public int size() {
        return size;
    }
```

```
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            private int currentIndex = 0;

            public boolean hasNext() {
                return currentIndex < size;
            }

            public Integer next() {
                return data[currentIndex++];
            }
        };
    }
}
```

How Do We Implement `Iterators` in `LinearList`?

We implement `Iterator` by creating an inner class that overrides `hasNext` and `next` methods.

```
import java.util.Iterator;

public class LinearList implements Iterable<Integer> {
    private int[] data;
    private int size;

    public LinearList(int capacity) {
        data = new int[capacity];
        size = 0;
    }

    public void add(int value) {
        if (size < data.length) {
            data[size++] = value;
        }
    }

    public int get(int index) {
```

```
        if (index >= 0 && index < size) {
            return data[index];
        }
        throw new IndexOutOfBoundsException();
    }

    public int size() {
        return size;
    }

    public Iterator<Integer> iterator() {
        return new LinearListIterator();
    }

    private class LinearListIterator implements Iterator<Integer> {
        private int currentIndex = 0;

        public boolean hasNext() {
            return currentIndex < size;
        }

        public Integer next() {
            return data[currentIndex++];
        }
    }
}
```

Using Iterator

Using an iterator to traverse a collection.

```
public class TestIterator {
    public static void main(String[] args) {
        LinearList list = new LinearList(10);
        list.add(1);
        list.add(2);
        list.add(3);
```

```
        Iterator<Integer> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Summary

Iterators provide a standardized way to traverse collections. They enhance the flexibility and readability of code.

Private Classes

Nested Objects

Nested classes are classes defined within other classes. They can be static or non-static (inner classes).

```
public class OuterClass {
    private int outerValue = 10;

    class InnerClass {
        void display() {
            System.out.println("Outer value: " + outerValue);
        }
    }
}

public class TestNestedClass {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display(); // Outputs: Outer value: 10
    }
}
```

WEEK-5

Generics Programming in Java

Generic Functions

Generic functions use type parameters to operate on objects of various types while providing compile-time type safety.

```
public class GenericMethod {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Java Generics — Type Quantifier

Generics in Java allow you to define classes, interfaces, and methods with placeholder types, making your code more flexible and type-safe.

```
public class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }
```

```
    public T getItem() {
        return item;
    }
}
```

Polymorphic Data Structures

Generics enable polymorphic data structures where the type can be specified at runtime.

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

NOTE

Generics provide compile-time type safety and eliminate the need for casting.

Summary

Generics enhance code reusability, type safety, and readability, allowing for more flexible and maintainable code.

Java Generics and Subtyping

Extending Subtyping in Contexts

Generics support subtyping where a generic type with a subclass as its type parameter is not a subtype of the same generic type with a superclass as its type parameter.

```
List<Object> objectList = new ArrayList<>();
List<String> stringList = new ArrayList<>();
objectList = stringList; // Compile-time error
```

Generics and Subtypes

Generics can use subtyping with bounded type parameters.

```
public <T extends Number> void printNumbers(T[] numbers) {
    for (T number : numbers) {
        System.out.println(number);
    }
}
```

Generic Methods

Methods can also be generic, using type parameters to work with different types.

```
public static <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```

Wildcards

Wildcards (?) are used in generics to represent an unknown type.

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

Wildcard Type Variables

Wildcards can be used with type variables to create more flexible code.

```
public void addNumbers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
    list.add(3);
}
```

Bounded Wildcards

Bounded wildcards restrict the types that can be used as type arguments.

```
public void printUpperBounded(List<? extends Number> list) {
    for (Number num : list) {
        System.out.println(num);
    }
}
```

Summary

Generics and subtyping allow for more flexible and reusable code, making it easier to write type-safe and maintainable programs.

Java Generics at Runtime

Erasure of Generic Information

During compilation, generic type information is erased and replaced with raw types, a process known as type erasure.

```
public class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
```

```
    }

    public T getItem() {
        return item;
    }
}
```

At runtime, `Box<Integer>` and `Box<String>` both become `Box`.

Erasure and Overloading

Overloading methods that differ only by their generic type parameters is not allowed due to type erasure.

```
public class ErasureExample {
    public void print(List<String> list) { }
    public void print(List<Integer> list) { } // Compile-time error
}
```

Arrays and Generics

Arrays and generics interact poorly because arrays are covariant and reifiable, while generics are invariant and non-reifiable.

```
List<Integer>[] array = new List<Integer>[10]; // Compile-time error
```

Wrapper Classes

Wrapper classes are used to wrap primitive types in objects.

```
Integer intObject = Integer.valueOf(5);
int intValue = intObject.intValue();
```

Using Wrapper Classes

Wrapper classes are often used in collections and generics where primitive types are not allowed.

```
List<Integer> intList = new ArrayList<>();
```

```
intList.add(10);
intList.add(20);
```

Polymorphism

Polymorphism

Polymorphism allows objects to be treated as instances of their parent class rather than their actual class.

```
Animal animal = new Dog();
animal.makeSound(); // Calls Dog's makeSound method
```

Structural Polymorphism

Structural polymorphism refers to the ability of different classes to be used interchangeably if they follow the same interface.

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Drawing Square");
    }
}
```

Type Consistency

Type consistency ensures that operations on variables are type-safe.

```
List<String> strings = new ArrayList<>();
strings.add("hello");
String s = strings.get(0); // No cast needed
```

Polymorphic Data Structures

Polymorphic data structures use generics to operate on different types.

```
public class GenericList<T> {
    private List<T> list = new ArrayList<>();

    public void add(T item) {
        list.add(item);
    }

    public T get(int index) {
        return list.get(index);
    }
}
```

Generic Programming in Java

Generic programming allows for the creation of flexible and reusable software components.

```
public class GenericMethod {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Reflection

Reflection

Reflection allows programs to inspect and manipulate the runtime behavior of applications.

```
import java.lang.reflect.Method;

public class ReflectionExample {
    public static void main(String[] args) {
        try {
            Class<?> c = Class.forName("java.util.ArrayList");
            Method[] methods = c.getDeclaredMethods();
            for (Method method : methods) {
                System.out.println(method.getName());
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Reflection in Java

Reflection in Java allows you to inspect classes, interfaces, fields, and methods at runtime without knowing their names at compile time.

```
public class TestReflection {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = Class.forName("java.util.ArrayList");
        System.out.println("Class name: " + clazz.getName());
    }
}
```

Introspection in Java

Introspection is a subset of reflection, specifically dealing with analyzing a class's properties and behavior.

```
import java.beans.Introspector;
import java.beans.PropertyDescriptor;

public class IntrospectionExample {
```

```
    public static void main(String[] args) throws Exception {
        PropertyDescriptor[] propertyDescriptors =
Introspector.getBeanInfo(Person.class).getPropertyDescriptors();
        for (PropertyDescriptor propertyDescriptor :
propertyDescriptors) {
            System.out.println(propertyDescriptor.getName());
        }
    }
}
```

The Class `Class`

The `Class` class represents classes and interfaces in a running Java application.

```
public class TestClass {
    public static void main(String[] args) {
        Class<String> stringClass = String.class;
        System.out.println(stringClass.getName());
    }
}
```

Using the `Class` Object

Using the `Class` object, you can get metadata about the class and create new instances.

```
public class ReflectionDemo {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = Class.forName("java.util.ArrayList");
        Object instance =
clazz.getDeclaredConstructor().newInstance();
        System.out.println(instance.getClass().getName());
    }
}
```

Reflection and Security

Reflection can pose security risks if used improperly. Ensure proper access controls are in place when using reflection.

```
import java.lang.reflect.Field;

public class ReflectionSecurity {
    public static void main(String[] args) throws Exception {
        Field field = String.class.getDeclaredField("value");
        field.setAccessible(true);
        char[] value = (char[]) field.get("test");
        value[0] = 'T';
        System.out.println("test"); // Outputs "Test"
    }
}
```

Limitations of Java Reflection

Java reflection has limitations, such as performance overhead, lack of compile-time checking, and potential security issues. It should be used judiciously.

WEEK-6

Collection

The `Collection` Interface

The `Collection` interface is the root interface in the collection hierarchy. It represents a group of objects known as elements.

```
Collection<String> collection = new ArrayList<>();
collection.add("Element 1");
```

```
collection.add("Element 2");
```

Using Iterators

Iterators allow traversing the elements of a collection.

```
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Removing Elements

Elements can be removed using an iterator's `remove` method.

```
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
    String element = iterator.next();
    if (element.equals("Element 1")) {
        iterator.remove();
    }
}
```

The `Collection` Interface — The Full Story

The `Collection` interface includes methods for adding, removing, and querying elements, and checking the collection's size.

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    boolean add(E e);
    boolean remove(Object o);
    void clear();
}
```

The `AbstractCollection` Class

The `AbstractCollection` class provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface.

```
public abstract class AbstractCollection<E> implements Collection<E> {
    public boolean isEmpty() {
        return size() == 0;
    }

    public boolean contains(Object o) {
        for (E element : this) {
            if (element.equals(o)) {
                return true;
            }
        }
        return false;
    }

    public abstract Iterator<E> iterator();
    public abstract int size();
}
```

Concrete Collections

Built-in Data Types

Java provides several concrete implementations of the `Collection` interface, such as `ArrayList`, `HashSet`, and `LinkedList`.

```
List<String> arrayList = new ArrayList<>();
Set<String> hashSet = new HashSet<>();
Queue<String> linkedList = new LinkedList<>();
```

The `List` Interface

The `List` interface extends `Collection` and represents an ordered collection.

```
List<String> list = new ArrayList<>();
list.add("Element 1");
list.add("Element 2");
System.out.println(list.get(0)); // Outputs: Element 1
```

The `List` Interface and Random Access

`List` allows random access to elements.

```
System.out.println(list.get(1)); // Outputs: Element 2
```

The `AbstractList` Class

The `AbstractList` class provides a skeletal implementation of the `List` interface.

```
public abstract class AbstractList<E> extends AbstractCollection<E>
implements List<E> {
    public boolean add(E e) {
        add(size(), e);
        return true;
    }

    public abstract E get(int index);
    public abstract E set(int index, E element);
    public abstract void add(int index, E element);
    public abstract E remove(int index);
}
```

Using Concrete List Classes

Concrete list classes like `ArrayList` and `LinkedList` can be used for various list operations.

```
List<String> arrayList = new ArrayList<>();
arrayList.add("Element 1");
arrayList.add("Element 2");

List<String> linkedList = new LinkedList<>();
linkedList.add("Element A");
```

```
linkedList.add("Element B");
```

The `Set` Interface

The `Set` interface extends `Collection` and represents a collection that does not allow duplicate elements.

```
Set<String> set = new HashSet<>();
set.add("Element 1");
set.add("Element 1"); // No effect, duplicate element
```

Concrete Sets

Concrete implementations of the `Set` interface include `HashSet`, `LinkedHashSet`, and `TreeSet`.

```
Set<String> hashSet = new HashSet<>();
Set<String> linkedHashSet = new LinkedHashSet<>();
Set<String> treeSet = new TreeSet<>();
```

The `Queue` Interface

The `Queue` interface extends `Collection` and represents a collection designed for holding elements prior to processing.

```
Queue<String> queue = new LinkedList<>();
queue.add("Element 1");
queue.add("Element 2");
System.out.println(queue.poll()); // Outputs: Element 1
```

Maps

Maps

Maps are collections that map keys to values. The `Map` interface does not extend `Collection`.

```
Map<String, String> map = new HashMap<>();
map.put("key1", "value1");
map.put("key2", "value2");
```

Updating a Map

Maps allow updating values associated with keys.

```
map.put("key1", "newValue1");
```

Extracting Keys and Values

You can extract keys, values, and key-value pairs from a map.

```
Set<String> keys = map.keySet();
Collection<String> values = map.values();
Set<Map.Entry<String, String>> entries = map.entrySet();
```

Concrete Implementation of Map

Concrete implementations of the `Map` interface include `HashMap`, `TreeMap`, and `LinkedHashMap`.

```
Map<String, String> hashMap = new HashMap<>();
Map<String, String> treeMap = new TreeMap<>();
Map<String, String> linkedHashMap = new LinkedHashMap<>();
```

The Benefits of Indirection

Abstract Data Types

Abstract data types (ADTs) provide a way to specify the behavior of a data structure independently of its implementation.

```
interface Stack<E> {
    void push(E item);
    E pop();
```

```
    boolean isEmpty();
}
```

Multiple Implementations

ADTs can have multiple implementations, allowing flexibility in choosing the most appropriate implementation.

```
class ArrayStack<E> implements Stack<E> {
    private List<E> list = new ArrayList<>();

    public void push(E item) {
        list.add(item);
    }

    public E pop() {
        if (list.isEmpty()) {
            throw new EmptyStackException();
        }
        return list.remove(list.size() - 1);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }
}
```

Adding Indirection

Indirection involves adding an extra layer to improve flexibility, such as using interfaces and abstract classes.

```
public class Client {
    private Stack<Integer> stack;

    public Client(Stack<Integer> stack) {
        this.stack = stack;
    }
```

```
    public void performOperations() {
        stack.push(10);
        stack.push(20);
        System.out.println(stack.pop()); // Outputs: 20
    }
}

public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new ArrayStack<>();
        Client client = new Client(stack);
        client.performOperations();
    }
}
```

WEEK-7

Assertions

Documenting and Checking Assumptions

Assertions are used to document and verify assumptions made in the code. They help identify bugs by catching incorrect assumptions during runtime.

```
public class AssertionsExample {
    public static void main(String[] args) {
        int age = -1;
        assert age >= 0 : "Age cannot be negative";
    }
}
```

Assertions

Assertions are a development tool used to test code during development.

```
public class AssertionsExample {
    public static void main(String[] args) {
        int num = -10;
        assert num > 0 : "Number must be positive";
    }
}
```

Enabling and Disabling Assertions

Assertions can be enabled or disabled at runtime using the `-ea` or `-da` flags.

```
java -ea AssertionsExample   # Enable assertions
java -da AssertionsExample   # Disable assertions
```

Dealing with Errors

Exception Handling

Exception handling is a mechanism to handle runtime errors. It ensures the flow of the program does not break when an exception occurs.

```
try {
    int division = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("ArithmeticException caught: " +
e.getMessage());
}
```

Java's Classification of Errors

Java classifies errors into `Error` and `Exception`. `Error` represents serious issues that a typical application should not try to catch, while `Exception` represents conditions that a reasonable application might want to catch.

Summary

Exception handling in Java is essential for building robust and error-resilient applications. Proper use of try-catch blocks, custom exceptions, and resource management can significantly enhance the stability and maintainability of your code.

Exceptions in Java

Catching and Handling Exceptions

Exceptions are caught using a `try-catch` block.

```
try {
    int[] array = new int[5];
    array[10] = 50; // This will throw ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Exception caught: " + e.getMessage());
}
```

Catching Multiple Exceptions

You can catch multiple exceptions in a single `catch` block.

```
try {
    int num = Integer.parseInt("XYZ");
    int division = 10 / 0;
} catch (NumberFormatException | ArithmeticException e) {
    System.out.println("Exception caught: " + e.getMessage());
}
```

Notify Checked Exceptions

Methods can declare checked exceptions they may throw.

```
public void readFile(String filePath) throws IOException {
```

```
    BufferedReader reader = new BufferedReader(new
FileReader(filePath));
    String line = reader.readLine();
    reader.close();
}
```

Throwing Exceptions

Exceptions can be thrown using the `throw` keyword.

```
public void validateAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Age must be 18 or older");
    }
}
```

Customized Exceptions

Custom exceptions can be created by extending the `Exception` class.

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

More on Catching Exceptions

Use multiple `catch` blocks to handle different exceptions separately.

```
try {
    int[] array = new int[5];
    array[10] = 50;
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index out of bounds: " +
e.getMessage());
} catch (Exception e) {
    System.out.println("General exception: " + e.getMessage());
```

```
}
```

Cleaning Up Resources

Use `finally` block to clean up resources.

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("file.txt"));
    String line = reader.readLine();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (reader != null) {
            reader.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Logging

Diagnostic Messages

Diagnostic messages provide information about the application's state, useful for debugging and monitoring.

```
System.out.println("Starting application...");
```

Logging

Java provides a logging API through `java.util.logging` package.

```
import java.util.logging.Logger;
```

```
public class LoggingExample {
    private static final Logger logger =
Logger.getLogger(LoggingExample.class.getName());

    public static void main(String[] args) {
        logger.info("Application started");
    }
}
```

Logging Levels

Different logging levels indicate the severity of the log messages.

```
logger.severe("Severe message");
logger.warning("Warning message");
logger.info("Info message");
logger.config("Config message");
logger.fine("Fine message");
logger.finer("Finer message");
logger.finest("Finest message");
```

Summary

Logging is a critical part of application development and maintenance. It helps in tracking the flow of the application and diagnosing issues. Java's logging framework offers a flexible and configurable way to handle logging.

Packages

Packages

Packages are used to group related classes and interfaces. They provide access protection and namespace management.

```
package com.example.myapp;

public class MyClass {
    // Class code here
```

```
}
```

Creating and Naming Packages

Packages are typically named using the domain name of the organization in reverse.

```
package com.mycompany.project.module;
```

More About Visibility

Access modifiers determine the visibility of classes, methods, and fields.

- **Public:** Accessible from any other class.
- **Protected:** Accessible within the same package and subclasses.
- **Default (package-private):** Accessible only within the same package.
- **Private:** Accessible only within the same class.

```
public class MyClass {
    public int publicField;
    protected int protectedField;
    int defaultField;
    private int privateField;
}
```

WEEK-8

Cloning

Copying an Object

Copying an object in Java can be done in two main ways: shallow copy and deep copy.

The `clone()` Method

The `clone()` method is used to create a copy of an object. The class must implement the `Cloneable` interface, and override the `clone()` method.

```
public class Person implements Cloneable {
    private String name;

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

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Shallow Copy

A shallow copy copies the object but not the objects it references.

```
public class Person implements Cloneable {
    private String name;
    private Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

Person original = new Person("John", new Address("123 Street"));
Person copy = (Person) original.clone();
```

Deep Copy

A deep copy copies the object and the objects it references.

```
public class Person implements Cloneable {
    private String name;
    private Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Person cloned = (Person) super.clone();
        cloned.address = (Address) address.clone();
        return cloned;
    }
}
```

Restrictions on `clone()`

The `clone()` method has restrictions:

- Classes must implement `Cloneable`.
- The `clone()` method in `Object` class performs a shallow copy.
- Custom deep cloning requires overriding the `clone()` method.

Higher Order Functions

Passing Functions

Java supports higher-order functions through functional interfaces and lambda expressions.

```
interface Operation {
    int execute(int a, int b);
```

```
}

public class HigherOrderFunctionExample {
    public static void main(String[] args) {
        Operation addition = (a, b) -> a + b;
        System.out.println(performOperation(5, 3, addition)); //
Outputs: 8
    }

    public static int performOperation(int a, int b, Operation op) {
        return op.execute(a, b);
    }
}
```

Functional Interfaces

Functional interfaces have a single abstract method and can be implemented using lambda expressions.

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        Greeting greeting = name -> System.out.println("Hello, " +
name);
        greeting.sayHello("Alice");
    }
}
```

Lambda Expressions

Lambda expressions provide a concise way to implement functional interfaces.

```
Operation subtraction = (a, b) -> a - b;
System.out.println(performOperation(5, 3, subtraction)); // Outputs: 2
```

Passed Named Functions

Methods can be passed as arguments using method references.

```
public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        names.forEach(System.out::println);
    }
}
```

Method References

Method references simplify the syntax for passing methods as arguments.

```
Operation multiplication = Integer::sum;
System.out.println(performOperation(5, 3, multiplication)); //
Outputs: 8
```

Streams

Operating on Collections

Streams allow operations on collections in a functional style.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream().forEach(System.out::println);
```

Why Streams?

Streams provide a way to process collections of data in a declarative manner, supporting operations such as filtering, mapping, and reducing.

Working with Streams

Streams are created from collections and support various operations.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
numbers.stream()
       .filter(n -> n % 2 == 0)
       .forEach(System.out::println); // Outputs: 2, 4
```

Creating Streams

Streams can be created from collections, arrays, or custom sources.

```
Stream<String> stream = Stream.of("Alice", "Bob", "Charlie");
```

Processing Streams

Streams can be processed using intermediate operations (e.g., `filter`, `map`) and terminal operations (e.g., `forEach`, `reduce`).

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
long count = names.stream()
                  .filter(name -> name.startsWith("A"))
                  .count();
System.out.println(count); // Outputs: 1
```

Stream Transformations

Stream transformations apply functions to elements in the stream.

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
List<Integer> squares = numbers.stream()
                               .map(n -> n * n)
                               .collect(Collectors.toList());
System.out.println(squares); // Outputs: [1, 4, 9]
```

Reducing a Stream to a Result

Streams can be reduced to a single result using the `reduce` method.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .reduce(0, Integer::sum);
```

```
System.out.println(sum); // Outputs: 15
```

Type Inference

Type Declarations vs Type Inference

Type declarations explicitly specify the type, while type inference allows the compiler to deduce the type.

```
List<String> names = new ArrayList<>(); // Type inference
List<String> names = new ArrayList<String>(); // Type declaration
```

Type Inference

Type inference simplifies code by reducing verbosity.

```
var names = new ArrayList<String>();
names.add("Alice");
System.out.println(names.get(0)); // Outputs: Alice
```

Type Inference in Java

Java supports type inference for local variables, method parameters, and return types.

```
var list = List.of("Alice", "Bob", "Charlie");
for (var name : list) {
    System.out.println(name);
}
```

WEEK-9

Collecting Results from Streams

Collecting Values from a Stream

Use the `collect` method to gather the elements of a stream into a collection.

```
List<String> names = List.of("Alice", "Bob", "Charlie");
List<String> filteredNames = names.stream()
                                  .filter(name ->
name.startsWith("A"))
                                  .collect(Collectors.toList());
System.out.println(filteredNames); // Outputs: [Alice]
```

Storing a Stream as a Collection

You can collect the stream elements into various collection types.

```
Set<String> nameSet = names.stream()
                           .collect(Collectors.toSet());
```

Stream Summaries

Use collectors to get summary statistics.

```
IntSummaryStatistics stats = numbers.stream()
                                    .mapToInt(Integer::intValue)
                                    .summaryStatistics();
System.out.println(stats.getAverage());
System.out.println(stats.getSum());
```

Converting a Stream to a Map

Convert a stream to a map using `Collectors.toMap`.

```
Map<String, Integer> nameLengthMap = names.stream()

.collect(Collectors.toMap(name -> name, name -> name.length()));
System.out.println(nameLengthMap);
```

Grouping and Partitioning Values

Group or partition the stream elements based on a criterion.

```
Map<Boolean, List<String>> partitionedNames = names.stream()

.collect(Collectors.partitioningBy(name -> name.length() > 3));
System.out.println(partitionedNames);
```

Input/Output Streams

Input and Output Streams

Java provides `InputStream` and `OutputStream` classes for handling byte-based input and output.

Reading Raw Bytes

Read raw bytes from an `InputStream`.

```
try (InputStream in = new FileInputStream("input.txt")) {
    int byteData;
    while ((byteData = in.read()) != -1) {
        System.out.print((char) byteData);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Writing Raw Bytes

Write raw bytes to an `OutputStream`.

```
try (OutputStream out = new FileOutputStream("output.txt")) {
    out.write("Hello, World!".getBytes());
} catch (IOException e) {
    e.printStackTrace();
}
```

Connecting a Stream to an External Source

Streams can be connected to files, network sockets, etc.

```
try (InputStream in = new URL("http://example.com").openStream()) {
    // Read from URL
} catch (IOException e) {
    e.printStackTrace();
}
```

Reading Text

Use `BufferedReader` for reading text efficiently.

```
try (BufferedReader reader = new BufferedReader(new
FileReader("input.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Writing Text

Use `BufferedWriter` for writing text efficiently.

```
try (BufferedWriter writer = new BufferedWriter(new
FileWriter("output.txt"))) {
    writer.write("Hello, World!");
} catch (IOException e) {
```

```
    e.printStackTrace();
}
```

Reading Binary Data

Use `DataInputStream` to read binary data.

```
try (DataInputStream in = new DataInputStream(new
FileInputStream("data.bin"))) {
    int number = in.readInt();
    System.out.println(number);
} catch (IOException e) {
    e.printStackTrace();
}
```

Writing Binary Data

Use `DataOutputStream` to write binary data.

```
try (DataOutputStream out = new DataOutputStream(new
FileOutputStream("data.bin"))) {
    out.writeInt(42);
} catch (IOException e) {
    e.printStackTrace();
}
```

Other Features

Java I/O streams support various features like buffering, character encoding, and more for efficient and flexible input/output operations.

Optional Types

Dealing with Empty Streams

Use `Optional` to handle cases where a stream might be empty.

```
Optional<String> firstName = names.stream()
                                  .filter(name ->
name.startsWith("Z"))
                                  .findFirst();
```

Handling Missing Optional Values

Use `orElse` or `orElseGet` to handle missing values.

```
String name = firstName.orElse("No name found");
```

Ignoring Missing Values

Use `ifPresent` to perform an action if a value is present.

```
firstName.ifPresent(System.out::println);
```

Creating an Optional Value

Create an `Optional` value using `Optional.of` or `Optional.ofNullable`.

```
Optional<String> optionalName = Optional.of("Alice");
Optional<String> optionalEmpty = Optional.ofNullable(null);
```

Passing on Optional Values

Pass `Optional` values between methods.

```
public Optional<String> findName(List<String> names) {
    return names.stream()
                .filter(name -> name.startsWith("A"))
                .findFirst();
}
```

Composing Optional Values of Different Types

Combine multiple `Optional` values using `flatMap`.

```
Optional<String> fullName = firstName.flatMap(fName ->
    lastName.map(lName -> fName + " " + lName));
```

Turning an Optional into a Stream

Convert an `Optional` to a stream.

```
List<String> nameList = firstName.stream()
                                 .collect(Collectors.toList());
```

Summary

`Optional` helps in handling null values gracefully, avoiding `NullPointerException`, and writing more readable and maintainable code.

Serialization

Reading and Writing Objects

Serialize objects to a stream using `ObjectOutputStream` and deserialize using `ObjectInputStream`.

```
try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("person.ser"))) {
    out.writeObject(new Person("Alice", 30));
} catch (IOException e) {
    e.printStackTrace();
}

try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("person.ser"))) {
    Person person = (Person) in.readObject();
    System.out.println(person);
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

How Serialization Works

Serialization converts an object into a byte stream, while deserialization reconstructs the object from the byte stream.

Customizing Serialization

Customize serialization by implementing `readObject` and `writeObject` methods.

```
private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    out.writeInt(age);
}

private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
    in.defaultReadObject();
    age = in.readInt();
}
```

WEEK-10

Concurrency: Threads & Processes

Concurrent Programming

Concurrent programming involves multiple threads executing independently but possibly interacting with each other. It is used to improve the efficiency and responsiveness of applications.

Creating Threads in Java

Threads can be created by extending the `Thread` class or implementing the `Runnable` interface.

```
// Extending Thread class
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}

// Implementing Runnable interface
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable is running");
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

Java Threads

Java threads are lightweight processes that share the same address space. They can be created using the `Thread` class or the `ExecutorService`.

Monitors

Problem Overview: Atomic Test-and-Set

The atomic test-and-set problem involves ensuring that a variable can only be modified by one thread at a time to avoid race conditions.

Monitors

Monitors are synchronization constructs that provide a way to achieve mutual exclusion and condition synchronization. They use `synchronized` methods or blocks in Java.

```
public class MonitorExample {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}
```

Monitors: External Queue

Monitors can have an external queue to manage waiting threads.

Making Monitors More Flexible

Monitors can be made more flexible by using condition variables to allow threads to wait and be notified under certain conditions.

Monitors — `wait()`

The `wait()` method causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for the same object.

```
public synchronized void waitMethod() throws InterruptedException {
    wait();
}
```

Monitors — `notify()`

The `notify()` method wakes up a single thread that is waiting on the object's monitor.

```
public synchronized void notifyMethod() {
    notify();
}
```

Monitors — `wait()` and `notify()`

Using `wait()` and `notify()` together allows threads to communicate about the state of the object.

```
public synchronized void waitAndNotify() throws InterruptedException {
    wait();
    // Some code
    notify();
}
```

Condition Variables

Condition variables are used to manage the state and the condition under which threads should wait and be notified.

Mutual Exclusion

Mutual Exclusion

Mutual exclusion ensures that only one thread can access a critical section at a time. This can be achieved using synchronized blocks or locks.

```
public class MutexExample {
    private final Object lock = new Object();

    public void criticalSection() {
        synchronized (lock) {
            // Critical section code
        }
```

```
    }
}
```

First Attempt: Mutual Exclusion for Two Processes

The simplest mutual exclusion can be achieved by using flags for each process to indicate whether it wants to enter the critical section.

Second Attempt: Mutual Exclusion for Two Processes

A more sophisticated approach uses turn variables to alternate access between processes.

Peterson’s Algorithm

Peterson's algorithm ensures mutual exclusion between two processes using two flags and a turn variable.

```
class Peterson {
    private volatile boolean[] flag = new boolean[2];
    private volatile int turn;

    public void enterCritical(int i) {
        int j = 1 - i;
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j) {
            // busy wait
        }
    }

    public void leaveCritical(int i) {
        flag[i] = false;
    }
}
```

Beyond Two Processes

For more than two processes, algorithms like the Bakery algorithm or using more sophisticated locking mechanisms are required.

Race Conditions

Example: Maintain Data Consistency

Race conditions occur when multiple threads access shared data concurrently and try to change it at the same time.

```
public class RaceConditionExample {
    private int count = 0;

    public void increment() {
        count++;
    }

    public static void main(String[] args) {
        RaceConditionExample example = new RaceConditionExample();
        Runnable task = example::increment;

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);

        t1.start();
        t2.start();
    }
}
```

Race Conditions and Mutual Exclusion

Race conditions can be prevented by using mutual exclusion techniques like synchronized blocks or locks to ensure only one thread can modify the shared data at a time.

Test and Set

Test and Set

The test-and-set operation is used to achieve mutual exclusion by atomically testing and setting a lock variable.

```
public class TestAndSetLock {
    private AtomicBoolean lock = new AtomicBoolean(false);

    public void lock() {
        while (lock.getAndSet(true)) {
            // busy wait
        }
    }

    public void unlock() {
        lock.set(false);
    }
}
```

Semaphores

Semaphores are synchronization constructs that control access to a resource by multiple threads. They can be used to implement mutual exclusion and manage resource allocation.

```
Semaphore semaphore = new Semaphore(1);

public void accessResource() throws InterruptedException {
    semaphore.acquire();
    try {
        // access resource
    } finally {
        semaphore.release();
    }
}
```

Using Semaphores

Semaphores can be used for various synchronization tasks like managing a pool of resources.

```
Semaphore semaphore = new Semaphore(5); // Allows up to 5 permits

public void accessResource() throws InterruptedException {
    semaphore.acquire();
```

```
    try {
        // access resource
    } finally {
        semaphore.release();
    }
}
```

Problems with Semaphores

Semaphores can lead to issues like deadlock if not used properly, where two or more threads are waiting for each other to release resources.

WEEK-11

Example: Concurrency Programming

An Exercise in Concurrent Programming

Create a simple counter that multiple threads will increment concurrently to demonstrate synchronization and race conditions.

An Example

Here's an example of a counter with multiple threads incrementing it without synchronization, leading to race conditions.

```
public class Counter {
    private int count = 0;
```

```
    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }

    public static void main(String[] args) throws InterruptedException
{
        Counter counter = new Counter();
        Runnable task = counter::increment;

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println(counter.getCount()); // Expected 2 but may
be less due to race conditions
    }
}
```

Analysis

The counter may not reach the expected value due to race conditions. Both threads read and write the `count` variable simultaneously, leading to inconsistent results.

Code for `enter`

Implement the `enter` method to handle synchronization using the `synchronized` keyword.

```
public synchronized void enter() {
    count++;
}
```

Code for `leave`

Ensure proper cleanup or state reset if needed in the `leave` method.

```
public synchronized void leave() {
    // Cleanup or state reset if necessary
}
```

Summary

Synchronization is crucial in concurrent programming to prevent race conditions and ensure data consistency.

Monitors in Java

Monitors in Java

Monitors in Java are implemented using the `synchronized` keyword, which can be applied to methods or blocks to ensure that only one thread can access the critical section at a time.

Object Locks

Each object in Java has an intrinsic lock or monitor that is used for synchronization. When a thread enters a synchronized method or block, it acquires the object's lock.

Object Locks...

An object lock ensures that only one thread can execute a synchronized method or block at any given time.

```
public class SynchronizedExample {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
```

```
}
```

Reentrant Locks

Reentrant locks are provided by the `java.util.concurrent.locks` package and offer more flexibility than the intrinsic locks.

```
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public int getCount() {
        return count;
    }
}
```

Thread Safe Collection

Concurrency and Collections

Standard collections in Java are not thread-safe and can lead to concurrency issues when accessed by multiple threads simultaneously.

Thread Safety and Correctness

Thread safety ensures that a class behaves correctly when accessed from multiple threads. For collections, this means preventing race conditions and ensuring data consistency.

Thread Safe Collections

Java provides thread-safe collections like `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `BlockingQueue`.

```
import java.util.concurrent.ConcurrentHashMap;

public class ThreadSafeCollectionsExample {
    private ConcurrentHashMap<String, Integer> map = new
ConcurrentHashMap<>();

    public void add(String key, Integer value) {
        map.put(key, value);
    }

    public Integer get(String key) {
        return map.get(key);
    }
}
```

Using Thread Safe Queues for Synchronization

Thread-safe queues like `BlockingQueue` can be used for producer-consumer scenarios.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueueExample {
    private BlockingQueue<Integer> queue = new
ArrayBlockingQueue<>(10);

    public void produce(int value) throws InterruptedException {
        queue.put(value);
    }

    public int consume() throws InterruptedException {
        return queue.take();
    }
}
```

Blocking Queues

`BlockingQueue` provides thread-safe operations for adding and removing elements, blocking when the queue is full or empty.

```
public class ProducerConsumer {
    private BlockingQueue<Integer> queue = new
ArrayBlockingQueue<>(10);

    public void produce(int value) throws InterruptedException {
        queue.put(value);
    }

    public int consume() throws InterruptedException {
        return queue.take();
    }

    public static void main(String[] args) throws InterruptedException
{
        ProducerConsumer pc = new ProducerConsumer();
        Runnable producer = () -> {
            try {
                for (int i = 0; i < 10; i++) {
                    pc.produce(i);
                    System.out.println("Produced: " + i);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        };

        Runnable consumer = () -> {
            try {
                for (int i = 0; i < 10; i++) {
                    int value = pc.consume();
                    System.out.println("Consumed: " + value);
                }
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        };

        Thread producerThread = new Thread(producer);
        Thread consumerThread = new Thread(consumer);

        producerThread.start();
        consumerThread.start();

        producerThread.join();
        consumerThread.join();
    }
}
```

Thread in Java

Life Cycle of a Java Thread

A Java thread goes through several states: New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated.

- **New**: Thread is created but not yet started.
- **Runnable**: Thread is eligible to run but may not be running.
- **Blocked**: Thread is blocked waiting for a monitor lock.
- **Waiting**: Thread is waiting indefinitely for another thread to perform a particular action.
- **Timed Waiting**: Thread is waiting for another thread to perform a specific action for up to a specified waiting time.
- **Terminated**: Thread has exited.

Interrupts

Threads can be interrupted to stop their execution or to handle particular events.

```
public class InterruptExample {
    public static void main(String[] args) throws InterruptedException
{
        Thread thread = new Thread(() -> {
```

```
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Running");
            }
            System.out.println("Interrupted");
        });

        thread.start();
        Thread.sleep(1000);
        thread.interrupt();
    }
}
```

More About Threads

Java provides various utilities for advanced thread management like `ExecutorService`, `CountDownLatch`, `CyclicBarrier`, and `ForkJoinPool` for handling complex concurrent tasks efficiently.

WEEK-12

Graphical Interfaces and Event-Driven Programming

GUIs and Events

Graphical User Interfaces (GUIs) in Java are typically created using the Swing framework. Events in GUIs are actions that occur as a result of user interaction, such as clicking a button or

typing in a text field. Event-driven programming is a paradigm where the flow of the program is determined by these events.

Keeping Track of Events

To handle events, you need to implement event listeners that define how the application should respond to various actions.

Better Programming Language (PL) Support for Events

Java provides robust support for event-driven programming through its event-handling model, which includes event listeners, event objects, and event sources.

Example

Here's a simple example of a GUI with a button that responds to a click event.

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SimpleGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple GUI");
        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button was clicked!");
            }
        });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Timer

Timers can be used to perform actions at regular intervals. The `javax.swing.Timer` class is useful for this purpose.

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TimerExample {
    public static void main(String[] args) {
        Timer timer = new Timer(1000, new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Timer ticked");
            }
        });
        timer.start();

        JOptionPane.showMessageDialog(null, "Close to stop timer");
        System.exit(0);
    }
}
```

Summary

Event-driven programming in Java using Swing involves creating GUI components and handling events with event listeners. This approach makes it easy to create interactive applications.

More Swing Examples

Connecting Multiple Events to a Listener

A single listener can handle multiple events from different sources.

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MultipleEventsExample {
    public static void main(String[] args) {
```

```
        JFrame frame = new JFrame("Multiple Events Example");
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");

        ActionListener listener = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println(e.getActionCommand() + " was
clicked");
            }
        };

        button1.addActionListener(listener);
        button2.addActionListener(listener);

        frame.setLayout(new java.awt.FlowLayout());
        frame.add(button1);
        frame.add(button2);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Multicasting: Multiple Listeners for an Event

An event source can have multiple listeners.

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MultipleListenersExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Multiple Listeners Example");
        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```

```
                System.out.println("First listener");
            }
        });

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Second listener");
            }
        });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Other Elements - Checkboxes

Checkboxes allow users to make binary choices.

```
import javax.swing.*;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

public class CheckBoxExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Checkbox Example");
        JCheckBox checkBox = new JCheckBox("Check Me");

        checkBox.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    System.out.println("Checkbox selected");
                } else {
                    System.out.println("Checkbox deselected");
                }
            }
```

```
        });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(checkBox);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Summary

Swing allows for sophisticated event handling, including connecting multiple events to a single listener and multicasting events to multiple listeners. It also provides various UI components like checkboxes.

Swing ToolKit

Event Driven Programming in Java

Event-driven programming with Swing involves creating UI components and defining how they respond to user actions through event listeners.

A Button That Paints Its Background Red

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class RedButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Red Button Example");
        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                button.setBackground(Color.RED);
            }
        });
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Embedding the Button Inside a Panel

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ButtonInPanelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button in Panel Example");
        JPanel panel = new JPanel();
        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                button.setBackground(Color.RED);
            }
        });

        panel.add(button);
        frame.add(panel);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Frame

A frame is the main window that can contain other UI components.

```
import javax.swing.*;

public class FrameExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Frame Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        frame.setVisible(true);
    }
}
```

`main` Function

The `main` function is the entry point of a Java application. It sets up and displays the GUI.

```
public class MainFunctionExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Main Function Example");
        JButton button = new JButton("Click Me");

        button.addActionListener(e -> System.out.println("Button
clicked"));

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```
