Java Basics

Basic Operators

Arithmetic Operators

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Modulus (Remainder): `%`

Assignment Operators

- Assignment: `=`
- Addition assignment: `+=`
- Subtraction assignment: `-=`
- Multiplication assignment: `*=`
- Division assignment: `/=`
- Modulus assignment: `%=`

Comparison Operators

- Equal to: `==`
- Not equal to: `!=`
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`

Logical Operators

- Logical AND: `&&`
- Logical OR: `||`
- Logical NOT: `!`

Increment and Decrement Operators

- Increment: `++`
- Decrement: `--`

Basic Syntax

Java Class Structure

```java
public class ClassName {
    // Fields (variables)
    int x;
    double y;
    String name;

    // Constructor(s)
    public ClassName(int x, double y, String name) {
        this.x = x;
        this.y = y;
        this.name = name;
    }

    // Methods
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("x: " + x + ", y: " + y);
    }

    public static void main(String[] args) {
        ClassName obj = new ClassName(10, 3.14, "John");
        obj.display();
    }
}
```

Variables and Data Types

```java
int age = 25;
double salary = 50000.50;
char grade = 'A';
boolean isStudent = true;
String name = "Alice";
```

Input and Output

```java
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);
System.out.print("Enter your name: ");
String inputName = scanner.nextLine();
System.out.println("Hello, " + inputName + "!");

System.out.println("Value of x: " + x);
System.out.printf("Formatted salary: %.2f", salary);
```

Strings and Arrays in Java

Java provides robust support for working with strings and arrays, which are fundamental data structures in programming. Here's an overview of strings and arrays in Java with code examples:

Strings

In Java, strings are sequences of characters enclosed in double quotation marks (""). Strings are widely used for storing and manipulating text.

String Declaration and Initialization

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

String Concatenation

You can concatenate strings using the `+` operator or the `concat()` method:

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;
```

String Length

You can get the length of a string using the `length()` method:

```
int length = fullName.length(); // length will be 8
```

String Comparison

You can compare strings using the `equals()` method for content comparison or `compareTo()` for lexicographic comparison:

```
String str1 = "apple";
String str2 = "banana";

boolean isEqual = str1.equals(str2); // isEqual will be false
int result = str1.compareTo(str2);    // result will be negative
```

Substrings

You can extract substrings from a string using the `substring()` method:

```
String text = "Hello, World!";
String sub = text.substring(7); // sub will be "World!"
```

Arrays

An array is a collection of elements of the same data type. In Java, arrays have a fixed size, which is defined when the array is created.

Array Declaration and Initialization

```
int[] numbers = new int[5]; // Declare an integer array of size 5
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 4;
numbers[4] = 5;
```

Array Initialization with Values

```
int[] numbers = {1, 2, 3, 4, 5}; // Initialize an array with values
```

Accessing Array Elements

You can access individual elements of an array using their index (starting from 0):

```
int firstElement = numbers[0]; // firstElement will be 1
```

Array Length

You can get the length (size) of an array using the `length` property:

```
int arrayLength = numbers.length; // arrayLength will be 5
```

Arrays of Objects

Arrays can hold objects of any class. For example, you can create an array of strings:

```
```

```
String[] names = {"Alice", "Bob", "Charlie"};
```

Multidimensional Arrays

Java supports multidimensional arrays, like 2D arrays:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Classes and Objects in Java

In Java, classes and objects are fundamental concepts used for creating and organizing code. Classes serve as blueprints for objects, defining their structure and behavior. Objects, on the other hand, are instances of classes, representing real-world entities. Here's an overview of classes and objects in Java with code examples:

Classes

A class is a template or blueprint for creating objects. It defines the attributes (fields) and methods that objects of that class will have.

```
// Example of a simple class definition
public class Car {
    // Fields (attributes)
    String brand;
    String model;
    int year;

    // Constructor
    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    // Method
    public void startEngine() {
        System.out.println("Starting the engine of " + brand + " " +
model);
    }
}
```

Creating Objects

You can create objects (instances) of a class using the `new` keyword followed by the class constructor:

```
Car myCar = new Car("Toyota", "Camry", 2023);
```

Objects

Objects are instances of classes and can have their own unique attributes and behavior.

Accessing Fields

You can access the fields (attributes) of an object using dot notation:

```
System.out.println("Brand: " + myCar.brand);
System.out.println("Model: " + myCar.model);
System.out.println("Year: " + myCar.year);
```

Invoking Methods

You can invoke methods defined in the class using the object:

```
myCar.startEngine();
```

Class Members

In addition to instance variables and methods, classes can also have static members (fields and methods) that belong to the class itself, not to individual objects.

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

You can access static members using the class name:

```
int sum = MathUtil.add(5, 3);
```

Object Initialization

When an object is created, its constructor is called to initialize its fields and perform any necessary setup. Constructors have the same name as the class and may have different parameter lists.

```
// Constructor example from the Car class
public Car(String brand, String model, int year) {
    this.brand = brand;
    this.model = model;
    this.year = year;
}
```

Inheritance

Java supports class inheritance, allowing you to create subclasses that inherit fields and methods from a superclass.

```
public class ElectricCar extends Car {
    int batteryCapacity;

    public ElectricCar(String brand, String model, int year, int
batteryCapacity) {
        super(brand, model, year);
        this.batteryCapacity = batteryCapacity;
    }

    public void chargeBattery() {
        System.out.println("Charging the battery of " + brand + " " +
model);
    }
}
```

Encapsulation

Encapsulation is the concept of hiding the internal implementation details of a class and providing controlled access to its data. It is achieved through access modifiers (public, private, protected) and getter/setter methods.

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

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

    public String getName() {
```

```
        return name;
    }

    public int getAge() {
        return age;
    }

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

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

Control Flow in Java

Control flow statements are used to control the execution of code based on certain conditions. In Java, control flow statements include if-else, switch, for, while, and do-while statements. Here's an overview of control flow statements in Java with code examples:

If-Else Statements

The if-else statement is used to execute a block of code if a condition is true, and another block of code if the condition is false.

```
int age = 25;

if (age >= 18) {
    System.out.println("You are an adult");
} else {
    System.out.println("You are a minor");
}
```

Switch Statements

The switch statement is used to execute different blocks of code based on the value of an expression.

```
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;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
}
```

For Loops

The for loop is used to execute a block of code a fixed number of times.

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

While Loops

The while loop is used to execute a block of code repeatedly as long as a condition is true.

```
int i = 1;

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

Do-While Loops

The do-while loop is similar to the while loop, except that the condition is checked at the end of the loop.

```
int i = 1;

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

Java Methods

A method is a block of code that performs a specific task. In Java, every method must be part of a class, and every program must have a `main()` method. Here's an overview of methods in Java with code examples:

Method Declaration and Invocation

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

    public static void main(String[] args) {
        int sum = add(5, 3);
        System.out.println(sum);
    }
}
```

Method Parameters

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

    public static void main(String[] args) {
        int sum = add(5, 3);
        System.out.println(sum);
```

```
    }
}
```

Method Return Values

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

    public static void main(String[] args) {
        int sum = add(5, 3);
        System.out.println(sum);
    }
}
```

Method Overloading

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

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

    public static void main(String[] args) {
        int sum1 = add(5, 3);
        double sum2 = add(5.5, 3.5);
        System.out.println(sum1);
        System.out.println(sum2);
    }
}
```

Method Scope

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

```
    public static void main(String[] args) {
        int sum = add(5, 3);
        System.out.println(sum);
    }
}
```

Method Overriding

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Animal();
        Animal animal2 = new Cat();
        Animal animal3 = new Dog();

        animal1.makeSound();
        animal2.makeSound();
        animal3.makeSound();
    }
}
```

Java Packages

A package is a collection of related classes and interfaces. Java provides a large number of packages containing classes and interfaces for common tasks. Here's an overview of packages in Java with code examples:

Package Declaration

```
package com.example;

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

Importing Packages

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

Creating Packages

```
package com.example;

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

Access Modifiers

```
package com.example;

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

```
    }
}
```

Static Imports

```
package com.example;

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

Java Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. In Java, exceptions are represented by classes. Here's an overview of exceptions in Java with code examples:

Exception Handling

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        try {
            System.out.println(numbers[10]);
        } catch (Exception e) {
            System.out.println("An error occurred");
        }
    }
}
```

Throwing Exceptions

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        try {
```

```
            System.out.println(numbers[10]);
        } catch (Exception e) {
            System.out.println("An error occurred");
        }
    }
}
```

Custom Exceptions

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        try {
            System.out.println(numbers[10]);
        } catch (Exception e) {
            System.out.println("An error occurred");
        }
    }
}
```

Java Interfaces

An interface is a collection of abstract methods and constants that form a common set of base rules/specifications for those classes that implement it. Here's an overview of interfaces in Java with code examples:

Interface Declaration

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

Implementing Interfaces

```
public class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
```

```
}

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

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Cat();
        Animal animal2 = new Dog();

        animal1.makeSound();
        animal2.makeSound();
    }
}
```

Extending Interfaces

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

public interface Mammal extends Animal {
    public void sleep();
}

public class Cat implements Mammal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }

    @Override
    public void sleep() {
        System.out.println("Zzz");
    }
}

public class Main {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.makeSound();
        cat.sleep();
    }
}
```

Java Enums

An enum is a special type of data type that contains a fixed set of constants. Here's an overview of enums in Java with code examples:

Enum Declaration

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

Enum Constants

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

Enum Methods

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

```java
public class Main {
    public static void main(String[] args) {
        Day day = Day.MONDAY;

        switch (day) {
            case MONDAY:
                System.out.println("Monday");
                break;
            case TUESDAY:
                System.out.println("Tuesday");
                break;
            case WEDNESDAY:
                System.out.println("Wednesday");
                break;
            case THURSDAY:
                System.out.println("Thursday");
                break;
            case FRIDAY:
                System.out.println("Friday");
                break;
            case SATURDAY:
                System.out.println("Saturday");
                break;
            case SUNDAY:
                System.out.println("Sunday");
                break;
        }
    }
}
```

```
# Java Annotations

An annotation is a special type of syntactic metadata that can be added
to Java source code. Here's an overview of annotations in Java with code
examples:

## Annotation Declaration

```java

public @interface MyAnnotation {
    String value();
}
```

Annotation Usage
```

```java
@MyAnnotation("Hello")

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

Built-in Annotations

```java
@Deprecated

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

Java Generics

Generics allow you to create classes, interfaces, and methods that take types as parameters (type parameters). Here's an overview of generics in Java with code examples:

Generic Classes

```java
public class GenericClass<T> {
    private T value;

    public GenericClass(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        GenericClass<String> genericClass = new GenericClass<>("Hello");
        System.out.println(genericClass.getValue());
    }
}
```

Generic Methods

```
public class GenericClass<T> {
    private T value;

    public GenericClass(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        GenericClass<String> genericClass = new GenericClass<>("Hello");
        System.out.println(genericClass.getValue());
    }
}
```

Bounded Type Parameters

```
public class GenericClass<T extends Number> {
    private T value;

    public GenericClass(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
```

```
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        GenericClass<Integer> genericClass = new GenericClass<>(10);
        System.out.println(genericClass.getValue());
    }
}
```

Java Reflection

Reflection is a feature in Java that allows you to inspect and modify code at runtime. Here's an overview of reflection in Java with code examples:

Getting Class Information

```
public class Main {
    public static void main(String[] args) {
        Class<?> cls = String.class;
        System.out.println(cls.getName());
        System.out.println(cls.getSimpleName());
        System.out.println(cls.getPackageName());
    }
}
```

Getting Class Modifiers

```
public class Main {
    public static void main(String[] args) {
        Class<?> cls = String.class;
        int modifiers = cls.getModifiers();

        System.out.println(Modifier.isPublic(modifiers));
        System.out.println(Modifier.isAbstract(modifiers));
        System.out.println(Modifier.isFinal(modifiers));
    }
}
```

Getting Class Constructors

```
public class Main {
    public static void main(String[] args) {
        Class<?> cls = String.class;
        Constructor<?>[] constructors = cls.getConstructors();

        for (Constructor<?> constructor : constructors) {
            System.out.println(constructor.getName());
        }
    }
}
```

Getting Class Fields

```
public class Main {
    public static void main(String[] args) {
        Class<?> cls = String.class;
        Field[] fields = cls.getFields();

        for (Field field : fields) {
            System.out.println(field.getName());
        }
    }
}
```

Getting Class Methods

```
public class Main {
    public static void main(String[] args) {
        Class<?> cls = String.class;
        Method[] methods = cls.getMethods();

        for (Method method : methods) {
            System.out.println(method.getName());
        }
    }
}
```

Creating Objects

```
public class Main {
```

```
    public static void main(String[] args) {
        Class<?> cls = String.class;

        try {
            String str = (String) cls.newInstance();
            System.out.println(str);
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Invoking Methods

```
public class Main {
    public static void main(String[] args) {
        Class<?> cls = String.class;

        try {
            String str = (String) cls.newInstance();
            Method method = cls.getMethod("toUpperCase");
            String result = (String) method.invoke(str);
            System.out.println(result);
        } catch (InstantiationException | IllegalAccessException |
NoSuchMethodException | InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

Accessing Fields

```
public class Main {
    public static void main(String[] args) {
        Class<?> cls = String.class;

        try {
            String str = (String) cls.newInstance();
            Field field = cls.getField("CASE_INSENSITIVE_ORDER");
            System.out.println(field.get(str));
        } catch (InstantiationException | IllegalAccessException |
NoSuchFieldException e) {
            e.printStackTrace();
        }
    }
}
```

Collections in Java

A collection is a group of objects of the same type. In Java, the Collection interface is the foundation of the collection framework. Here's an overview of collections in Java with code examples:

Collection Interface

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

List Interface

```
public interface List<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean addAll(int index, Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
```

```
    boolean equals(Object o);
    int hashCode();
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);
}
```

Set Interface

```
public interface Set<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean retainAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

Queue Interface

```
public interface Queue<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
```

```
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    boolean offer(E e);
    E remove();
    E poll();
    E element();
    E peek();
}
```

Deque Interface

```
public interface Deque<E> extends Queue<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    boolean offerFirst(E e);
    boolean offerLast(E e);
    E removeFirst();
    E removeLast();
    E pollFirst();
    E pollLast();
    E getFirst();
    E getLast();
    E peekFirst();
    E peekLast();
    boolean removeFirstOccurrence(Object o);
    boolean removeLastOccurrence(Object o);
    boolean add(E e);
    boolean offer(E e);
    E remove();
    E poll();
    E element();
    E peek();
```

```
    void push(E e);
    E pop();
    boolean remove(Object o);
    boolean contains(Object o);
    int size();
    Iterator<E> iterator();
    Iterator<E> descendingIterator();
}
```

Map Interface

```
public interface Map<K, V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
    boolean equals(Object o);
    int hashCode();
    interface Entry<K, V> {
        K getKey();
        V getValue();
        V setValue(V value);
        boolean equals(Object o);
        int hashCode();
    }
}
```

Collection Classes

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, java.io.Serializable {
    ArrayList();
    ArrayList(Collection<? extends E> c);
    ArrayList(int initialCapacity);
    void trimToSize();
    void ensureCapacity(int minCapacity);
    int size();
```

```
    boolean isEmpty();
    boolean contains(Object o);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    Object clone();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    E get(int index);
    E set(int index, E element);
    boolean add(E e);
    void add(int index, E element);
    E remove(int index);
    boolean remove(Object o);
    void clear();
    boolean addAll(Collection<? extends E> c);
    boolean addAll(int index, Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    boolean equals(Object o);
    int hashCode();
    ListIterator<E> listIterator(int index);
    ListIterator<E> listIterator();
    Iterator<E> iterator();
    List<E> subList(int fromIndex, int toIndex);
    void forEach(Consumer<? super E> action);
    Spliterator<E> spliterator();
    static <E> ArrayList<E> of();
    static <E> ArrayList<E> of(E e1);
    static <E> ArrayList<E> of(E e1, E e2);
    static <E> ArrayList<E> of(E e1, E e2, E e3);
    static <E> ArrayList<E> of(E e1, E e2, E e3, E e4);
    static <E> ArrayList<E> of(E e1, E e2, E e3, E e4, E e5);
    static <E> ArrayList<E> of(E e1, E e2, E e3, E e4, E e5, E e6);
    static <E> ArrayList<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E
e7);
    static <E> ArrayList<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7,
E e8);
    static <E> ArrayList<E> of(E e1, E e2,


public class LinkedList<E> extends AbstractSequentialList<E> implements
List<E>, Deque<E>, Cloneable, java.io.Serializable {

    LinkedList();
    LinkedList(Collection<? extends E> c);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
    void addFirst(E e);
    void addLast(E e);
    boolean contains(Object o);
    int size();
```

```
    boolean add(E e);
    boolean remove(Object o);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    E peek();
    E element();
    E poll();
    E remove();
    boolean offer(E e);
    boolean offerFirst(E e);
    boolean offerLast(E e);
    E peekFirst();
    E peekLast();
    E pollFirst();
    E pollLast();
    void push(E e);
    E pop();
    boolean removeFirstOccurrence(Object o);
    boolean removeLastOccurrence(Object o);
    ListIterator<E> listIterator(int index);
    Iterator<E> descendingIterator();
    Object clone();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    Spliterator<E> spliterator();
    static <E> LinkedList<E> of();
    static <E> LinkedList<E> of(E e1);
    static <E> LinkedList<E> of(E e1, E e2);
    static <E> LinkedList<E> of(E e1, E e2, E e3);
    static <E> LinkedList<E> of(E e1, E e2, E e3, E e4);
    static <E> LinkedList<E> of(E e1, E e2, E e3, E e4, E e5);
    static <E> LinkedList<E> of(E e1, E e2, E e3, E e4, E e5, E e6);
    static <E> LinkedList<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E
e7);
    static <E> LinkedList<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E
e7, E e8);
```

Example Implementation of Priority Queue

```
public class PriorityQueue<E> extends AbstractQueue<E> implements
java.io.Serializable {
    PriorityQueue();
    PriorityQueue(int initialCapacity);
    PriorityQueue(Comparator<? super E> comparator);
    PriorityQueue(int initialCapacity, Comparator<? super E>
comparator);
    PriorityQueue(Collection<? extends E> c);
    PriorityQueue(PriorityQueue<? extends E> c);
    PriorityQueue(SortedSet<? extends E> c);
    boolean add(E e);
    boolean offer(E e);
    E peek();
    E poll();
    Iterator<E> iterator();
    int size();
    boolean contains(Object o);
    boolean remove(Object o);
    void clear();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    Comparator<? super E> comparator();
    Spliterator<E> spliterator();
    boolean removeIf(Predicate<? super E> filter);
    Stream<E> stream();
    Stream<E> parallelStream();
    void forEach(Consumer<? super E> action);
    static <E> PriorityQueue<E> of();
    static <E> PriorityQueue<E> of(E e1);
    static <E> PriorityQueue<E> of(E e1, E e2);
    static <E> PriorityQueue<E> of(E e1, E e2, E e3);
    static <E> PriorityQueue<E> of(E e1, E e2, E e3, E e4);
    static <E> PriorityQueue<E> of(E e1, E e2, E e3, E e4, E e5);
    static <E> PriorityQueue<E> of(E e1, E e2, E e3, E e4, E e5, E e6);
    static <E> PriorityQueue<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E
e7);
    static <E> PriorityQueue<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E
e7, E e8);
    static <E> PriorityQueue<E> of(E... elements);
    static <E> PriorityQueue<E> copyOf(Collection<? extends E> coll);
    static <E> PriorityQueue<E> copyOf(Comparator<? super E> comparator,
Collection<? extends E> coll);
    static <E> PriorityQueue<E> copyOf(Comparator<? super E> comparator,
Iterable<? extends E> elements);
    static <E extends Comparable<? super E>> PriorityQueue<E>
copyOf(Iterable<? extends E> elements);
    static <E> PriorityQueue<E> copyOf(SortedSet<? extends E> ss);
    static <E> PriorityQueue<E> copyOf(PriorityQueue<? extends E> pq);
}
```

Java Streams

A stream is a sequence of elements that can be processed in a pipeline. In Java, streams are used to process collections of objects. Here's an overview of streams in Java with code examples:

Stream Creation

```
public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        Stream<String> stream1 = names.stream();
        Stream<String> stream2 = Stream.of("Alice", "Bob", "Charlie");
    }
}
```

Stream Operations

```
public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        Stream<String> stream1 = names.stream();
        Stream<String> stream2 = Stream.of("Alice", "Bob", "Charlie");

        stream1.forEach(System.out::println);
        stream2.forEach(System.out::println);
    }
}
```

Stream Filtering

```
public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        Stream<String> stream1 = names.stream();
        Stream<String> stream2 = Stream.of("Alice", "Bob", "Charlie");

        stream1.filter(name ->
name.startsWith("A")).forEach(System.out::println);
        stream2.filter(name ->
```

```
name.startsWith("A")).forEach(System.out::println);
    }
}
```

Stream Mapping

```
public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        Stream<String> stream1 = names.stream();
        Stream<String> stream2 = Stream.of("Alice", "Bob", "Charlie");

        stream1.map(String::toUpperCase).forEach(System.out::println);
        stream2.map(String::toUpperCase).forEach(System.out::println);
    }
}
```

Stream Collecting

```
public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        Stream<String> stream1 = names.stream();
        Stream<String> stream2 = Stream.of("Alice", "Bob", "Charlie");

        List<String> list1 = stream1.collect(Collectors.toList());
        Set<String> set1 = stream2.collect(Collectors.toSet());

        System.out.println(list1);
        System.out.println(set1);
    }
}
```

Stream Reducing

```
public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        Stream<String> stream1 = names.stream();
```

```
        Stream<String> stream2 = Stream.of("Alice", "Bob", "Charlie");

        Optional<String> result1 = stream1.reduce((a, b) -> a + " " +
b);
        String result2 = stream2.reduce("", (a, b) -> a + " " + b);

        System.out.println(result1.get());
        System.out.println(result2);
    }
}
```

Stream Parallelism

```
public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        Stream<String> stream1 = names.stream();
        Stream<String> stream2 = Stream.of("Alice", "Bob", "Charlie");

        stream1.parallel().forEach(System.out::println);
        stream2.parallel().forEach(System.out::println);
    }
}
```

Java Concurrency

Concurrency is the ability to run multiple tasks at the same time. In Java, concurrency is achieved through multithreading. Here's an overview of concurrency in Java with code examples:

Creating Threads

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Hello from thread");
        });

        thread.start();
    }
}
```

Thread States

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Hello from thread");
        });

        System.out.println(thread.getState());
        thread.start();
        System.out.println(thread.getState());
    }
}
```

Thread Priorities

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Hello from thread");
        });

        thread.setPriority(Thread.MAX_PRIORITY);
        thread.start();
    }
}
```

Thread Synchronization

```
public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
```

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

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(counter.getCount());
    }
}

public class Counter {
    private int count = 0;

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

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

Thread Communication

```


public class Main {
    public static void main(String[] args) {
        Message message = new Message("Hello");

        Thread thread1 = new Thread(() -> {
            String text = message.read();
            System.out.println(text);
        });

        Thread thread2 = new Thread(() -> {
            message.write("World");
        });

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


public class Message {
    private String text;
```

```
    public Message(String text) {
        this.text = text;
    }

    public synchronized String read() {
        return text;
    }

    public synchronized void write(String text) {
        this.text = text;
    }
}
```

Thread Pools

```
public class Main {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        executor.submit(() -> {
            System.out.println("Hello from thread");
        });

        executor.shutdown();
    }
}
```