# Module 1: Java Essentials and Core Concepts

## Object Oriented Programming Essential Features (Overview)

### Opening the Source code

1. Open Intellij
2. Choose open option
3. Choose pom.xml from the Source\OopsBasics folder
4. Open as project

### Topics

- Encapsulation
- Overloading
- Inheritance
- Overriding
- Abstract classes
- Polymorphism

### Encapsulation

The process of combining the data and operations over the data is referred to as encapsulation. This gives us some additional flexibility to restrict direct access to data and allow only operations to operate on it. This process of restricting direct access to data is referred as data hiding i.e. hidden from direct access.

These are object oriented concepts and they are not related to security in anyway.

In the following example there are four members present in the class Account

1. balance field - private
2. debit method  - public

3. credit method - public
4. showBalance method - public

Out of these balance is a private member and the rest are public members, which means that balance is private to the class Account and no outsider can access it. Where as other members are made public i.e. given public access.

Think about this, can you change the balance in your bank account directly? NO, you need to go through the credit and debit operations to modify it. And also balance is not hidden from you, it is only hidden from direct modifications.

You may have another question, balance being a private member can I access it in debit operation ?

Yes, because debit is also part of the same class definition and hence there is no restriction.

```
class Account {

    // Attributes

    // default is zero hence initialization not required.
    private double balance = 0;

    // Operations

    public void debit(double amount) {
     if (amount > 0 && amount <= balance) {
        balance -= amount;
     }
    }

    public void credit(double amount) {
     if (amount > 0) {
        balance += amount;
     }
    }

    public void showBalance() {
        System.out.println("Current balance - " + balance);
    }
}
```

```
public class EncapsulationEx1 {

    public static void main(String[] args) {
        Account a1;

        a1 = new Account();

        a1.credit(2000);
        a1.debit(1000);

        a1.showBalance();

        // balance is a private field in
        // the class Account, hence
        // directly referring to it through object
        // is invalid.

        // a1.balance = -2000; // invalid

        // debit operation validates the given amount
        // and rejects it if invalid. In the below case
        // because amount is not greater than 0 it won't
        // modify the balance.

        a1.debit(-2000);

        a1.showBalance();

    }

}
```

By restricting the direct access to data and allowing only operations to operate on it we can also ensure the modifications are valid by performing any validation checks.

Technical Note -

Does this mean operations are always public ?

Doesn't matter, we need to understand that all are members and any member which need to be restricted from outside the class definition should be private and which should be given access might be public.

E.g.

```
class Demo {
    private void f() {}
    public void g() {}
}

class Main {
    public static void main(String[] args) {
      Demo obj = new Demo();
      d.f(); // invalid as f() is private
      d.g(); // valid as it is public.
    }
}
```

You can think of private member as something that is written for the internal purpose of the class and not for outsiders.

## Overloading -

A method or a function is said to be overloaded when there are multiple defintions available for that function with different signatures. In the below example print method is said to be overloaded. i.e. there are three methods with name print.

You may get a question, when you call print which method gets invoked ?

Although there are three methods, there is a change in the parameter. i.e. first one accepts integer argument, where as second method is written for double and third is written for String.

Based on the type of value you pass the corresponding definition gets invoked.

e.g.

```
print(10) calls the print method which accepts integer.

print(10.2) calls the print method which accepts double

print("abc") calls the print method that accepts String.
```

Example -

```
class Sample {

    public void print(int a) {
        System.out.println("int - " + a);
    }

    public void print(double a) {
        System.out.println("double - " + a);
    }

    public void print(String a) {
        System.out.println("string - " + a);
    }

}

public class OverloadingEx1 {
    public static void main(String[] args) {
        Sample obj = new Sample();
        obj.print(10);
        obj.print(10.2);
        obj.print("abc");
    }
}
```

Technical Notes -

1) You are using System.out.println() for printing values, this itself is a best example for function overloading. The println method in the class PrintStream is overloaded for accepting different types, e.g. println(int), println(float) etc. It is a good idea, why? just think if you need to use printInt(..) printFloat(..) etc. how many names you need to remember? instead you are asked to remember one name i.e. println and use the same for printing different types of values, compiler determines which method to call based on the arguments passed to the function. The behavior is said to be polymorphic behavior (explained later) i.e. single element takes many forms.

2. Note then you can not overload by simply changing the return type, you must change at least one parameter.

e.g. INVALID - because parameters are of same type for both the methods i.e. (int, int), so when you say add(10, 20) it will match with both the methods and create ambiguity.

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

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

e.g. VALID - because there is at least one change in the parameter types, first one has (int, int) and second one has (int, double). If you call add(10, 20) it matches with first definition where as add(10, 20.0) will match with the second definition.

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

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

## Inheritance -

Allows us to extend an existing class inorder to reuse its functionality. Use "extends" keyword to inherit the properties of an existing class. A class that is used as the basis for inheritance is called a superclass or base class. A class that inherits from a superclass is called a subclass or derived classes.

Example -

In the following example ScientificCalc extends BasicCalc, here BasicCalc is referred as base class and ScientificCalc is referred as derived class or sub class. Through extension ScientificCalc inherits the properties of BasicCalc. So, operations present in BasicCalc are [add, sub] where as operations present in ScientificCalc are [add, sub, sin].

```
class BasicCalc {

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

```
    public int sub(int a, int b) {
        return a - b;
    }
}

// Inheritance
//                    is a
class ScientificCalc extends BasicCalc {

    public double sin(int deg) {
        double rad = deg * 3.14159 / 180;
        return Math.sin(rad);
    }
}

public class InheritanceEx1 {

    public static void main(String[] args) {

        BasicCalc bcalc1 = new BasicCalc();

        // Through BasicCalc reference
        // we can call add and sub methods.

        System.out.println( bcalc1.add(10, 20));
        System.out.println( bcalc1.sub(10, 20));


        ScientificCalc sCalc1 = new ScientificCalc();

        // Through ScientificCalc reference
        // we can call add, sub and sin methods.

        System.out.println( sCalc1.add(10, 20));
        System.out.println( sCalc1.sub(10, 20));
        System.out.println( sCalc1.sin(90) );

    }
}
```

Also to note that extends establishes an "is a" relation between classes, i.e. ScientificCalc is a BasicCalc.

What does this mean?

If some argument or LHS of the assignment is expecting BasicCalc object, we can replace it with ScientificCalc object as well.

e.g.

```
// bcalc2 is a reference variable(not object)
BasicCalc bcalc2;
bcalc2 = new ScientificCalc();
```

LHS is expecting an object of type BasicCalc. But the object supplied in RHS is of type ScientificCalc. This is also valid because it extends BasicCalc and hence there is an "is a" relation, i.e. ScientificCalc is a BasicCalc.

But there is a limitation, because bcalc2 is of type BasicCalc we can only refer to the members of BasicCalc. As compiler only knows type information but not the object, object on which operation is being performed is only known at the runtime.

```
int x = bcalc2.add(10, 20); // valid.
int y = bcalc2.sub(10, 20); // valid
double z = bcalc2.sin(90);  // Invalid.
```

To resolve this we need to do an explicit type cast inorder to make a call.

```
double z = ((ScientificCalc)bcalc2).sin(90);
                // valid but dangerous.
```

Here you are instructing the compiler to consider bcalc2 as ScientificCalc and then make a call.

General Example - Let us take some general example to help you understand this easily, if I say

```
Animal a = ........;
```

Here the expectation is that RHS will be an Animal. And through 'a' we can only refer to the properties of Animal. Can I say ?

```
a.bark();
```

NO.  And this is the state of the compiler, compiler doesn't know what the object is? compiler only knows the type information and not the object information, hence it declares this as INVALID. If I say

```
((Dog)a).bark();
```

Is this valid ?

May be !! because 'a' might point to any Animal, and it may be Dog as well, here you are instructing the compiler that "consider the a as Dog" and validate, then it say YES, it is VALID. As you suspected, if it is not a Dog then this type cast will result in ClassCastException hence you should be careful with this.

## Overriding -

Overriding is the process of redefining an existing method of the base class in the subclass. This is done in order to modify an existing definition with effect from the sub class. This is how we modify the existing behavior inherited from the base class.

Example -

In the following example class Base has two methods f() and g() and Derived extends Base, which means that it inherits the properties of Base, hence it too has two methods f() and g(). Having said that g() is redefined in Derived i.e. g() is said to be overridden in Derived. If you consider the Base object g() definition in Base is used where as if you consider the Derived object g() definition in Derived is used.

```
class Base {
    public void f() {
        System.out.println("f() in Base");
    }

    public void g() {
        System.out.println("g() in Base");
    }
}
```

```
// Derived inherits the functionality from Base.

class Derived extends Base {

    // overrides the definition g()
    public void g() {
        System.out.println("g() in Derived");
    }

    public void h() {
        System.out.println("h() in Derived");
    }
}

public class Main {

    public static void main(String[] args) {

        Base b1 = new Base();
        b1.f(); // in Base
        b1.g(); // in Base

        System.out.println();

        Derived d1 = new Derived();
        d1.f(); // in Base
        d1.g(); // in Derived
        d1.h(); // in Derived

        // Here b2 is a reference variable of type Base
        // Object is of type Derived (RHS).

        Base b2 = new Derived();
        b2.f(); // in Base
        b2.g(); // in Derived because Object is of type Derived.
    }
}
```

Whether you can refer to the member or not is based on type, which method to call is based on Object, this is referred as Dynamic Binding or Runtime Polymorphism (Explained later).

In the above example third case is a peculiar case, b2 is expecting Base object, but it was given Derived object. Through b2 you can only call the members of the Base directly, but which method is invoked depends on object, since g() is overridden in Derived the method in Derived is invoked during the call b2.g().

General Example -

Consider the statements,

```
Dog g = .....;
g.bark();
```

Is g.bark() a valid call over the Dog ? YES

Which Dog is barking ? depends on Object(the actual Dog instance pointed by g).

This process is called dynamic binding i.e. the actual behavior is determined at runtime i.e. when the object is known.

## Abstract classes -

Abstract in object oriented programming indicate incomplete in definition or not to be considered as concrete. You can not create object for an abstract class, it is meant to be a base class always. In Java if you declare a class as abstract  using abstract keyword. The opposite of this is a concrete class which can be instantiated.

abstract method - is the one with out definition i.e. only declaration of the method is present. If a class has an abstract method then the corresponding class should be declared abstract.

Subclasses which extend an abstract class must define all the abstract methods of that class to become concrete, otherwise they too remain abstract.

In the following example class Graphic is declared as abstract, it also has an abstract method draw(). As you can imagine Graphic doesn't know what to draw. The subclasses Line, Rectangle etc define the draw method and are concrete.

Example -

```
abstract class Graphic {

    protected int x1, y1;
    protected int x2, y2;

    public void setStart(int x, int y) {
        x1 = x;
        y1 = y;
    }

    public void setEnd(int x, int y) {
        x2 = x;
        y2 = y;
    }

    // abstract method i.e. only declaration, no definition
    public abstract void draw();

}

class Line extends Graphic {

    @Override
    public void draw() {
        System.out.printf(
            " draw line from (%d, %d) to (%d, %d) %n", x1, y1, x2, y2 );
    }
}

class Rectangle extends Graphic {

    @Override
    public void draw() {
        System.out.printf(
            " draw rectangle from (%d, %d) to (%d, %d) %n", x1, y1, x2, y2 );
    }
}

public class AbstractEx1 {

    // This method is applicable for all Graphics

    static void drawUtil(int x1, int y1, int x2, int y2, Graphic g) {
        g.setStart(x1, y1);
        g.setEnd(x2, y2);
```

```
            g.draw();
        }

        public static void main(String[] args) {

            // INVALID: You can not instantiate Graphic as it is abstract.
            // drawUtil(10, 10, 20, 20, new Graphic());

            // VALID
            drawUtil(30, 30, 40, 40, new Line());

            // VALID
            drawUtil(50, 50, 60, 60, new Rectangle());
        }
    }
```

You can see that the drawUtil method is expecting a Graphic object, and hence it is applicable for all Graphic sub classes. This way we can write the generalized code.

Q: What if I don't even declare draw() in Graphic ?

Ans: You can not invoke it through Graphic reference, i.e.

Graphic g = .....;

in order to call

g.draw();

draw should be a member in Graphic.

## Polymorphism -

The behavior of an identifier is said to be polymorphic if it takes many forms, i.e. it has multiple definitions and the definition is picked based on the context. In the following example the behaviour of the print method is said to be polymorphic.

If you supply integer then print(int) is selected. If you supply double then print(double) is selected etc.

Example -

```
class Sample {

    public void print(int a) {
        System.out.println("int value - " + a);
    }

    public void print(double a) {
        System.out.println("double value - " + a);
    }

    public void print(String a) {
        System.out.println("String value - " + a);
    }
}

class Main {

    public static void main(String[] args) {
        Sample s = new Sample();
        s.print(10);
        s.print(10.2);
        s.print("abc");
    }
}
```

**Static binding vs Dynamic binding -**

Binding actually stands for linking, in other words linking the call with the corresponding definition. Suppose you are making a method call, compiler should actually link it to the actual definition. And this is what we call binding. Static binding is also called as compile time binding or early binding. In this case compiler itself determines which method should be invoked. Whereas in case of dynamic binding the actual definition could not determined till runtime. And hence it is also called as runtime binding or runtime polymorphism or late binding.

Example -

```
class Base {
    public static void f() {
        System.out.println("f() in Base");
    }
```

```
    public final void g() {
        System.out.println("g() in Base");
    }

    public void h() {
        System.out.println("h() in Base");
    }
}

class Derived extends Base {

    @Override
    public void h() {
        System.out.println("h() in Derived");
    }
}

class Main {

    public static void main(String[] args) {

        // Since f() is static
        Base.f(); // f() in Base (1)

        Base b1 = new Base();
        b1.g(); // g() in Base (2)
        b1.h(); // h() in Base (3)

        Base b2 = new Derived();
        b2.g(); // g() in Base
        b2.h(); // h() in Derived
    }
}
```

(1) & (2)

In case of static method call, we can term this as static binding. Static methods can not be overridden and hence compiler can pick the exact method definition.

Now in case of g() method call, g() is a final method and it can not be overridden. Hence it is also a candidate for static binding because compiler knows that this is the method in Base which should be linked with.

In both the cases compiler an directly bind the call with the exact function. Hence this is called static binding or compile time binding.

(3)

When it comes to h() it is neither static not final. And hence the actual definition could not be determined unless we know the object. Hence Dynamic Binding is used here

i.e. the binding in other words linking to the actual definition is delayed till runtime until the object is actually known. In this case if the object is a Base object then the definition in Based is invoked. If the object is a Derived object then the definition in Derived is invoked.

```
// Since f() is static
Base.f(); // f() in Base (1)

Base b1 = new Base();
b1.g(); // g() in Base (2)
b1.h(); // h() in Base (3)

Base b2 = new Derived();
b2.g(); // g() in Base
b2.h(); // h() in Derived
```

## Summary

Object-Oriented Programming (OOP) relies on four core principles: encapsulation, inheritance, abstraction, and polymorphism. These principles enable modular, reusable, and maintainable code by modeling real-world entities as objects.

1. **Encapsulation**: Encapsulation bundles data and methods within a class, restricting direct access to internal details using access modifiers like private or public. This protects object state and exposes only necessary interfaces.

2. **Inheritance**: Allows a new class (subclass) to inherit properties and behaviors from an existing class (superclass), promoting code reuse.

3. **Abstraction**: Abstraction hides complex implementation details behind simple interfaces, letting users interact with high-level methods without knowing internals. Abstract classes or interfaces enforce this.

4. **Polymorphism**: Polymorphism enables objects of different classes to be treated uniformly through a common interface, often via method overriding or overloading. This supports flexibility, like different shapes sharing a "draw" method.

By leveraging these OOP principles, developers can create software that is easier to understand, extend, and maintain over time.

# Collections

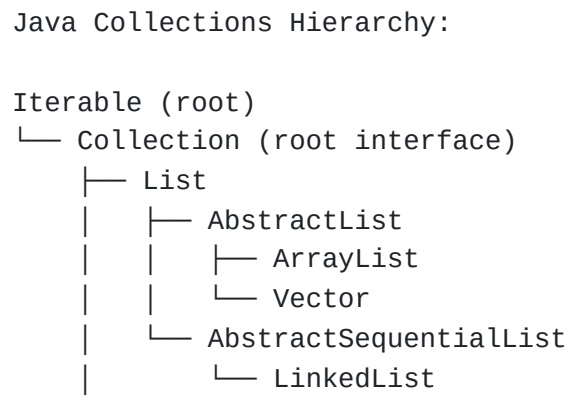
Java Collections Framework provides resizable data structures for storing and manipulating groups of objects.

## Collection Interface

Collection is the root interface for most collection classes, defining basic operations like add, remove, size, and iteration. It serves as the foundation for List and Set.

Some of the mainly used methods in Collection interface are:

- `boolean add(E e)` : Adds an element to the collection.
- `boolean remove(Object o)` : Removes a single instance of the specified element from the collection, if present.
- `int size()` : Returns the number of elements in the collection.
- `boolean isEmpty()` : Checks if the collection is empty.
- `boolean contains(Object o)` : Checks if the collection contains the specified element.
- `Iterator<E> iterator()` : Returns an iterator over the elements in the collection.



```
    ├── Set
    │   ├── AbstractSet
    │   │   ├── HashSet
    │   │   └── TreeSet
    │   └── LinkedHashSet
    └── Queue
        ├── AbstractQueue
        │   └── PriorityQueue
        └── Deque
            └── LinkedList (also implements List)

Map (separate hierarchy)
├── AbstractMap
│   ├── HashMap
│   ├── LinkedHashMap
│   └── TreeMap (also implements SortedMap)
```

## List Interface

List maintains insertion order and allows duplicate elements with indexed access. ArrayList and LinkedList implement are some wellknown classes which implement the List interface.

Some of the mainly used methods in List interface are:

- `void add(int index, E element)` : Inserts the specified element at the specified position in the list.
- `E get(int index)` : Returns the element at the specified position in the list.
- `E remove(int index)` : Removes the element at the specified position in the list.
- `int indexOf(Object o)` : Returns the index of the first occurrence of the specified element in the list, or -1 if not found.
- `List<E> subList(int fromIndex, int toIndex)` : Returns a view of the portion of the list between the specified indices.

### ArrayList (Class)

ArrayList uses dynamic arrays for fast random access but slower insertions/deletions in the middle. It grows automatically as elements are added.

```java
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(10);
        numbers.add(15);
        numbers.add(20);
        System.out.println(numbers); // [10, 15, 20]
        System.out.println(numbers.get(0)); // 10
    }
}
```

### LinkedList (Class)

LinkedList uses doubly-linked nodes for efficient insertions/deletions anywhere but slower random access. It also implements Queue and Deque interfaces.

Some of the mainly used methods in LinkedList class are:

- `void addFirst(E e)` : Inserts the specified element at the beginning of the list.
- `void addLast(E e)` : Appends the specified element to the end of the list.
- `E getFirst()` : Returns the first element in the list.
- `E getLast()` : Returns the last element in the list.
- `E removeFirst()` : Removes and returns the first element from the list.
- `E removeLast()` : Removes and returns the last element from the list.

```java
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> animals = new LinkedList<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.addFirst("Lion");
        System.out.println(animals); // [Lion, Dog, Cat]
    }
}
```

## Set Interface

Set stores unique elements without duplicates. HashSet, LinkedHashSet, and TreeSet provide different ordering guarantees.

Some of the mainly used methods in Set interface are:

- `boolean add(E e)` : Adds the specified element to the set if it is not already present.
- `boolean remove(Object o)` : Removes the specified element from the set if it is present.
- `boolean contains(Object o)` : Checks if the set contains the specified element.
- `int size()` : Returns the number of elements in the set.
- `void clear()` : Removes all elements from the set.

### HashSet (Class)

HashSet stores elements in hash table with no guaranteed order and constant-time performance for basic operations. Duplicates are automatically prevented.

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> cars = new HashSet<>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("BMW"); // Duplicate ignored
        cars.add("Ford");
        System.out.println(cars); // [Volvo, BMW, Ford] (order not guaranteed)
    }
}
```

While using hashed collections, it's important to ensure that the objects stored in them properly override the `hashCode()` and `equals()` methods. This ensures that the uniqueness constraint of the Set is maintained correctly.

```
    import java.util.HashSet;

    class Fruit {
        private String name;
```

```
        public Fruit(String name) {
            this.name = name;
        }

        @Override
        public int hashCode() {
            return name.hashCode();
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            Fruit fruit = (Fruit) obj;
            return name.equals(fruit.name);
        }
    }

    class Main {
        public static void main(String[] args) {
            HashSet<Fruit> fruitSet = new HashSet<>();
            fruitSet.add(new Fruit("Apple"));
            fruitSet.add(new Fruit("Banana"));
            fruitSet.add(new Fruit("Apple")); // Duplicate based on name
            System.out.println(fruitSet.size()); // Output: 2
        }
    }
```

### LinkedHashSet (Class)

LinkedHashSet maintains insertion order using hash table and linked list while ensuring uniqueness. It offers predictable iteration order.

```
import java.util.LinkedHashSet;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<String> cars = new LinkedHashSet<>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("BMW"); // Duplicate ignored
        System.out.println(cars); // [Volvo, BMW, Ford]
```

```
    }
}
```

## SortedSet Interface

SortedSet maintains elements in ascending order with additional methods like first() and last(). TreeSet implements this interface.

### TreeSet (Class)

TreeSet uses red-black tree for sorted storage, automatic uniqueness, and log(n) performance. Elements must be comparable or provide Comparator.

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<String> cars = new TreeSet<>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("BMW"); // Duplicate ignored
        System.out.println(cars); // [BMW, Ford, Volvo]
    }
}
```

Custom objects stored in a TreeSet must implement the Comparable interface or be provided with a Comparator to define their natural ordering.

```
    import java.util.TreeSet;

    class Person implements Comparable<Person> {
        private String name;
        private int age;

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

        @Override
```

```
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age); // Sort by age
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

class Main {
    public static void main(String[] args) {
        TreeSet<Person> people = new TreeSet<>();
        people.add(new Person("A", 30));
        people.add(new Person("B", 25));
        people.add(new Person("C", 35));
        System.out.println(people); // Output: [B (25), A (30), C (35)]
    }
}
```

The same applies when using a Comparator:

```
import java.util.Comparator;
import java.util.TreeSet;

class Person {
    private String name;
    private int age;

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

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

class AgeComparator implements Comparator<Person> {

    @Override
    public int compare(Person p1, Person p2) {
```

```
            return Integer.compare(p1.age, p2.age); // Sort by age
        }

    }

    class Main {
        public static void main(String[] args) {
            TreeSet<Person> people = new TreeSet<>(new AgeComparator());
            people.add(new Person("A", 30));
            people.add(new Person("B", 25));
            people.add(new Person("C", 35));
            System.out.println(people); // Output: [B (25), A (30), C (35)]
        }
    }
```

## Map Interface

Map stores key-value pairs with unique keys. HashMap, LinkedHashMap, and TreeMap offer different performance and ordering characteristics.

### HashMap (Class)

HashMap provides fast key-based access using hash table with no order guarantee. Keys must be unique; null keys/values allowed.

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> fruits = new HashMap<>();
        fruits.put("Apple", 3);
        fruits.put("Banana", 5);
        fruits.put("Apple", 4); // Overwrites previous value
        System.out.println(fruits.get("Apple")); // 4
    }
}
```

### LinkedHashMap (Class)

LinkedHashMap maintains insertion order (or access order) while providing hash table performance. Ideal for caches requiring predictable iteration.

```java
import java.util.LinkedHashMap;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> numbers = new LinkedHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println(numbers); // {One=1, Two=2, Three=3}
    }
}
```

## SortedMap Interface

SortedMap maintains keys in ascending order with additional methods like firstKey() and lastKey(). TreeMap implements this interface.

### TreeMap (Class)

TreeMap uses red-black tree for sorted keys and log(n) operations. Keys must implement Comparable or provide Comparator.

```java
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("Second", 2);
        numbers.put("First", 1);
        numbers.put("Third", 3);
        System.out.println(numbers); // {First=1, Second=2, Third=3}
    }
}
```

## Iterator Interface

Iterator provides a way to traverse collections sequentially without exposing the underlying structure. It supports safe element removal during iteration.

Some of the mainly used methods in Iterator interface are:

- `boolean hasNext()` : Returns true if there are more elements to iterate over.
- `E next()` : Returns the next element in the iteration.
- `void remove()` : Removes the last element returned by the iterator from the underlying collection.

Example of using Iterator:

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> colors = new ArrayList<>();
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        Iterator<String> iterator = colors.iterator();
        while (iterator.hasNext()) {
            String color = iterator.next();
            System.out.println(color);
            if (color.equals("Green")) {
                iterator.remove(); // Safely remove "Green"
            }
        }

        System.out.println("After removal: " + colors); // [Red, Blue]
    }
}

For each loop can also be used as a simpler alternative to Iterator:

```java

import java.util.ArrayList;

public class ForEachExample {
    public static void main(String[] args) {
        ArrayList<String> colors = new ArrayList<>();
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        for (String color : colors) {
```
```

```
            System.out.println(color);
        }
    }
}
```

### Shortcuts -

- List - Allows duplicates
- Set - No duplicates
- Hash - Uses hashing ( insertion order not preserved)
- Linked - Insertion order preserved.
- Tree - Sorted

## Solid Principles

SOLID principles are five fundamental guidelines for object-oriented design that promote maintainable, scalable, and flexible code. Introduced by Robert C. Martin, they help avoid common pitfalls like rigid or fragile systems.

The five SOLID principles are:

### S - Single Responsibility Principle

A class should have one and only one reason to change.

```
class AccountService {
    private AccountRepository accountRepository;
    private NotificationService notificationService;

    public AccountService(AccountRepository accountRepository,
NotificationService notificationService) {
        this.accountRepository = accountRepository;
        this.notificationService = notificationService;
    }

    public void createAccount(User user) {
        accountRepository.save(user);
        notificationService.sendWelcomeEmail(user);
```

```
    }
}
```

How Single Responsibility Principle Is Applied Here?

- AccountService: Coordinates the high-level process of creating an account — it doesn't concern itself with how data is stored or how emails are sent.
- AccountRepository: Takes care solely of database save operations.
- NotificationService: Manages email content and delivery.

If you change your database logic or email provider, you don't have to modify the AccountService class — only the respective component.

Key benefits:

- Maintainability: Each class can be modified independently without affecting others.
- Reusability: NotificationService can be reused for other notifications, and AccountRepository for other database operations.
- Testability: You can mock dependencies (e.g., test AccountService without a real database).

## O - Open Closed Principle

The Open–Closed Principle states that a software module (class, function, etc.) should be open for extension but closed for modification. That means you should be able to add new behavior to existing code without changing the already tested and working code.

```
interface ValueComparator {
    int compare(int value1, int value2);
}

class ArrayUtil {
    public static final void sort(int a[], ValueComparator comparator) {
        for(int i = 0 ; i < a.length; i++) {
            for( int j = i + 1; j < a.length; j++) {
                if (comparator.compare(a[i], a[j]) > 0) {
                    int temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
```

```
            }
        }
    }
```

In the ArrayUtil class, the sort() method sorts an array — but importantly, it depends on the abstract interface ValueComparator, not on any specific comparison logic.

We can pass different comparator implementations (ascending, descending, even custom comparisons) without changing a single line of code in ArrayUtil

How It Follows the Open–Closed Principle?

- Closed for modification: The ArrayUtil class is complete and does not need editing if we need new sorting orders.
- Open for extension: We can extend functionality by adding new classes that implement ValueComparator.

For example:

```
class AscComparator implements ValueComparator {
    public int compare(int value1, int value2) {
        return value1 - value2;
    }
}

class DescComparator implements ValueComparator {
    public int compare(int value1, int value2) {
        return value2 - value1;
    }
}
```

Key Benefits:

- Flexibility: New comparison strategies can be added without touching existing code.
- Maintainability: Existing code remains stable and tested, reducing the risk of introducing bugs.

## L - Liskov Substitution Principle

Liskov Substitution Principle (LSP) — the "L" in SOLID. LSP states that objects of a superclass should be replaceable by objects of a subclass without altering the correctness of the program. Subclasses must honor the base class contract without introducing unexpected behavior.

```
class Vehicle {
    public void start() { System.out.println("starting a vehicle"); }
    public void stop() { System.out.println("stopping a vehicle"); }
}

class Bike extends Vehicle {
    public void start() { System.out.println("starting a bike"); }
    public void stop() { System.out.println("stopping a bike"); }
}

class Car extends Vehicle {
    public void start() {
        System.out.println("starting a car");
    }
    public void stop() { System.out.println("stopping a car"); }
}

class VechicleDemo {
    static void testDrive(Vehicle vehicle) {
        vehicle.start();  // Expects this to work reliably
        vehicle.stop();
    }
}
```

The testDrive() method works with any Vehicle. It calls:

```
testDrive(new Vehicle());  // "starting a vehicle" → "stopping a vehicle"
testDrive(new Bike());     // "starting a bike" → "stopping a bike"
testDrive(new Car());      // "starting a car" → "stopping a car"
```

This example follows LSP because:

- Bike and Car substitute perfectly for Vehicle in testDrive().
- Both override methods but preserve expected behavior — start() succeeds, stop() succeeds.
- testDrive() behaves predictably regardless of the concrete type passed.

## I - Interface Segregation Principle

Interface Segregation Principle (ISP) — the "I" in SOLID. ISP states that clients should not be forced to depend on interfaces they do not use. Instead of one large, "fat" interface, create multiple small, specific interfaces.

```
    interface VegRestaurant {
        void vegMeals();
    }

    interface NonVegRestaurant {
        void nonVegMeals();
    }

    class ABCVegRestaurant implements VegRestaurant {  // Only veg interface
        public void vegMeals() {
            System.out.println("provide veg meals");
        }
    }

    class XYZRestaurant implements VegRestaurant, NonVegRestaurant {  // Both
interfaces
        public void vegMeals() {
            System.out.println("provide veg meals");
        }
        public void nonVegMeals() {
            System.out.println("provide non veg meals");
        }
    }
```

This design follows ISP because:

- Small, focused interfaces: Each interface has one specific role (veg meals vs non-veg meals).
- No forced implementation: ABCVegRestaurant (pure veg restaurant) is not forced to implement non-veg methods.
- Client-specific: Classes implement only what they need.

Bad design (fat interface):

```
    interface Restaurant {
        void vegMeals();
```

```
        void nonVegMeals();
    }

    class ABCVegRestaurant implements Restaurant {  // Forced to implement
non-veg
        public void vegMeals() {
            System.out.println("provide veg meals");
        }
        public void nonVegMeals() {
            throw new UnsupportedOperationException("No non-veg meals");
        }
    }
```

Key Benefits:

- Flexibility: Easier to change or extend specific parts of the system.
- Maintainability: Changes in one interface do not affect unrelated clients.
- Clearer contracts: Each interface clearly defines its purpose.
- Segregated interfaces (VegRestaurant, NonVegRestaurant).
- No forced dependencies — each class implements only relevant methods.

The result is loosely coupled, maintainable code where classes stay true to their actual responsibilities.

## D - Dependency Inversion Principle

Dependency Inversion Principle (DIP) — the "D" in SOLID. DIP states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details — details depend on abstractions.

```
    class MyMessenger {  // HIGH-LEVEL
        ProtocolHandler handler;  // ← DEPENDS ON ABSTRACTION (not concrete
classes)

        public MyMessenger(String protocol) {
            handler = ProtocolHandlerFactory.getProtocolHandler(protocol);  //
Factory provides concrete impl
        }

        public void send(String to, String message) {
            handler.sendMessage(...);  // Uses abstraction
```

```
        }
    }

    interface ProtocolHandler {  // ABSTRACTION
        void sendMessage(String message);
    }
```

BAD DESIGN (VIOLATES DIP):

```
MyMessenger → TCPProtocolHandler  (direct concrete dependency - VIOLATION)
```

DIP-compliant flow (GOOD):

```
MyMessenger → ProtocolHandler ← TCPProtocolHandler, UDPProtocolHandler
```

In this design:

- MyMessenger depends on the ProtocolHandler interface, not on concrete implementations.
- Concrete protocol handlers implement ProtocolHandler, allowing MyMessenger to work with any protocol without modification.

Key Benefits:

- Flexibility: Easily switch or add new protocols without changing MyMessenger.
- Maintainability: High-level logic is decoupled from low-level details.
- Testability: You can mock ProtocolHandler for unit testing MyMessenger.

## Summary

SOLID principles are five guidelines for object-oriented design that create maintainable, scalable code. Each principle from the provided examples demonstrates separation of concerns, flexibility, and reliability.

- Single Responsibility (SRP) : Classes should have one reason to change.
- Open-Closed (OCP) : Modules open for extension, closed for modification.
- Liskov Substitution (LSP) : Subclasses replaceable for base classes without breaking behavior.

- Interface Segregation (ISP) : Clients depend only on needed interfaces.
- Dependency Inversion (DIP) : High-level modules depend on abstractions.

# Threading and Concurrency

## Introduction

In the previous modules we have seen how to write sequential programs, where each statement is executed one after another. However, in real world applications, there are multiple tasks that need to be performed simultaneously. For example, in a web server, it needs to handle multiple client requests at the same time. To achieve this, we need to use threading and concurrency concepts in programming.

## True Parallelism vs Logical Parallelism

True parallelism is achieved by assigning tasks to individual CPUs or Processors. This is possible through multicore processors or executing the tasks using multiple CPUs.

If there is one CPU and you want to perform multiple tasks in parallel, then CPU will be shared across those tasks for some stipulated time interval, which stands for interleaved execution, and this way of executing the tasks is known as logical parallelism or psuedo parallelism.

In case of logical parallelism, lets assume there are two tasks T1 and T2, when executed in parallel and using one CPU, CPU is switched between T1 and T2 i.e. CPU executed T1 for some stipulated time interval and then switched to T2. Once T2's time slice is completed then it is switched back to T1 and starts from where it stops.

## Threads

A thread is a lightweight process that can run concurrently with other threads within the same program. Each thread has its own call stack, program counter, and local variables, but shares the same memory space with other threads in the same process. Java provides built-in support for threading through the `java.lang.Thread` class and the `java.lang.Runnable` interface. You can create a thread by extending the `Thread` class or implementing the `Runnable` interface.

## Approach1 : Extending the Thread class

In the below example Task is a Thread(explained later), and run is the entry point of the thread execution where it starts printing 1500 T's.

main() runs in Thread i.e. the Main thread which is started by the JVM.

Note In the main method we are not calling doTask directly, instead we are using the start() method of the Thread class, which runs the Task using a separate Thread.

```
class Task extends Thread {

    // Thread execution begins here.
    public void run() {
        // performs the task i.e. prints 1500 T's
        doTask();
    }

    public void doTask() {
        for(int i=1; i <= 1500; i++) {
            System.out.print("T");
        }
    }
}

public class Main {

    // Runs with in the Main thread started by JVM.
    public static void main(String[] args) {

        Task t1 = new Task();

        // Starts a separate Thread using the
        // the start method of the Thread class.
        t1.start();

        // runs in the Main thread and prints 1500 M's
        for(int i=1; i <= 1500; i++) {
            System.out.print("M");
        }
    }
}
```

Here main() and Task are run using two separate threads, which means they are executed in parallel (logical parallelism in case of single CPU) and hence you will see output like MMMTTTMMMTTT...

## Approach2 : Implementing the Runnable interface

Another way to create a thread is by implementing the `Runnable` interface. This approach is preferred when your class needs to extend another class, as Java does not support multiple inheritance. In this approach, you create a class that implements the `Runnable` interface and override the `run()` method. Then, you create a `Thread` object and pass an instance of your `Runnable` class to the `Thread` constructor.

```
class Task implements Runnable {
    // Thread execution begins here.
    public void run() {
        // performs the task i.e. prints 1500 T's
        doTask();
    }

    public void doTask() {
        for(int i=1; i <= 1500; i++) {
            System.out.print("T");
        }
    }
}

public class Main {

    // Runs with in the Main thread started by JVM.
    public static void main(String[] args) {

        Task task = new Task();

        // Creating a Thread and passing the
        // Runnable object to it.
        Thread t1 = new Thread(task);

        // Starts a separate Thread using the
        // the start method of the Thread class.
        t1.start();

        // runs in the Main thread and prints 1500 M's
        for(int i=1; i <= 1500; i++) {
```

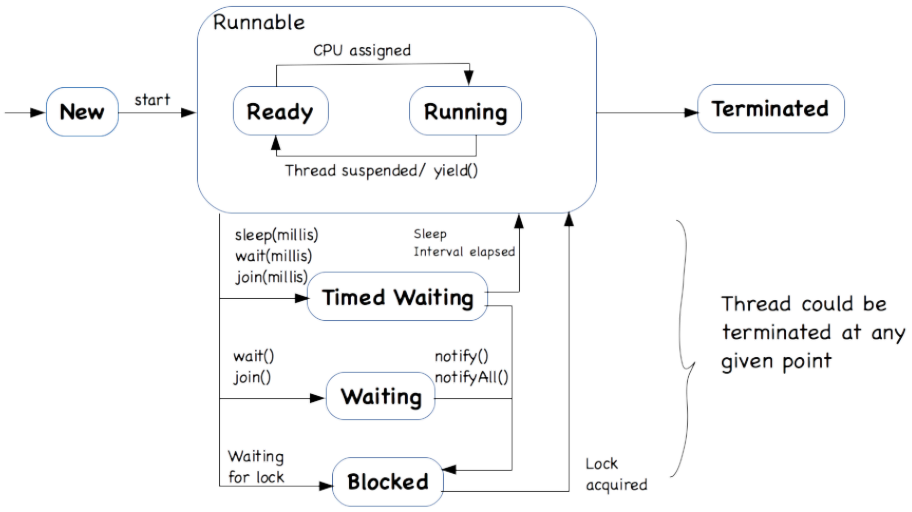
```
            System.out.print("M");
        }
    }
}
```

Both approaches achieve the same result, but implementing the `Runnable` interface is generally preferred for better flexibility and to avoid the limitations of single inheritance in Java.

## Thread States

A thread can be in one of the following states:

illustrated in the diagram below:




- NEW: A thread that is created but not yet started is in this state.
- RUNNABLE: A thread executing in the Java virtual machine is in this state, internally we can think of it as a combination of two sub states Ready and Running, i.e. when you start the thread it comes to Ready state and wait for the CPU, and if CPU is allocated then it goes into Running state. Once allocated CPU time is completed, in other words when the Thread schedular suspends the thread then it goes back to the Ready state and waits for its turn again.
- BLOCKED: A thread is blocked if it is waiting for a monitor lock is in this state. Refer synchronized methods and blocks.
- WAITING: A thread that is waiting indefinitely for another thread to perform a particular action is in this state. Refer wait(), join()

- TIMED_WAITING: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state. Refer wait(millis), join(millis)
- TERMINATED: A thread that has exited i.e. it has either completed its task or terminated in the middle of the execution.

## Thread Synchronization

Thread synchronization is used to solve concurrency problems that exist in parallel processing. Concurrency problem exist when more than one thread is accessing the same shared resource.

E.g.

1. More than one transaction is being performed on the same account
2. Multiple resources are booking tickets for the same train from different locations. etc.

### Synchronization in Java Threads

It can be achieved through 1) Synchronized methods 2) Synchronized block 3) Locks (out of scope)

#### synchronized method

Consider the class Sample

```
class Sample {
    synchronized void f() {...}
}
```

Consider Three threads T1, T2, T3 and two objects for Sample they are A,B.

```
T1  ..........A.f();  locks A and proceeds
T2  ..........B.f();  locks B and proceeds
T3  ..........A.f();  wait till T1 unlocks A.
```

To run a synchronized method object must be locked.

#### synchronized block

When synchronization is not required for the entire method i.e only certain part of the code must be synchronized then we use synchronized block.

```
synchronized( object ) {
  // operations over the object
}
```

The above code is executed only after obtaining lock over the object.

#### Thread Safe Code or Re-entrant code:-

When a code block is safe from concurrency problems then the code is referred as thread safe or re-entrant.In case of the below operation incr() operation is considered as thread safe or re-entrant.

Example -

In the below example try removing synchronized keyword before the incr() operation and check the result. You will find inconsistent result. By making the method synchronized, we are forcing the thread to lock the object before performing the incr() operation. Though control is intentionally passed to other thread, other thread won't be able to proceed with the operation as it need to first lock (obj) before proceeding forward.

i.e. lets assume t1 locks obj then t2 should wait till t1 releases the lock, hence object is modified by only one thread at a time and you will see consistent results.

```
class Sample {

    private int x;

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    /*
     * Try removing synchronized.
     */
    public synchronized void incr() {
```

```
        // read the value of x.
        int y = getX();

        // Increment the value
        y++;

        // Just assume if control is switched to
        // some other thread and it too looks at
        // the old value of x and proceeds with
        // modification. Such scenarios lead to
        // in consistent results.
        // To simulate such scenarios lets us just
        // pass the control to some other thread.

            // with sleep this thread will go to blocked state
            // for the given time interval, hence other thread
            // will get a chance.
        try { Thread.sleep(1); } catch(Exception e) {}

        // set x to new value.
        setX(y);
    }
}

class MyThread extends Thread {

    Sample obj;

    public MyThread(Sample obj) {
        this.obj = obj;
    }

    public void run() {
        obj.incr();
    }

}


public class Main {

    public static void main(String[] args) {

    Sample obj = new Sample();
    obj.setX(10);
```

```
    // In this case both the threads t1 and t2
    // are sharing the same Sample object obj.
    // Both the threads will try to perform the
    // increment operation simultaneously.

    MyThread t1 = new MyThread(obj);
    MyThread t2 = new MyThread(obj);

    t1.start();
    t2.start();

    // Here main thread called the join operation
    // on t1 and t2. join() operations waits for
    // thread to complete before returning.
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println( obj.getX() );
    }

}
```

**Synchronizing static data operations -**

Is the below increment() operation thread-safe ?

```
class Sample {

    static int a = 5;
    int b = 10;

    public synchronized void increment( ) {
        a++;
        b++;
    }

    // ....
}
```

Answer is NO. Because we made it synchronized we may think that it is thread-safe but it is not. And it is because of the static variable a. Lets assume that there are two objects for Sample, in that case both of them will share the same copy of a because it is a class member, where as they get different copies of b, because b is non static i.e. the instance member and each instance will get a separate copy of b.

Assume Thread 1 invoked the increment method over the first object and Thread 2 invoked the increment method over the second object. Because the increment() method is non-static and it is synchronized, object should be locked before getting into the method.

Here Thread1 locks the first object and gets in and also Thread 2 locks the second object and gets in, because both are different objects and hence both the thread acquire locks and they both proceed with the operation.

You can see that b++ is not having any issues, because both the threads are operating on different copies of b, but what about a++ it is still not thread safe.

Solution 1 -

```
class Sample {

    static int a = 5;
    int b = 10;

    public  void increment( ) {
        // lock the Class object before modifying
        // static content.
        synchronized(Sample.class) {
            a++;
        }

        // lock the object before modifying
        // instance members.
        synchronized(this) {
            b++;
        }
    }

    // ....
}
```

In this case the increment() operation is thread-safe, because for modifying the static member 'a' we are locking the class object, Sample.class returns a reference to class object and synchronized block will acquire lock over the object and then proceed forward with the operation. And for b++ we are locking the current object using the this reference. And hence both the operations are now thread-safe, as we properly locked the corresponding object before modification.

Solution 2 -

```
class Sample {

    static int a = 5;
    int b = 10;

    // This method is static and hence it locks the Class object.
    public static synchronized void incrementA( ) {
        a++;
    }

    // This method is non static and hence it locks the object
    // on which it is invoked.
    public synchronized void increment( ) {
        incrementA();
        b++;
    }

    // ....

}
```

Create a static method for incrementing a and declare it as synchronized, so that when the thread enters this method it locks the class object. Just remember that whenever you are modifying the static content in a multi-threaded environment you should lock the class object to make your code thread-safe.

#### Issue with synchronized methods - (Out of scope, but important to understand)

Synchronized methods doesn't always solve concurrency problems. Lets consider a simple List class and assume that the size() and add() operations are synchronized

```
class List {
    ...
    public synchronized int size(){
    ...
    }

    public synchronized void add(Object value) {
    ....
    }
}
```

You can think that there is no concurrency issue here. But, lets analyze a simple scenario consider that list should not contain more than 5 elements, and assume that list is already having 4 elements and two threads are trying to insert an element into the list.

```
Thread 1 -

1) if (list.size() < 5) {
2)   list.add(value1);
   }

Thread 2 -

a) if (list.size() < 5) {
b)   list.add(value2);
   }
```

Lets assume this execution sequence (1)(a)(2)(b) in this case both threads will see that list size is 4 and is less than 5. Hence both will add an element into the list, which makes the list size as 6 violating the condition. You can see the issue is not resolved even with both size() and add() being synchronized methods.

Solution -

You should apply thread synchronization at operation level with the help of synchronized block. i.e.

```
Thread 1 -

1) synhronized(list) {
2)   if (list.size() < 5) {
```

```
3)       list.add(value1);
     }
   }

Thread 2 -

a) synchronized(list) {
b)   if (list.size() < 5) {
c)      list.add(value2);
     }
   }
```

Now consider the execution sequence `(1)(a)(2)(3)(b)(c)`

You can see that list object is locked by Thread1 and hence even when the control switched to Thread2 it can not proceed as the lock is with Thread1. And Thread1 will add the element where as Thread2 will fail.

So when it comes to synchronizing operations synchronized blocks are always better choice over synchronized methods.

# Generics and Streams

This module covers the concepts of generics and streams in programming. Generics allow you to create classes, interfaces, and methods that operate on a parameterized type, enabling code reusability and type safety. Streams provide a way to process sequences of elements, such as collections, in a functional style.

## Generics

Generics enable you to define classes, interfaces, and methods with a placeholder for the type of data they operate on. This allows you to create more flexible and reusable code. For example, you can create a generic class `Box<T>` that can hold any type of object:

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

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

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

You can then create instances of `Box` for different types:

```
Box<Integer> intBox = new Box<>();
intBox.setItem(10);

Box<String> strBox = new Box<>();
strBox.setItem("Hello");
```

Generics also support bounded type parameters, allowing you to restrict the types that can be used. For example:

```
public <T extends Number> double add(T number1, T number2) {
    // add the the number
    return number1.doubleValue() + number2.doubleValue();
}
```

This method can only accept types that are subclasses of `Number`. For example, `Integer`, `Double`, etc.

```
add(10, 20); // valid
add(10.5, 20.5); // valid

add("Hello", "World"); // invalid
```

## Functional Style Programming (Functional Programming)

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It emphasizes the use of higher-order functions, first-class functions, and immutability.

- higher-order functions: Functions that can take other functions as arguments or return them as results.

- first-class functions: Functions that can be treated like any other variable, meaning they can be assigned to variables, passed as arguments, and returned from other functions.
- immutability: The concept of data that cannot be changed after it is created.

Example for functional style programming using Java:

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

List<Integer> squaredNumbers = numbers.stream()
    .map(n -> n * n)
    .collect(Collectors.toList());

System.out.println(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

In this example, we use the `map` function to apply a lambda expression that squares each number in the list, demonstrating the functional programming style.

**Why Functional Programming?**

Functional programming offers several advantages, including:

- Improved readability: Functional code is often more concise and easier to understand, as it focuses on what to do rather than how to do it.
- Easier maintenance: Functional code is often easier to maintain and modify, as functions are self-contained and do not rely on external state.
- Enhanced modularity: Functions can be reused and composed, promoting code reuse and modular design.
- Better support for parallelism: Functional programming can make it easier to write concurrent and parallel code, as functions do not have side effects and can be executed independently

**Drawbacks of Functional Programming**

Functional programming can sometimes lead to performance issues due to the creation of intermediate objects and the use of recursion instead of iteration. Additionally, it may not be suitable for all types of problems, especially those that require mutable state or side effects(means changes to data).

For example, in scenarios where performance is critical and low-level optimizations are necessary, functional programming may introduce overhead due to the creation of multiple intermediate data structures. Consider the following example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> doubledNumbers = numbers.stream()
    .map(n -> n * 2)
    .collect(Collectors.toList());
```

In this case, each operation in the stream may create intermediate lists, which can lead to increased memory usage and slower performance compared to imperative programming approaches that modify data in place.

**Functional Interfaces and Lambdas**

Functional interfaces are interfaces that contain a single abstract method. They are used to represent functions as first-class citizens in programming languages that support functional programming concepts. In Java, functional interfaces can be implemented using lambda expressions or method references.

Interface `Runnable` is a functional interface because it has a single abstract method `run()`, similarly `Callable`, `Comparator`, `Supplier`, `Consumer`, and `Function` are also functional interfaces.

Consider the following add example of a functional interface:

```
@FunctionalInterface
interface MyAdder {
    int add(int a, int b);
}
```

Now this can be implemented using a lambda expression:

```
MyAdder adder = (a, b) -> a + b;
int result = adder.add(5, 10);
System.out.println(result); // Output: 15
```

In the above example, we define a functional interface `MyAdder` with a single abstract method `add()`. We then create an instance of `MyAdder` using a lambda expression that implements the `add()` method. Lambda expressions provide a concise way to represent functions and can be used wherever a functional interface is expected.

## Streams

Streams provide a powerful way to process sequences of elements in a functional style. They support operations such as filtering, mapping, and reducing. The purpose of streams is to enable functional-style operations on collections of data.

- filtering: Selecting elements that meet certain criteria.
- mapping: Transforming elements from one form to another.
- reducing: Combining elements to produce a single result.

Streams are composed of three main components:

- source: The data source for the stream (e.g., a collection).
- intermediate operations: Operations that transform the stream (e.g., filter, map).
- terminal operations: Operations that produce a result or side effect (e.g., collect, forEach, reduce).

Streams can be created from various data sources, such as collections, arrays, or I/O channels.

For example, you can create a stream from a list and perform various operations:

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

You can also use streams to perform transformations on data. For example, you can convert a list of strings to uppercase:

```
List<String> upperCaseNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

```
System.out.println(upperCaseNames); // Output: [ALICE, BOB, CHARLIE, DAVID]
```

Streams also support reduction operations, such as finding the sum of a list of numbers:

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

int sum = numbers.stream()
    .reduce(0, Integer::sum);

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