Functional Programming in Java 8.

SOLID Principles

- The SOLID principles are a set of design principles in object-oriented programming that help developers to create robust, maintainable, and scalable software.
    1. **S**ingle Responsibility Principle.
    2. **O**pen/Closed Principle.
    3. **L**iskov Substitution Principle.
    4. **I**nterface Segregation Principle.
    5. **D**ependency Inversion Principle.
- **Reference**: SOLID Principles

Single Responsibility Principle (SRP)

- According to Single Responsibility Principle, "**A class should have only one reason to change, meaning that it should have only one job or responsibility**".
- Let's look at an example of a class that violates the Single Responsibility Principle:

```
public class Employee {
    private String name;
    private int id;
    private String department;

    public Employee(String name, int id, String department) {
        this.name = name;
        this.id = id;
        this.department = department;
    }

    public double calculateSalary() {
        double salary = 0;
        // logic for calculating salary
        return salary;
    }

    public void saveToDatabase() {
        // logic to save employee to the database
    }

    public void printEmployeeDetails() {
        System.out.println("Employee Name: " + this.name);
        System.out.println("Employee ID: " + this.id);
        System.out.println("Employee Department: " + this.department);
    }
}
```

- This class has multiple responsibilities:
    - managing employee data,
    - calculating salary,
    - saving data to the database, and

    - printing information.
- Each responsibility is related to a different area of the application, so the class has multiple reasons to change.
- To follow the **Single Responsibility Principle**, we can refactor the code by breaking down the Employee class into smaller, more focused classes, each handling a specific responsibility.

```
// Class that manages employee data
public class Employee {
    private String name;
    private int id;
    private String department;

    public Employee(String name, int id, String department) {
        this.name = name;
        this.id = id;
        this.department = department;
    }

    // Getters for employee attributes
    public String getName() { return name; }
    public int getId() { return id; }
    public String getDepartment() { return department; }
}
```

```
// Class responsible for salary calculation
public class SalaryCalculator {
    public double calculateSalary(Employee employee) {
        double salary = 0;
        // Logic to calculate salary
        return salary;
    }
}
```

```
// Class responsible for saving employee to database
public class EmployeeRepository {
    public void saveToDatabase(Employee employee) {
        // Logic to save employee to database
    }
}
```

```
// Class responsible for printing employee details (SRP: printing)
public class EmployeePrinter {
    public void printEmployeeDetails(Employee e) {
        System.out.println("Employee Name: " + e.getName());
        System.out.println("Employee ID: " + e.getId());
        System.out.println("Employee Department: " + e.getDepartment());
    }
}
```

- Now, if there is a change in the way salaries are calculated, we only need to modify the SalaryCalculator class. Similarly, if the method of printing or saving data changes, we only need to modify the respective classes.
- This design follows Single Responsibility Principle because:
    - Each class has a single responsibility.
    - Each class has only one reason to change.
    - Changes in one part of the system are less likely to affect other parts.
- **Conclusion**
    - By following SRP, our code becomes simpler and easier to manage.
    - SRP helps in keeping different tasks separate from each other. This makes it easier to change or add things in the future without disturbing the entire system.
    - When classes are small and focused on one specific task (following SRP), they become easier to test, maintain, and grow as our program expands.

Design Patterns

- A general reusable solution to a commonly occurring problem in software design is called as design pattern.
- Design Pattern Classification:
    1. Creational Design Patterns
    2. Structural Design Patterns
    3. Behavioral Design Patterns
- **Creational Design Patterns**
    - These are Object Creation Patterns.
    - Below design patterns are creational design patterns:
        - Singleton Pattern
        - Factory Method Pattern
        - Abstract Factory Pattern
        - Builder Pattern
        - Prototype Pattern
- **Structural Design Patterns**
    - These are Object Composition Patterns.
    - Below are the structural design patterns:
        - Adapter Pattern
        - Bridge Pattern
        - Composite Pattern
        - Decorator Pattern
        - Facade Pattern
        - Flyweight Pattern
        - Proxy Pattern
- **Behavioral Patterns**
    - These are Object Interaction Patterns.
    - Below are the Behavioral Patterns:
        - Chain of Responsibility Pattern
        - Command Pattern
        - Interpreter Pattern
        - Iterator Pattern
        - Mediator Pattern
        - Memento Pattern
        - Observer Pattern
        - State Pattern

    - Strategy Pattern
    - Template Method Pattern
    - Visitor Pattern
- Reference: Design Patterns

Builder Design Pattern

- Consider class Employee with attributes name, empid, department, designation and salary.
- **Approach 1**: Class implementation by overloading constructor
    - Define class Employee with appropriate fields

```
public class Employee {
    private String name;
    private int id;
    private String department;
    private String designation;
    private double salary;

    @Override
    public String toString(){
        return name+" "+id+" "+department+" "+designation+" "+salary;
    }
}
```

    - Instantiate class Employee

```
Employee emp1 = new Employee("Sandeep", 102, "IT", "Tech Lead", 55000);

Employee emp2 = new Employee("Chetan", 101, "IT", "Manager");

Employee emp3 = new Employee("Mahesh", 103, 40000);
```

    - To instantiate class, we must overload constructor inside class.

```
class Employee{
    public Employee(String name, int id, String department,
                            String designation, double salary){
        this.name = name;
        this.empid = empid;
        this.department = department;
        this.designation = designation;
        this.salary = salary;
    }
    public Employee(String name, int id, String department, String designation){
        this(name, id, department, designation, 0 );
    }
    public Employee(String name, int id, double salary){
        this(name, id, null, null, salary );
```

```
        }
    }
```

- If we add more attributes inside Employee class then number of constructor will also gets increased.
- Constructor is responsible for validating and setting values to the attributes. It is violating the Single Responsibility Principle (SRP) as the constructor is taking care of more than one job.
- During instantiation, we must follow the order of arguments. Also, if we want to make any field optional then we must specify default value. In other words, there is no flexibility in instantiation.

- **Approach 2**: Class implementation by using Builder pattern

```
public class Employee {
    private String name;
    private int id;
    private String department;
    private String designation;
    private double salary;

    // Private constructor to enforce Builder usage
    private Employee(EmployeeBuilder builder) {
        this.name = builder.name;
        this.id = builder.id;
        this.department = builder.department;
        this.designation = builder.designation;
        this.salary = builder.salary;
    }

    @Override
    public String toString() {
        return name+" "+id+" "+department+" "+designation+" "+salary;
    }

    // Builder Class
    public static class EmployeeBuilder {
        private String name;
        private int id;
        private String department;
        private String designation = "Not Provided";  // Default value
        private double salary = 0.0;  // Default value

        // Mandatory fields
        public EmployeeBuilder(String name, int id, String department) {
            this.name = name;
            this.id = id;
            this.department = department;
        }

        // Optional fields
        public EmployeeBuilder designation(String designation) {
            this.designation = designation;
            return this;
        }

        public EmployeeBuilder salary(double salary) {
```

```
            this.salary = salary;
            return this;
        }

        // Build the Employee object
        public Employee build() {
            return new Employee(this);
        }
    }
}
```

- How to use it?

```
public class Program {
    public static void main(String[] args) {
        // Using the Builder pattern to create employees
        Employee e1 = new Employee.EmployeeBuilder("Sandeep", 102, "IT")
            .salary(55000)
            .designation("Tech Lead")
            .build();

        Employee e2 = new Employee.EmployeeBuilder("Chetan", 101, "IT")
            .designation("Manager")
            .build();

        Employee e3 = new Employee.EmployeeBuilder("Mahesh", 103, "IT")
            .build();

        System.out.println(e1);
        System.out.println(e2);
        System.out.println(e3);
    }
}
```

- **Advantages of Using Builder Pattern**

1. Improved Readability:
    - We can easily see what parameters are being set with named methods like .designation() and .salary(). No need to worry about the order of parameters in a constructor.
2. Flexibility:
    - we can skip optional fields (e.g., salary, designation) when creating objects, while still providing default values.
3. No Constructor Overload:
    - Instead of managing multiple constructors with different combinations of parameters, we use a single builder to set the fields that we need, making the code cleaner.
4. No Defaults in Constructor:
    - Defaults like "Not Provided" for designation and 0.0 for salary are handled internally, so we don't have to deal with meaningless values when parameters are not provided.

Java Version History

Sr.No.	Version	Release Date	Support	Sr.No.	Version	Release Date	Support
1	JDK Beta	23 May 1995		16	Java SE 14	17 March 2020	
2	JDK 1.0	23 January 1996		17	Java SE 15	15 September 2020	
3	JDK 1.1	19 February 1997		18	Java SE 16	16 March 2021	
4	J2SE 1.2	08 December 1998		19	Java SE 17	14 September 2021	LTS
5	J2SE 1.3	08 May 2000		20	Java SE 18	22 March 2022	
6	J2SE 1.4	06 February 2002		21	Java SE 19	20 September 2022	
7	J2SE 5.0	30 September 2004		22	Java SE 20	21 March 2023	
8	Java SE 6	11 December 2006		23	Java SE 21	19 September 2023	LTS
9	Java SE 7	28 July 2011		24	Java SE 22	19 March 2024	
10	Java SE 8	18 March 2014	LTS	25	Java SE 23	17 September 2024	
11	Java SE 9	21 September 2017		26	Java SE 24		
12	Java SE 10	20 March 2018		LTS => Long Term Support			
13	Java SE 11	25 September 2018	LTS				
14	Java SE 12	19 March 2019					
15	Java SE 13	17 September 2019					

Long Term Support

- A stable release of Java is maintained for a longer period of time than the standard edition.
- Java SE 8, 11, 17 and 21 versions are LTS versions.
- Reference: Java end of life date

Java 8

- Java 8 was released in March 2014. **Java 8 changes enable us to write programs more easily, instead of writing verbose code**.
- Let us sort list of apples in inventory based on their weight without Java 8 features.
    - Let us define Apple class

```
class Apple {
    private int weight = 0;
    private String color = "";

    public Apple(int weight, String color){
        this.weight = weight;
        this.color = color;
    }
    public int getWeight() {
        return weight;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return color+"  "+weight;
    }
}
```

    - Lets create inventory of Apple.

```
    class Program{
        public static void main( String[] args ){
             List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                                   new Apple(155, "green"),
                                                   new Apple(120, "red"));
        }
    }
```

- Let us sort inventory using Collections.sort() method.

```
    class Program{
        public static void main( String[] args ){
             List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                                   new Apple(155, "green"),
                                                   new Apple(120, "red"));
            //Instance of Anonymous Inner class
            Comparator<Apple> comparator = new Comparator<Apple>( ){
                public int compare( Apple a1, Apple a2 ){
                    return a1.getWeight() - a2.getWeight();
                }
            };

            Collections.sort( inventory, comparator );
        }
    }
```

- Or we can rewrite as

```
class Program{
        public static void main( String[] args ){
             List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                                   new Apple(155, "green"),
                                                   new Apple(120, "red"));

            //Instance of Anonymous Inner class as a argument
            Collections.sort( inventory, new Comparator<Apple>( ){
                public int compare( Apple a1, Apple a2 ){
                    return a1.getWeight() - a2.getWeight();
                }
            } );
        }
    }
```

- Let us sort list of apples in inventory based on their weight with Java 8 features.
    - Let us use Collections.sort() method using method reference.

```
public class Program {
    public static void main(String[] args) {
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
```

```
                new Apple(155, "green"),
                new Apple(120, "red"));

        //Using Lambda Expression
        //Function<Apple, Integer> function = apple -> apple.getWeight();

        //Using Method reference
        Function<Apple, Integer> function = Apple::getWeight;

        Comparator<Apple> comparator = Comparator.comparing(function);

        Collections.sort( inventory, comparator );
    }
}
```

- or we can rewrite as

```
public class Program {
    public static void main(String[] args) {
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                new Apple(155, "green"),
                new Apple(120, "red"));

        Collections.sort( inventory, Comparator.comparing( Apple::getWeight) );
    }
}
```

- We can use static import to access comparing method

```
import static java.util.Comparator.comparing;
public class Program {
    public static void main(String[] args) {
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                new Apple(155, "green"),
                new Apple(120, "red"));

        Collections.sort( inventory, comparing( Apple::getWeight) );
        //How to read? --> Sort inventory by comparing apple weight
    }
}
```

- Now observe the difference

Prior Java 8	Using Java 8
`Collections.sort( inventory, new Comparator<Apple>( ){ public int compare( Apple a1, Apple a2 ){ return a1.getWeight() - a2.getWeight(); } } );`	`Collections.sort( inventory, comparing( Apple::getWeight) );`

- Why **Java Parallelism** is **Important** and How It Evolved
    - Modern computers have **multicore processors**, meaning our laptop or desktop has multiple cores (like 4 or more). However, most Java programs only use **one core**, leaving the other cores **idle**. This means the computer isn't fully utilizing its processing power.
    - In the past, if we wanted to take advantage of all the cores, **threads** were the solution. But working with threads is **complex** and **prone to mistakes**. Over the years, Java has improved how it handles **parallel processing** (using multiple cores) to make it easier and safer.
        - **Java 1.0**: Threads and locks were available, but they were **difficult** to use properly.
        - **Java 5**: Introduced tools like **thread pools** and **concurrent collections** to manage threads more easily.
        - **Java 7**: Introduced the **fork/join framework**, which made it easier to split tasks for parallel processing, but it was still a bit tricky to use.
        - **Java 8**: Java introduced a **simpler** and **more powerful** way to work with multiple cores, making parallelism more accessible and less error-prone.
    - Java 8 introduces a new feature called **Streams**, which makes it easier to process large amounts of data in parallel. It’s similar to how we might write **queries** in a database—by simply saying what we want, and letting the system figure out the best way to execute it.
    - With **Streams**, we don’t need to worry about writing complex code with **synchronized blocks**, which are difficult to manage and can cause problems in parallel processing. Synchronized code can also be slow, especially on **multicore processors**.
    - **Multicore processors** have separate **fast memory (caches)** for each core. When we use locks (like synchronized), it forces these separate caches to communicate with each other, which is slow and inefficient.

- Java 8's ability to pass code to methods brings us closer to **functional programming**, where we can treat code as functions.
    - We can think of the introduction of **Streams** in Java 8 as the reason why two other important features were added:
        - **Lambdas** and **method references**, which let us pass code to methods in a more concise way.
        - **Default methods** in interfaces, which allow interfaces to have methods with default implementations.
    - While **passing code to methods** might seem like it’s just a feature of **Streams**, it actually opens up a lot of new possibilities in Java 8. This feature gives us a **cleaner and shorter way** to customize behavior. For example, if we want to write two methods that are almost the same except for a few lines, we can now just **pass the different parts** of the code as arguments, instead of copying and pasting. This makes the code **clearer, shorter**, and less **error-prone**.
    - Before Java 8, we could do something similar using **anonymous classes**, but it would have been much more complicated and less clear.

- **Java 8 New Features**
    - Default Method
    - Static Interface Method
    - Functional Interface
    - Lambda Expression
    - Method Reference
    - Constructor Reference
    - Streams API
    - New Date/Time API

- Optional class
- CompletableFuture
- Type Annotations
- Nashorn JavaScript Engine

Abstract Method

- Array is a collection. Let's define interface Collection with methods:
  - void acceptRecord( );
  - void printRecord( );
  - int[] toArray( );

```
interface Collection{
    void acceptRecord();
    //public abstract void acceptRecord();

    void printRecord();
    //public abstract void printRecord();

    int[] toArray();
    //public abstract int[] toArray();
}
```

- Interface methods are by default considered as public and abstract.
- Note that abstract methods do not contain body. Implementation class must override method to provide body.
- If we do not redefine/override abstract method inside sub class then sub class will also considered as abstract.
- We can create reference but we can not create instance of abstract class. In other words, we can not instantiate abstract class.
- Let's define Collection implementation class Array to provide concrete implementation.

```
class Array implements Collection{
    private int[] arr;
    public Array( int length ){
        this.arr = new int[ length ];
    }
    @Override
    public void acceptRecord() {
        try(Scanner sc = new Scanner(System.in)){
            for( int index = 0; index < this.arr.length; ++ index ){
                System.out.print("Enter arr[ "+index+" ]    ::  ");
                this.arr[ index ] = sc.nextInt();
            }
        }
    }

    @Override
    public void printRecord() {
        for( int index = 0; index < this.arr.length; ++ index ){
            System.out.println("arr[ "+index+" ]    ::"+this.arr[ index ] );
        }
    }
```

```
    @Override
    public int[] toArray() {
        return this.arr;
    }
}
```

- Test the functionality in main method.

```
public class Program {
    public static void main(String[] args) {
        Collection c = new Array( 5 );
        c.acceptRecord();
        c.printRecord();
    }
}
```

- Prior Java 8, What if we want to modify(add/update) interface design? In this case we must make changes in all the implementation classes.
- Let's say, we want to include "void sort( )" method inside interface.

```
interface Collection{
    void acceptRecord();
    //public abstract void acceptRecord();

    void printRecord();
    //public abstract void printRecord();

    int[] toArray();
    //public abstract int[] toArray();

    void sort();
    //public abstract void sort();
}
```

- Now we must override "void sort()" method inside Array class. Otherwise Array class will be considered as Abstract.
- Consider scenario, Collection interface is having multiple implementation classes and some of them want to override and some of them dont want to override "void sort()" method then?
- To address this issue we can take help of abstract helper class. In this case we will define intermediate class between interface(I/F) and I/F implementation class. This class will override all the methods of I/F with empty body. It will not force developer to override all the methods of I/F. Now Developer has freedom to override few/all the methods of interface.
- Generally framework designers use this technique. For Example:
    - List<E> is an interface. AbstractList<E> is abstract class. ArrayList<E> is concrete class that extends AbstractList<E> class.
    - Set<E> is an interface. AbstractSet<E> is abstract class. HashSet<E> is concrete class that extends AbstractSet<E> class.
    - Queue<E> is an interface. AbstractQueue<E> is abstract class. PriorityQueue<E> concrete class that extends AbstractQueue<E> class.

- Map<E> is an interface. AbstractMap<E> is abstract class. HashMap<E> concrete class that extends AbstractMap<E> class.

- Consider below code:

```
abstract class AbstractCollection implements Collection{
    @Override
    public void acceptRecord() {
    }

    @Override
    public void printRecord() {
    }

    @Override
    public int[] toArray() {
        return null;
    }

    @Override
    public void sort() {
    }
}
```

- Without declaring method abstract, we can declare class abstract. Now implementation class should not implement interface rather it should extend abstract helper class. It will not force implementation class to override all the methods of class.
- Consider implementation class:

```
class Array extends AbstractCollection{
    private int[] arr;
    public Array( int length ){
        this.arr = new int[ length ];
    }
    @Override
    public void acceptRecord() {
        try(Scanner sc = new Scanner(System.in)){
            for( int index = 0; index < this.arr.length; ++ index ){
                System.out.print("Enter arr[ "+index+" ]    ::  ");
                this.arr[ index ] = sc.nextInt();
            }
        }
    }

    @Override
    public void printRecord() {
        for( int index = 0; index < this.arr.length; ++ index ){
            System.out.println("arr[ "+index+" ]    ::"+this.arr[ index ] );
        }
    }

    @Override
    public int[] toArray() {
        return this.arr;
```

```
    }

    @Override
    public void sort() {
        int[] arr = this.toArray();
        //Bubble Sort Logic
        for( int i = 0; i < arr.length - 1; ++ i ){
            for( int j = 0; j < arr.length - 1 - i; ++ j ){
                if( arr[ j ] > arr[ j + 1 ] ){
                    int temp = arr[ j ];
                    arr[ j ] = arr[ j + 1 ];
                    arr[ j + 1 ] = temp;
                }
            }
        }
    }
}
```

- Let's test the functionality in main method.

```
public class Program {
    public static void main(String[] args) {
        Collection c = new Array( 5 );
        c.acceptRecord();
        c.sort();
        c.printRecord();
    }
}
```

Default Method

- In Java, an interface is used to define a group of related methods, which act as a contract that any class implementing the interface must follow. This means that the class must provide an implementation for all the methods declared in the interface.
- However, there is a challenge when the library designers need to update the interface by adding new methods. In such cases, any existing class that implements the interface will have to be updated to provide implementations for the new methods. This can be a problem because many existing classes (such as those from third-party libraries like Guava or Apache Commons) may not be under the control of the library designers, and they would have to modify their code to match the new interface.
- As we saw in the previous section ("Abstract Method"), before Java 8, the problem was often solved by defining an intermediate abstract helper class.
- To address this issue, Java 8 introduces two key features:
    - Default Methods
    - Static Interface Methods
- Java 8 allows interfaces to define methods with actual implementations. These are called default methods. When a class implements an interface, it can either provide its own implementation of a default method or use the default implementation provided by the interface. This feature allows the interface to evolve without breaking existing classes that implement it.
- Let us Consider Collection interface once again.

```
interface Collection{
    void acceptRecord();
    void printRecord();
    int[] toArray();
}
```

- Let us add "void sort()" method inside interface, without forcing implementation classes to override it.

```
interface Collection{
    void acceptRecord();
    void printRecord();
    int[] toArray();
    //default void sort();    //Not OK: Must provide body
    default void sort( ){   //OK
        int[] arr = this.toArray();
        java.util.Arrays.sort( arr );
    }
}
```

- Providing body to the default method is mandatory. Now it is optional for sub class to override it.
- Let us use Bubble sort algorithm in default method:

```
interface Collection {
    void acceptRecord();
    void printRecord();
    int[] toArray();

    default void sort() {
        int[] arr = this.toArray();
        boolean swapped;
        for (int i = 0; i < arr.length - 1; i++) {
            swapped = false;
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) { // Array is already sorted
                break;
            }
        }
    }
}
```

- Let's define Collection implementation class Array to provide concrete implementation.

```
class Array implements Collection{
    private int[] arr;
    public Array( int length ){
        this.arr = new int[ length ];
    }
    @Override
    public void acceptRecord() {
        try(Scanner sc = new Scanner(System.in)){
            for( int index = 0; index < this.arr.length; ++ index ){
                System.out.print("Enter arr[ "+index+" ]    ::  ");
                this.arr[ index ] = sc.nextInt();
            }
        }
    }

    @Override
    public void printRecord() {
        for( int index = 0; index < this.arr.length; ++ index ){
            System.out.println("arr[ "+index+" ]    ::"+this.arr[ index ] );
        }
    }

    @Override
    public int[] toArray() {
        return this.arr;
    }
}
```

- Test the functionality in main method.

```
public class Program {
    public static void main(String[] args) {
        Collection c = new Array( 5 );
        c.acceptRecord();
        c.sort( );  //calling default method    //OK
        c.printRecord();
    }
}
```

- If required, we can override default method inside implementation class. Let us implement selection sort inside overriden method.

```
class Array implements Collection{
    private int[] arr;
    public Array( int length ){
        this.arr = new int[ length ];
    }
    @Override
    public void acceptRecord() {
        //TODO
    }
```

```
    @Override
    public void printRecord() {
        //TODO
    }

    @Override
    public int[] toArray() {
        //TODO
    }

    @Override
    public void sort( ){
        for( int i = 0; i < this.arr.length - 1; ++ i ){
            for( int j = i + 1; j < this.arr.length; ++ j ){
                if( arr[ i ] > arr[ j ] ){
                    int temp = arr[ i ];
                    arr[ i ] = arr[ j ];
                    arr[ j ] = temp;
                }
            }
        }
    }
}
```

- Test the functionality in main method.

```
public class Program {
    public static void main(String[] args) {
        Collection c = new Array( 5 );
        c.acceptRecord();
        c.sort( );  //OK
        c.printRecord();
    }
}
```

- We can call default method of interface from overriden method of sub class. consider below code

```
interface Collection{
    default void sort( ){   //OK
        int[] arr = this.toArray();
        java.util.Arrays.sort( arr );
    }
}
```

```
class Array implements Collection{
    @Override
    public void sort(){
        Collection.super.sort();    //Calling default method of interface
    }
}
```

- Overriding default method is optional. But in case of multiple inheritance, if name of default methods are same then we must override it. Consider below code:

```
interface A{
    default void sayHello(){
        System.out.printl("Hello from A");
    }
}
```

```
interface B{
    default void sayHello(){
        System.out.printl("Hello from A");
    }
}
```

```
class C implements A, B{    //Compiler Error
}
```

- To fix the issue, we must override sayHello() method inside sub class. Consider modified code:

```
class C implements A, B{
    @Override
    public void sayHello(){
        //Only for the demo purpose
        A.super.sayHello();
        B.super.sayHello();
    }
}
```

- **Three resolution rules to know**
    - There are three rules to follow when a class inherits a method with the same signature from multiple places (such as another class or interface):
        - Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.

```
interface A {
    default void show() {
        System.out.println("Interface A");
    }
}

class B {
    public void show() {
        System.out.println("Class B");
    }
```

```
}

class C extends B implements A {
    // No need to override show() because Class B has already defined it
}

public class Program {
    public static void main(String[] args) {
        C obj = new C();
        obj.show();  // Calls show() from Class B, not from Interface A
    }
}
```

- Otherwise, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected. (If B extends A, B is more specific than A).

```
interface A {
    default void show() {
        System.out.println("Interface A");
    }
}

interface B extends A {
    default void show() {
        System.out.println("Interface B");
    }
}

class C implements B {
    // No need to override; B's show() is automatically chosen because B is
more specific than A
}

public class Program {
    public static void main(String[] args) {
        C obj = new C();
        obj.show();  // Calls B's show(), as it's more specific
    }
}
```

- Finally, If a class implements multiple interfaces that have the same default method, the class must choose which one to use. It does this by overriding the default method and explicitly calling the desired version.

```
interface A {
    default void show() {
        System.out.println("Interface A");
    }
}

interface B {
    default void show() {
```

```
            System.out.println("Interface B");
        }
    }

    class C implements A, B {
        @Override
        public void show() {
            // Explicitly choosing which default method to use
            A.super.show(); // Calls the default method from A
        }
    }

    public class Program {
        public static void main(String[] args) {
            C c = new C();
            c.show();  // Calls the chosen method
        }
    }
```

- Let us consider scenario in case of **diamond inheritance**:
    - Example 1

```
interface A{
    default  void sayHello(){
        System.out.println("Hello from A");
    }
}
interface B extends   A{    }
interface C extends   A{    }
class D implements  B, C{   }   //OK
public class Program {
    public static void main(String[] args) {
        D d = new D();
        d.sayHello();   //Hello from A
    }
}
```

    - Example 2

```
interface A{
    default  void sayHello(){
        System.out.println("Hello from A");
    }
}
interface B extends   A{
    default  void sayHello(){
        System.out.println("Hello from B");
    }
}
interface C extends   A{    }
class D implements  B, C{   }   //OK
public class Program {
```

```
    public static void main(String[] args) {
        D d = new D();
        d.sayHello();   //Hello from B
    }
}
```

- B is more specific than A. Hence B's default method will call.

- Example 3

```
interface A{
    default  void sayHello(){
        System.out.println("Hello from A");
    }
}
interface B extends   A{
    default  void sayHello(){
        System.out.println("Hello from B");
    }
}
interface C extends   A{
    default  void sayHello(){
        System.out.println("Hello from C");
    }
}
class D implements  B, C{   //Not OK
    //Error: D inherits unrelated defaults for sayHello() from types B and C
}
public class Program {
    public static void main(String[] args) {
        D d = new D();
        d.sayHello();
    }
}
```

- **Remember:** Java 8 default method helps developer to evolve interface as well as it reduces the need of abstract helper classes.

Static Interface Method

- If we want to define helper/utility methods that are related to interface then we should define static method inside interface.
- Consider below example:

```
interface Collection {
    void acceptRecord();
    void printRecord();
    int[] toArray();

    // Static method to swap two elements in an array
    static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
```

```
            arr[j] = temp;
        }

        // Default sort method using the static swap method
        default void sort() {
            int[] arr = this.toArray();
            boolean swapped;
            for (int i = 0; i < arr.length - 1; i++) {
                swapped = false;
                for (int j = 0; j < arr.length - 1 - i; j++) {
                    if (arr[j] > arr[j + 1]) {
                        // Using the static swap method to swap elements
                        Collection.swap(arr, j, j + 1);
                        swapped = true;
                    }
                }
                if (!swapped) { // Array is already sorted
                    break;
                }
            }
        }
    }
```

- Static interface method provide a way to define factory methods that can be used to create instances of classes that implement the interface.
- Consider below example:

```
interface Collection{
    void acceptRecord();
    void printRecord();
    //Factory Method
    static Collection createCollection( int length ){
        return new Array( length );
    }
}
```

```
class Array implements  Collection{
    private int[] arr;
    public Array(int length) {
        this.arr = new int[ length ];
    }
    public void acceptRecord(){}
    public void printRecord(){}
}
```

```
public class Program {
    public static void main(String[] args) {
        Collection c = Collection.createCollection(5);
        c.acceptRecord();
        c.printRecord();
```

```
    }
}
```

- **Points to Remember:**
  - It is mandatory to provide body to the static interface method.
  - We can not override static interface method inside sub class but we can use it in sub class.
  - We can use it to define utility/helper method and factory methods inside interface.

Functional Interface

- If we define interface with **S**ingle **A**bstract **M**ethod then interface is called as SAM interface or functional interface.
- Consider below example:

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

  - Here Printable is functional interface and "void print( )" is method descriptor / functional method.
- Using @FunctionalInterface annotation, we can ensure weather interface is functional or not. But it is not mandatory to use @FunctionalInterface on interface.
- Consider below example:

```
@FunctionalInterface
interface Printable{
    void print( );
}
```

- Functional interface can contain:
  - Single abstract method
  - Default method(s)
  - Static interface method(s)
  - Final field(s)
  - Nested type(s)

```
@FunctionalInterface
interface Test{
    interface Printable{
        void show();
        void print();
    }

    int VALUE = 10;

    void f1();

    default  void f2(){}
    default  void f3(){}
```

```
    static  void f4(){}
    static  void f5(){}
}
```

- Consider below code to check whether interface is functional or not?
  - Example 1

```
@FunctionalInterface
interface Printable{
}
```

    - Printable is empty interface. Hence it will be compiler error
  - Example 2

```
@FunctionalInterface
interface Printable{
    void show( );
    void display( );
}
```

    - Printable contains two abstract method. Hence it will be compiler error
  - Example 3

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

    - Printable contains single abstract method. Hence it is functional interface. @FunctionalInterface is optional to use.
  - Example 4

```
@FunctionalInterface
interface Printable{
    void print();
}
@FunctionalInterface
interface  Writable extends  Printable{
}
```

    - Printable and Writable are functional interfaces.
- Lambda expression and method reference is used to implement functional interface.
- Functional interfaces declared in java.lang and java.util package
  - java.lang.Comparable
  - java.lang.Runnable
  - java.lang.AutoCloseable
  - java.lang.Iterable

    - java.lang.Readable
    - java.lang.Thread.UncaughtExceptionHandler
    - java.util.Comparator
- The java.util.function package in Java 8 provides a set of functional interfaces that are used extensively in functional programming and the Streams API.
- Reference: Functional interfaces
    - Predicate
        - boolean test(T t);
    - Function<T, R>
        - R apply(T t);
    - Consumer
        - void accept(T t);
    - Supplier
        - T get();
    - UnaryOperator
        - T apply(T t);
    - BinaryOperator
        - T apply(T t1, T t2);
    - BiFunction<T, U, R>
        - R apply(T t, U u);
    - BiConsumer<T, U>
        - void accept(T t, U u);
    - ToIntFunction
        - int applyAsInt(T t);
    - ToDoubleFunction
        - double applyAsDouble(T t);
    - ToLongFunction
        - long applyAsLong(T t);
    - IntPredicate
        - boolean test(int value);
    - IntFunction
        - R apply(int value);
    - DoubleFunction
        - R apply(double value);
    - LongFunction
        - R apply(long value);
    - IntUnaryOperator
        - int applyAsInt(int operand);
    - IntBinaryOperator
        - int applyAsInt(int left, int right);
    - DoubleUnaryOperator
        - double applyAsDouble(double operand);
    - DoubleBinaryOperator
        - double applyAsDouble(double left, double right);
    - LongUnaryOperator
        - long applyAsLong(long operand);
    - LongBinaryOperator
        - long applyAsLong(long left, long right);
    - IntSupplier

        - int getAsInt();
    - DoubleSupplier
        - double getAsDouble();
    - LongSupplier
        - long getAsLong();
    - ObjIntConsumer
        - void accept(T t, int value);
    - ObjDoubleConsumer
        - void accept(T t, double value);
    - ObjLongConsumer
        - void accept(T t, long value);
    - ToIntBiFunction<T, U>
        - int applyAsInt(T t, U u);
    - ToDoubleBiFunction<T, U>
        - double applyAsDouble(T t, U u);
    - ToLongBiFunction<T, U>
        - long applyAsLong(T t, U u);

Lambda Expression

- A well-known problem in software engineering is that no matter what we do, user requirements will change.
- Let us consider Apple example once again.

```
class Apple {
    private int weight = 0;
    private String color = "";

    public Apple(int weight, String color){
        this.weight = weight;
        this.color = color;
    }
    public int getWeight() {
        return weight;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return color+"  "+weight;
    }
}
```

- **Coping with changing requirements**
    - Filtering green, red and yellow apples

```
class Program{
    public static List<Apple> filterGreenApples(List<Apple> inventory){
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
```

```
            if(apple.getColor().equals("green")){
                result.add(apple);
            }
        }
        return result;
    }
    public static List<Apple> filterRedApples(List<Apple> inventory){
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
            if(apple.getColor().equals("red")){
                result.add(apple);
            }
        }
        return result;
    }
    public static List<Apple> filterYellowApples(List<Apple> inventory){
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
            if(apple.getColor().equals("yellow")){
                result.add(apple);
            }
        }
        return result;
    }
}
```

```
class Program{
    public static void main(String[] args){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));

        List<Apple> greenApples = Program.filterGreenApples(inventory);
        System.out.println(greenApples);

        List<Apple> redApples = Program.filterRedApples(inventory);
        System.out.println(redApples);

        List<Apple> yellowApples = Program.filterYellowApples(inventory);
        System.out.println(yellowApples);
    }
}
```

- Filtering apples by parameterizing the color.

```
class Program{
    public static List<Apple> filterApplesByColor(List<Apple> inventory, String
color){
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
            if(apple.getColor().equals(color)){
                result.add(apple);
```

```
            }
        }
        return result;
    }
}
```

```
class Program{
    public static void main(String[] args){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));

        List<Apple> greenApples = Program.filterApplesByColor(inventory,"green");
        System.out.println(greenApples);

        List<Apple> redApples = Program.filterApplesByColor(inventory,"red");
        System.out.println(redApples);

        List<Apple> yellowApples;
        yellowApples = Program.filterApplesByColor(inventory,"yellow");
        System.out.println(yellowApples);
    }
}
```

- Filtering apples by weight.

```
class Program{
    public static List<Apple> filterApplesByColor(List<Apple> inventory,
                                                  String color){
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
            if(apple.getColor().equals(color)){
                result.add(apple);
            }
        }
        return result;
    }
    public static List<Apple> filterApplesByWeight(List<Apple> inventory,
                                                   int weight){
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
            if(apple.getWeight() > weight){
                result.add(apple);
            }
        }
        return result;
    }
}
```

```
```

```
class Program{
    public static void main(String[] args){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));

        List<Apple> greenApples;
        greenApples = Program.filterApplesByColor(inventory, "green");
        System.out.println(greenApples);

        List<Apple> heavyApples;
        heavyApples = Program.filterApplesByWeight(inventory, 150);
        System.out.println(heavyApples);
    }
}
```

- Filtering apples by color and weight

```
class Program{
    public static List<Apple> filterApplesByColorAndWeight(List<Apple> inventory,
                                                           String color,
                                                           int weight){
        List<Apple> result = new ArrayList<>();
        for(Apple apple: inventory){
            if(apple.getColor().equals(color) && apple.getWeight() > weight ){
                result.add(apple);
            }
        }
        return result;
    }
}
```

```
class Program{
    public static void main(String[] args){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));

        List<Apple> heavyGreenApples;
        heavyGreenApples = Program.filterApplesByColorAndWeight(inventory,
"green", 150 );
        System.out.println(heavyGreenApples);

        List<Apple> heavyRedApples;
        heavyRedApples = Program.filterApplesByColorAndWeight(inventory, "red",
100);
        System.out.println(heavyRedApples);
    }
}
```

- What if we ask to filter with different attributes of an apple, for example, its size, its shape, its origin, and so on?
- Behavior parameterization is a software development pattern that lets us handle frequent requirement changes.
- A function, which represents condition is called as Predicate. Let us use predicate to fulfill above requirement.

```
interface ApplePredicate{
    boolean test(Apple a);
}
```

- We can now declare multiple implementations of ApplePredicate to represent different selection criteria.

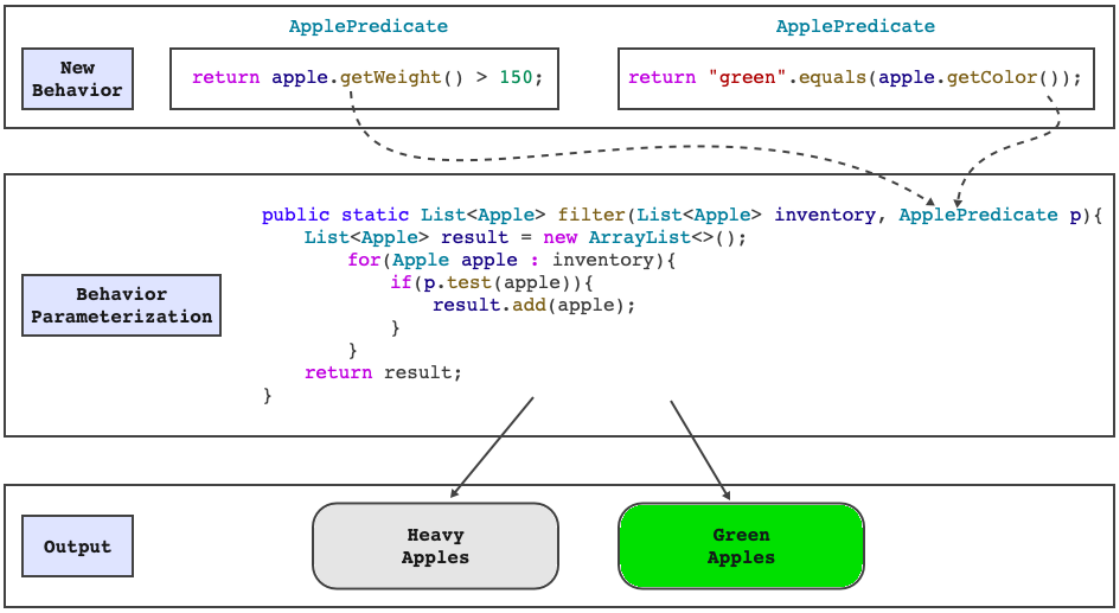
```
class AppleWeightPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
class AppleColorPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return "green".equals(apple.getColor());
    }
}

class AppleRedAndHeavyPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return "red".equals(apple.getColor()) && apple.getWeight() > 150;
    }
}
```

- Filtering Apples by abstract criteria

```
class Program{
    public static List<Apple> filter(List<Apple> inventory, ApplePredicate p){
        List<Apple> result = new ArrayList<>();
        for(Apple apple : inventory){
            if(p.test(apple)){
                result.add(apple);
            }
        }
        return result;
    }
}
```

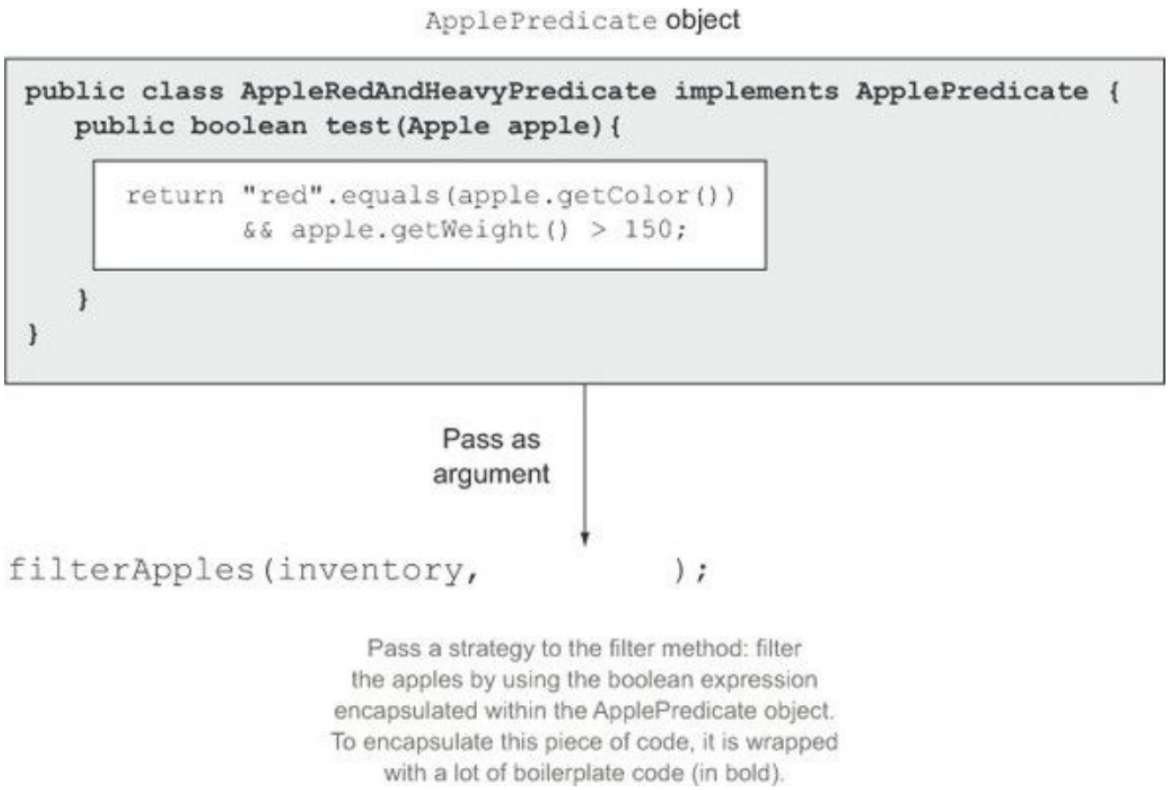
```
class Program{
    public static void main( String[] args ){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));
```

```
        //ApplePredicate p = new AppleColorPredicate();   //Upcasting
        //List<Apple> greenApples = filter(inventory, p);   //or

        List<Apple> greenApples = filter(inventory, new AppleColorPredicate());
        System.out.println(greenApples);

        List<Apple> heavyApples = filter(inventory, new AppleWeightPredicate());
        System.out.println(heavyApples);

        List<Apple> redAndHeavyApples;
        redAndHeavyApples = filter(inventory, new AppleRedAndHeavyPredicate());
        System.out.println(redAndHeavyApples);
    }
}
```



- In the context of filterXXX() method, the only code that really matters is the implementation of the test method. Unfortunately, because the filterXXX() method can only take objects, we have to wrap that code inside an ApplePredicate object. We can not pass method as a argument to another method.

- Consider below image:



- What we are doing is similar to “passing code” inline, because we are passing a boolean expression through an object that implements the test method.
- At the moment, when we want to pass new behavior to our filterXXX() method, we are forced to declare several classes that implement the ApplePredicate interface and then instantiate several ApplePredicate objects. There is a lot of verbosity involved and it is a time-consuming process!
- This is unnecessary overhead; can we do better?
- Java has a mechanism called anonymous classes, which let us declare and instantiate a class at the same time. They enable us to improve our code one step further by making it a little more concise.
- Let us filter red apples using an anonymous class.

```
class Program{
    public static void main( String[] args ){
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));

        List<Apple> redApples = filter(inventory, new ApplePredicate(){
            public boolean test( Apple apple){
                return "red".equals(apple.getColor());
            }
        });

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

- But anonymous classes are still not good enough.
  - First, they tend to be very bulky because they take a lot of space.
  - Second, many programmers find them confusing to use.

- Even though anonymous classes somewhat tackle the verbosity associated with declaring multiple concrete classes for an interface, they are still unsatisfactory.
- Behavior parameterization pattern makes our code more adaptive to requirement changes.
- In Java 8, If we want to pass code more concise way then we should use Lambda Expression.
- At the moment, the filterXXX() method works only for Apple. Using generics, we can make it generic.
    - Example 1:

```
interface ApplePredicate{
    boolean test(Apple a);
}

class Program{
    public static List<Apple> filter(List<Apple> inventory,
                                      ApplePredicate p){
        List<Apple> result = new ArrayList<>();
        for(Apple apple : inventory){
            if(p.test(apple)){
                result.add(apple);
            }
        }
        return result;
    }
}
```

    - Example 2:

```
interface Predicate<T>{
    boolean test(T a);
}

class Program{
    //Generic method: Observe <T> before return type
    public static <T> List<T> filter(List<T> inventory, Predicate<T> p){
        List<T> result = new ArrayList<>();
        for(T element : inventory){
            if(p.test(element)){
                result.add(element);
            }
        }
        return result;
    }
}
```

- A **lambda expression** can be understood as a concise representation of an anonymous function that can be passed around:
    - **Concise**: It is shorter and cleaner than using an anonymous class.
    - **Anonymous**: It doesn't have a name, unlike regular methods.
    - **Function**: It works like a method and can have parameters, a body, a return type, and may throw exceptions.
    - **Passed around**: You can pass a lambda as an argument or store it in a variable.

- In general, a statement which contains and operands + operator is called a expression.
    - Plus(+) is an arithmetic operator
        - Hence c = a + b; is arithmetic expression
    - && and || are logical operator
        - Hence boolean isLeapYear = (year % 4 == 0) && (year % 100 != 0 || year % 400 == 0); is logical expression
- In Java 8, arrow operator(**->**) is called as lambda operator. Its meaning is "goes to".
- If an expression contains lambda operator(->) then it is called as lambda expression. Consider below example:

```
(Apple a1, Apple a2) -> a1.getColor().compareTo(a2.getColor());
```

    - **(Apple a1, Apple a2)** : are called as input parameters or lambda parameters
    - **->** : is Arrow operator. Also called as lambda operator
    - **a1.getColor().compareTo(a2.getColor());** : is lambda body.
- How to read it?
    - Input parameter's goes to lambda body.
- We have already dicussed, If an interface contains single abstract method then it is called as SAM / functional interface and abstract method is called as function descriptor/functional method.
- To implement functional method, we can use lambda expression / method reference. But remember, overriden functional method is anonymous. Hence lambda expression is also called as anonymous method. we will discuss its working later.
- Syntax:
    - FunctionalInterfaceName f = Lambda Expression //OK
    - FunctionalInterfaceName f = (I/P params) -> Lambda Body; //OK
    - FunctionalInterfaceName f = (I/P params) -> { Lambda Body; }; //OK
- Let us consider ApplePredicate interface once again

```
@FunctionalInterface
interface ApplePredicate{
    boolean test(Apple a);
}
```

- Implementation Style 1:

```
class ApplePredicateImpl implements ApplePredicate{
    public boolean test(Apple a){
        return "red".equals(a.getColor());
    }

    public static void main( String[] args ){
        ApplePredicate predicate = new ApplePredicateImpl();
```

```
        //TODO
    }
}
```

- Implementation Style 2:

```
class Program{
    public static void main( String[] args ){
        //Using anonymous inner class
        ApplePredicate predicate = new ApplePredicate(){
            public boolean test(Apple a){
                return "red".equals(a.getColor());
            }
        };
        //TODO
    }
}
```

- **Note:** to create lambda expression, always check return type in parameters of abstract method(also called as functional method) of functional interface.

- Implementation Style 3:

```
class Program{
    public static void main( String[] args ){
        //Using Lambda Expression

        //Syntax
        //FunctionalInterface f = (Input Parameters) -> Lambda Body

        //public boolean test(Apple a) : is a functional method.
        // Apple a should be imput parameter
        // Expression should return boolean value
        ApplePredicate predicate1 = (Apple apple) -> {
            boolean result = "red".equals(apple.getColor());
            return result;
        };

        //Lambda body may contain one or statements.
        ApplePredicate predicate2 = (Apple apple) -> {
            return "red".equals(apple.getColor());
        };

        //If lambda body contains single statement
        //then { } and return is not required.
        ApplePredicate predicate3;
        predicate3 = (Apple apple) -> "red".equals(apple.getColor());

        //We can use different parameter name
        ApplePredicate predicate4 = (Apple a) -> "red".equals(a.getColor());
    }
}
```

- In the context of lambda expressions, type inference allows the Java compiler to determine the types of the parameters and the return type based on the functional interface.
    - Type inference example in Lambda Expression

```
class Program{
    public static void main(String[] args){
        ApplePredicate predicate1 = (Apple a) -> "red".equals(a.getColor());

        //Type inference: the parameter types and return type are inferred
        ApplePredicate predicate2 = (a) -> "red".equals(a.getColor());
        ApplePredicate predicate3 = (apple) -> "red".equals(apple.getColor());
    }
}
```

- In the lambda expression, if input parameter list is empty then use of parentheses is must.
    - ( ) -> System.out.println("Hello World"); //OK
- In the lambda expression, if input parameter list contains more than one parameter then use of parentheses is must.
    - (Apple a1, Apple a2) -> a1.getColor().compareTo(a2.getColor); //OK
    - Apple a1, Apple a2 -> a1.getColor().compareTo(a2.getColor); //Not OK
- If the lambda expression has exactly one parameter then we can omit the parentheses around the parameter.
    - (Apple apple) -> "red".equals(a.getColor()); //OK
    - (apple) -> "red".equals(a.getColor()); //OK
    - apple -> "red".equals(a.getColor()); //OK

```
class Program{
    public static void main(String[] args){
        ApplePredicate predicate = apple -> "red".equals(apple.getColor());  //OK
    }
}
```

- **Valid lambda expressions in Java 8**:
    - ( ) -> { }
        - Valid
        - This lambda has no parameters and returns void. It is similar to a method with an empty body: public void print() { }.
    - ( ) -> "Welcome!"
        - Valid
        - This lambda has no parameters and returns a String
    - ( ) -> { return "Welcome!"; }
        - Valid
        - This lambda has no parameters and returns a String (using an explicit return statement).
        - **Note**: to use return keyword, { } are must.

    - (Integer i) -> return "Count" + i;
        - Invalid
        - To make this lambda valid, curly braces are required as follows: (Integer i) -> {return "Count" + i;}.
    - (String s) -> { "Welcome!"; }
        - Invalid
        - To make this lambda valid:
            - Either we can remove the curly braces and semicolon as follows: (String s) -> "Welcome".
            - Or we can use an explicit return statement as follows: (String s) -> {return "Welcome";}.

- Below are the functional interfaces declared in java.util.function package:

    - java.util.Predicate

        - boolean test(T t) : Functional Method
        - Lambda expression should accept single parameter and return boolean value.

```
//Check the declaration of boolean test(T t) method. Here T means Apple
Predicate<Apple> predicate = (Apple apple) -> apple.getWeight() > 150;  //or
Predicate<Apple> predicate = Apple apple -> apple.getWeight() > 150;   //or
Predicate<Apple> predicate = apple -> apple.getWeight() > 150;

Apple redApple = new Apple(200, "Red");
Apple greenApple = new Apple(120, "Green");

System.out.println("Is redApple Heavy? "+ predicate.test( redApple ) );
System.out.println("Is greenApple Heavy? "+ predicate.test( greenApple ) );
```

    - java.util.function.BiPredicate<T,U>

        - boolean test(T t, U u) : Functional Method
        - Lambda expression should accept two parameter and return boolean value.
        - Check if apple's weight is greater than 150 and color is "red"

```
BiPredicate<Apple,Integer> predicate;
predicate = ( apple, weight ) -> apple.getWeight() > weight &&
                                 apple.getColor().equals("red");

Apple apple1 = new Apple(200, "Red");
Apple apple2 = new Apple(120, "Green");

System.out.println("Is apple1 Heavy? "+ predicate.test( apple1, 150 ) );
System.out.println("Is apple2 Heavy? "+ predicate.test( apple2, 150 ) );
```

    - java.util.Consumer

        - void accept(T t) : Functional Method
        - Lambda expression should accept single parameter and must not return any value.

```
//Check the declaration of void accept(T t) method. Here T means Apple
Consumer<Apple> consumer = apple -> System.out.println( apple );
```

```
Apple redApple = new Apple(200, "Red");
Apple greenApple = new Apple(120, "Green");

consumer.acccept( redApple );
consumer.acccept( greenApple );
```

- java.util.function.BiConsumer<T,U>
  - void accept(T t, U u) : Functional Method
  - Lambda expression should accept two parameters and must not return any value.

```
BiConsumer<Apple,message> consumer = (apple,message) ->
System.out.println(message +": "+ apple );

Apple redApple = new Apple(200, "Red");
Apple greenApple = new Apple(120, "Green");

consumer.acccept( redApple , "Red Apple");
consumer.acccept( greenApple, "Green Apple" );
```

- java.util.Supplier
  - T get() : Functional Method
  - Lambda expression can not accept parameter but it must return a value.

```
//Check the declaration of T get() method. Here T means Apple
Supplier<Apple> supplier = ( ) -> { return new Apple(175, "Yellow"); }; //or
Supplier<Apple> supplier = ( ) -> new Apple(175, "Yellow");

Apple apple = supplier.get( );
System.out.println( apple );
```

- java.util.Function<T,R>
  - R apply(T t) : Functional Method
  - Lambda expression must accept single parameter and must return a value.

```
//Check the declaration of R apply(T t) method. Here T means Apple and R means
String
Function<Apple,String> function = apple -> { return apple.getColor(); };    //or
Function<Apple,String> function = apple -> apple.getColor();

Apple greenApple = new Apple(120, "Green");
String color = function.apply( greenApple );
System.out.println("The color of apple is: " + color);  // Green
```

- java.util.function.UnaryOperator
  - It is sub interace of java.util.function.Function<T,R> interface.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    T apply(T t);
}
```

- It represents an operation on a single operand that produces a result of the same type as its operand.

```
UnaryOperator<Apple> increaseWeight = apple -> {
    apple.setWeight(apple.getWeight() + 20);
    return apple;
};

Apple apple = new Apple(150, "Green");

System.out.println("Before::"+ apple) ;
apple = increaseWeight.apply( apple );
System.out.println("After::"+ apple) ;
```

```
UnaryOperator<String> operator = str -> str.toUpperCase();

String message = "Hello";
message = operator.apply( message );
System.out.println(message);
```

- java.util.function.BiFunction<T,U,R>
  - R apply(T t, U u) : Functional Method
  - Lambda expression must accept two parameters and must return a value.

```
BiFunction<Apple,String,String> function = (apple,message) -> message+"
"+apple.getColor();

Apple greenApple = new Apple(120, "Green");
String str = function.apply( greenApple,  "The color of apple is: ");
System.out.println(str);
```

- java.util.function.BinaryOperator
  - It is sub interface of BiFunction interface.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
    T apply(T t1, T t2);
}
```

    - It represents an operation on two operands of the same type that produces a result of the same type.

```
BinaryOperator<Integer> binOperator = ( num1, num2 ) -> num1 + num2;
Integer result = binOperator.apply( 10, 20 );
System.out.println("Result  :   "+result);
```

- Boxing is the process of converting value of primitive type into its corresponding Wrapper class or String.

```
int number = 10;
Integer i1 = new Integer(number);  //Boxing
Integer i2 = Integer.valueOf(number);  //Boxing
String str1 = Integer.toString(number);  //Boxing
String str2 = String.valueOf(number)    //Boxing
```

- If boxing is done implicity then it is called as AutoBoxing.

```
int number = 123;
Integer i = number; //AutoBoxing
//Integer i = Integer.valueOf(number);  //Boxing
```

- Unboxing is the process of converting value of non primitive type(Wrapper class or String) into primitive type.

```
Integer i = new Integer(10);
int number = i.intValue();  //Unboxing

String str = "123";
int value = Integer.parseInt( str ); //Unboxing
```

- If unboxing is done implicity then it is called as AutoUnboxing.

```
Integer i = new Integer(10);
int number = i; //AutoUnboxing
```

- Since there is CPU and Memory overhead involved in Boxing/AutoBoxing and Unboxing/AutoUnboxing, we should avoid use of it in high-performance scenarios.
- Predicate is functional interface but IntPredicate is it's Primitive specialization.
    - Predicate requires boxing/unboxing.
    - IntPredicate works without boxing/unboxing.

```
package com.example;
import java.util.function.Predicate;
public class Program {
```

```
    public static final int SIZE = 10000000;  // Number of iterations

    public static void main(String[] args) {
        Predicate<Integer> isEven = num -> num % 2 == 0;

        long startTime = System.nanoTime();
        for (int i = 0; i < SIZE; i++) {
            isEven.test(i);  // Boxing happens here
        }
        long endTime = System.nanoTime();

        long durationPredicate = endTime - startTime;
        System.out.println("Predicate execution time: " + durationPredicate + " ns");
    }
}
//Predicate execution time: 18957723 ns
```

```
package com.example;
import java.util.function.IntPredicate;

public class Program {
    public static final int SIZE = 10000000;  // Number of iterations

    public static void main(String[] args) {
        IntPredicate isEven = num -> num % 2 == 0;

        long startTime = System.nanoTime();

        for (int i = 0; i < SIZE; i++) {
            isEven.test(i);
        }

        long endTime = System.nanoTime();
        long durationIntPredicate = endTime - startTime;

        System.out.println("IntPredicate execution time: " + durationIntPredicate + "
ns");
    }
}
//IntPredicate execution time: 10483609 ns
```

- Functional Interface and Its Primitive specializations
    - Predicate
        - IntPredicate
        - LongPredicate
        - DoublePredicate
    - Consumer
        - IntConsumer
        - LongConsumer
        - DoubleConsumer
    - BiConsumer<T, U>

        - ObjIntConsumer
        - ObjLongConsumer
        - ObjDoubleConsumer
    - Supplier
        - BooleanSupplier
        - IntSupplier
        - LongSupplier
        - DoubleSupplier
    - Function<T, R>
        - IntFunction
        - IntToDoubleFunction
        - IntToLongFunction
        - LongFunction
        - LongToDoubleFunction
        - LongToIntFunction
        - DoubleFunction
        - ToIntFunction
        - ToDoubleFunction
        - ToLongFunction
    - UnaryOperator
        - IntUnaryOperator
        - LongUnaryOperator
        - DoubleUnaryOperator
    - BiFunction<T, U, R>
        - ToIntBiFunction<T, U>
        - ToLongBiFunction<T, U>
        - ToDoubleBiFunction<T, U>
    - BinaryOperator
        - ToIntBiFunction<T, U>
        - ToLongBiFunction<T, U>
        - ToDoubleBiFunction<T, U>

Passing Lambda to the method

- A program that filters out empty strings from a given list of strings using a predicate.

```
class Program{
    private static List<String> filter( List<String> list, Predicate<String> p){
        List<String> result = new ArrayList<>( );
        for( String str : list ){
            if( p.test(str) )
                result.add(str);
        }
        return result;
    }

    public static void main(String[] args){
        List<String> listOfStrings = List.of("apple", "", "banana", " ", "cherry");

        List<String> nonEmptyStrinds = Program.filter( listOfStrings, str ->
!str.isEmpty());
```

```
    }
}
```

- A program to print elements from list using consumer.

```
class Program{
    private static void forEach( List<Integer> list, Consumer<Integer> c){
        for( Integer element : list )
            c.accept( element );
    }

    public static void main(String[] args){
        List<Integer> intList = Arrays.asList(1,2,3,4,5);
        Program.forEach( intList, number -> System.out.println(number))
    }
}
```

- A program to map list of Strings into list of integers using function.

```
class Program{
    private static List<Integer> map( List<String> list, Function<String> f){
        List<Integer> result = new ArrayList<>();

        int length;
        for(String str : list ){
            length = f.apply( str );
            result.add( length );
        }

        return result;
    }

    public static void main(String[] args){
        List<String> strings = Arrays.asList("Secrets", "of", "the", "Millionaire",
"Mind");
        List<Integer> intList = Program.map( strings, str -> str.length() );
        System.out.println( intList );
    }
}
```

Lambda Expression Under The Hood

- Let us consider instantiation of Thread using anonymous inner class:

```
class Program{  //Program.class
    public static void main(String[] args){
        //Anonymous inner class
        Runnable target = new Runnable(){   //Program$1.class
            @Override
            public void run(){
```

```
                System.out.println("Hello from run() method!");
            }
        }
        Thread thread = new Thread( target );
        thread.start();
    }
}
```

- As shown above, Java compiler generates .class file for anonymous inner class.
- Consider same implementation using Lambda expression:

```
class Program{  //Program.class
    public static void main(String[] args){
        Runnable target1 = () -> System.out.println("Hello from Lambda Expression 1");
        Thread thread1 = new Thread(target1);
        thread1.start();

        Runnable target2 = () -> System.out.println("Hello from Lambda Expression 2");
        Thread thread2 = new Thread(target2);
        thread2.start();
    }
}
```

- Let us discuss, what happens behind the scene. First we will see Bytecode of Program.class file

```
sandeep@Sandeeps-Mackbook-Air Demo1 % javap -c -p Program.class
Compiled from "Program.java"
class Program {
  Program();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: invokedynamic #7,  0              // InvokeDynamic #0:run:()Ljava/lang/Runnable;
       5: astore_1
       6: new           #11                 // class java/lang/Thread
       9: dup
      10: aload_1
      11: invokespecial #13                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
      14: astore_2
      15: aload_2
      16: invokevirtual #16                 // Method java/lang/Thread.start:()V
      19: invokedynamic #19,  0             // InvokeDynamic #1:run:()Ljava/lang/Runnable;
      24: astore_3
      25: new           #11                 // class java/lang/Thread
      28: dup
      29: aload_3
      30: invokespecial #13                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
      33: astore        4
      35: aload         4
      37: invokevirtual #16                 // Method java/lang/Thread.start:()V
      40: return

  private static void lambda$main$1();
    Code:
       0: getstatic     #20                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #26                 // String Hello from Lambda Expression 2
       5: invokevirtual #28                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return

  private static void lambda$main$0();
    Code:
       0: getstatic     #20                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #34                 // String Hello from Lambda Expression 1
       5: invokevirtual #28                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

- TODO

- Reference:
  - Lambda Expression, Behind the scene 1
  - Lambda Expression, Behind the scene 2

Method Reference

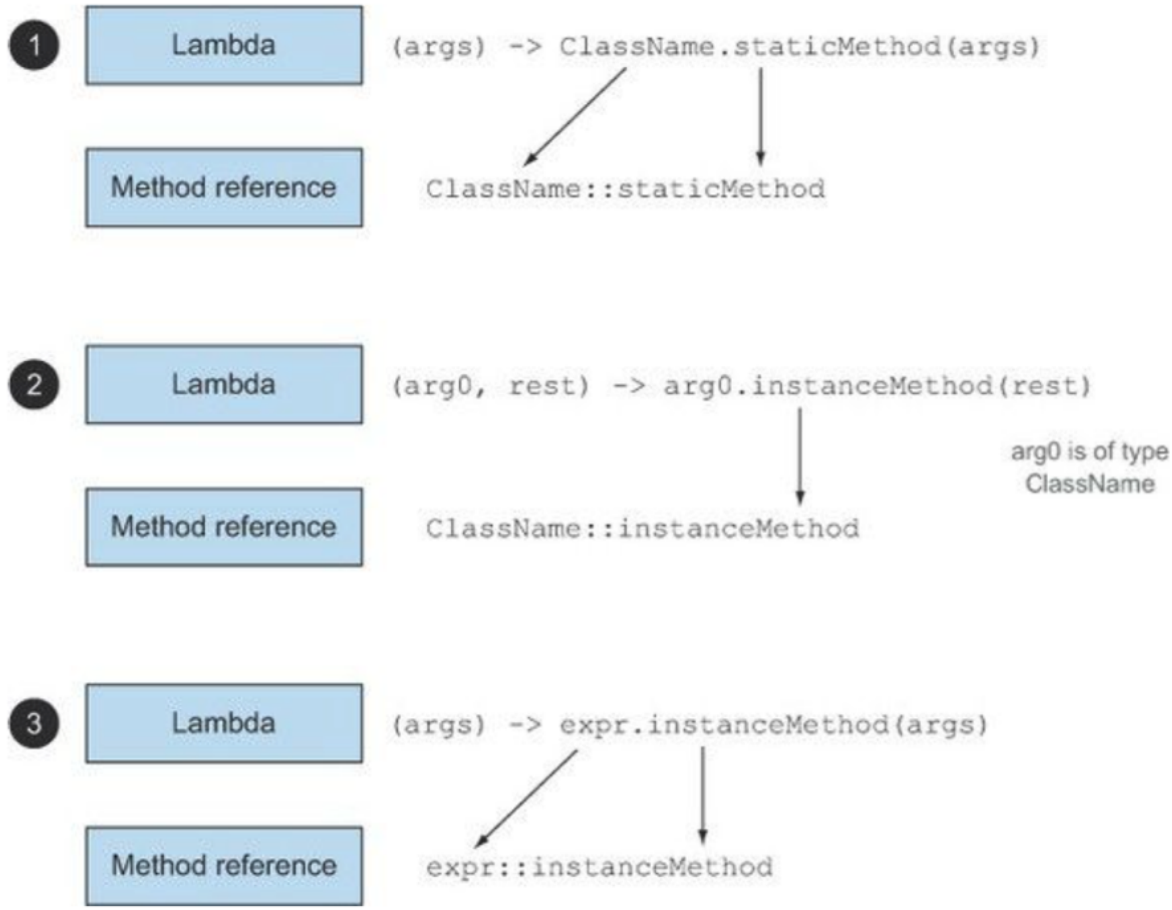
- Method references can be seen as shorthand for lambdas calling only a specific method. Method references let us reuse existing method definitions and pass them just like lambdas.
- In some cases method reference appear more readable and feel more natural than using lambda expressions. Consider below example:
- Using Lambda Expression:

```
//Let us assume getWeight() return Integer not int
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

- Using Method Reference

```
inventory.sort( Comparator.comparing( Apple::getWeight ) ); //or

import static java.util.Comparator.comparing;   //required
inventory.sort( comparing( Apple::getWeight ) );
```

- Method reference allows us to create lambda expression from existing method implementation. It improves code readability.
- When we need method reference then we should use **Target Reference::MethodName**.
  - Apple::getWeight
  - Remember that no brackets are needed because we are not actually calling the method.
- There are **three main kinds of method references**:
  1. A method reference to a static method (for example, the method parseInt of Integer, written Integer::parseInt)
  2. A method reference to an instance method of an existing object (for example, suppose we have a local variable expensiveTransaction of type Transaction, which supports an instance method getValue; we can write expensiveTransaction::getValue)
  3. A method reference to an instance method of an arbitrary type (for example, the method length of a String, written String::length)



- Example of a method reference to a static method:

```
package com.example;

import java.util.function.BiFunction;
import java.util.function.BinaryOperator;

public class Program {
    public static void main1(String[] args) {
        BiFunction<Integer, Integer, Integer> bf = (num1, num2 ) -> num1 + num2;
        int result = bf.apply(10, 20);
        System.out.println("Result  :   "+result);
    }

    public static void main2(String[] args) {
        BinaryOperator<Integer> bo = ( num1, num2 ) -> num1 + num2;
        int result = bo.apply(10, 20);
        System.out.println("Result  :   "+result);
    }

    private static int sum( int num1, int num2 ){
        return num1 + num2;
    }
    public static void main3(String[] args) {
        //BinaryOperator<Integer> bo = (num1,num2)->Program.sum(num1, num2);

        //Method Reference to Static Method
        BinaryOperator<Integer> bo = Program::sum;

        int result = bo.apply(10, 20);
```

```
        System.out.println("Result  :   "+result);
    }

    public static void main(String[] args) {
        //BinaryOperator<Integer> bo = (num1, num2) -> Integer.sum(num1,num2);

        //Method Reference to Static Method
        //public static int sum(int a, int b) method of Integer
        BinaryOperator<Integer> bo = Integer::sum;

        int result = bo.apply(10, 20);
        System.out.println("Result  :   "+result);
    }
}
```

- Example of a method reference to an instance method of an existing object

```
class Program{
    public static void main1(String[] args){
        List<Integer> intList = Arrays.asList(1,2,3,4,5);
        Program.forEach( intList, number -> System.out.println(number))
    }

    public static void main(String[] args){
        List<Integer> intList = Arrays.asList(1,2,3,4,5);
        Program.forEach( intList, System.out::println)
        //out is object reference and println is instance method
    }

    private static void forEach( List<Integer> list, Consumer<Integer> c){
        for( Integer element : list )
            c.accept( element );
    }
}
```

- First Example of a method reference to an instance method of an arbitrary type

```
class Program{
    public static void main(String[] args){
        //Using Lambda Expression
        Function<String, Integer> function = str -> str.length();
        Integer length = function.apply( str );
        System.out.println( "Length ::"+length );
    }
}
```

  - In lambda expression, str is an Input Parameter and Lambda body, length method is called on Input parameter. Hence evemn though it is instance method we can use String::length. Consider below code:

```
    class Program{
    public static void main(String[] args){
        //Using Method Reference
        Function<String, Integer> function = String::length;
        Integer length = function.apply( str );
        System.out.println( "Length ::"+length );
    }
}
```

- Second Example of a method reference to an instance method of an arbitrary type

```
class Employee{
    private String name;
    private int empid;
    private float salary;
    //TODO: define constructor's
    //TODO: getter and setter
}
class Program{
    public static void main(String[] args){
        //Function<Employee, String> function = emp -> emp.getName();
        Function<Employee, String> function = Employee::getName;

        Employee emp = new Employee("Soham",33,45000.50f);
        String name = function.apply( emp );
        System.out.println( "Name ::"+name );
    }
}
```

- What are equivalent method references for the following lambda expressions?
    - Function<String, Integer> stringToInteger = (String s) -> Integer.parseInt(s);
        - Function<String, Integer> stringToInteger = Integer::parseInt;
    - BiPredicate<List, String> contains = (list, element) -> list.contains(element);
        - BiPredicate<List, String> contains = List::contains;
- We should use a method reference to make your code slightly less verbose.

Constructor references

- We can create a reference to an existing constructor using its name and the keyword new as follows:
    - ClassName::new.
- java.util.function.Supplier is a functional interface and "T get()" is functional method. Consider class Apple as follows

```
class Apple{
    private String color;
    private int weight;
    public Apple(){
        this("None", 0);
```

```
    }
    public Apple(int weight){
        this("None", weight);
    }
    public Apple(String color, int weight){
        this.color = color;
        this.weight = weight;
    }
    //TODO: getter/setter and toString method
}
```

```
class Program{
    public static void main(String[] args){
        //Lambda exression with { }
        Supplier<Apple> supplier = ( )->{
            Apple apple = new Apple();
            return apple;
        };

        Apple a = supplier.get();
    }
}
```

```
class Program{
    public static void main(String[] args){
        //Lambda exression without return statement
        Supplier<Apple> supplier = ( ) -> new Apple( );

        Apple a = supplier.get( );
    }
}
```

```
class Program{
    public static void main(String[] args){
        //Constructor reference
        Supplier<Apple> supplier = Apple::new;
        //get method do not take any parameter
        //hence here Apple's parameterless ctor will class

        Apple apple = supplier.get( );
    }
}
```

- java.util.function.BiFunction<T,U,R> is Functional interface and "R apply(T t,U u)" is functional method.

```
class Program{
    public static void main(String[] args){
        //Lambda Expression with { } and return statement
```

```
        BiFunction<String, Integer, Apple> biFunction = ( color, weight ) -> {
            Apple apple = new Apple( name, color );
            return apple;
        };

        Apple apple = biFunction.apply( color, weight );
    }
}
```

```
class Program{
    public static void main(String[] args){
        //Lambda Expression without { } and without return statement
        BiFunction<String, Integer, Apple> biFunction = ( color, weight ) -> new Apple(
name, color );
        Apple apple = biFunction.apply( color, weight );
    }
}
```

```
class Program{
    public static void main(String[] args){
        //Constructor reference
        BiFunction<String, Integer, Apple> biFunction = Apple::new;
        //apply method take 2 parameters: String, int
        //Hence here Apple's 2 parameter constructor will call.

        Apple apple = biFunction.apply( color, weight );
    }
}
```

- In the following code, each element of a List of Integers is passed to the constructor of Apple using a map method, resulting in a List of apples with different weights:

```
class Program{
    private static List<Apple> map(List<Integer> list, Function<Integer, Apple>
function){
        List<Apple> result = new ArrayList<>();
        for(Integer element : list){
            Apple apple = function.apply( element );
            result.add( element ); //Giving call to constructor
        }
        return result;
    }
    public static void main(String[] args){
        List<Integer> weights = Arrays.asList(10,3,8,4);
        List<Apple> apples = Program.map( weights, Apple::new);
        System.out.println(apples);
    }
}
```

- Using Supplier, we invoked parameterless constructor, using Function, we invoked 1 parameter constructor and using BiFunction, we invoked 2 parameter constructor.
- In other cases, we need to define functional interface to fullfil the requirement. Consider below code:

```
interface TriFunction<T,U,V,R>{
    R apply( T t, U u, V v);
}
```

```
class Program{
    public static void main(String[] args){
        TriFunction<String, Integer, Float> function = Employee::new;
        Employee emp = function.apply("Soham", 33, 45000.50f);
        //Here 3 parameter ctor of class Employee will call.
    }
}
```

- Putting lambdas and method references into practice!

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

class Apple {
    private Integer weight;
    private String color;

    public Apple(Integer weight, String color) {
        this.weight = weight;
        this.color = color;
    }

    public Integer getWeight() {
        return this.weight;
    }

    public String getColor() {
        return this.color;
    }

    @Override
    public String toString() {
        return this.color+" "+this.weight;
    }
}
```

- Let us sort apple inventory using "void sort(Comparator<? super E> c)" method of java.util.List.
- First Solution

```
class AppleComparator implements Comparator<Apple> {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
}

public class Program {
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(new Apple(80,"green"),
                                             new Apple(155, "green"),
                                             new Apple(120, "red"));


        inventoy.sort( new AppleComparator() );

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

- Second Solution: Using anonymous class

```
public class Program {
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(new Apple(80,"green"),
                                             new Apple(155, "green"),
                                             new Apple(120, "red"));


        inventoy.sort(new Comparator<Apple>() {
            @Override
            public int compare(Apple a1, Apple a2) {
                return a1.getWeight().compareTo(a2.getWeight());
            }
        });

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

- Third Solution: Using lambda expression

```
public class Program {
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(new Apple(80,"green"),
                                             new Apple(155, "green"),
                                             new Apple(120, "red"));


        //inventoy.sort( (Apple a1, Apple a2) ->
a1.getWeight().compareTo(a2.getWeight()));
        inventoy.sort( (a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

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

- Comparator has a static helper method called comparing that takes a Function extracting a Comparable key and produces a Comparator object
    - static <T,U extends Comparable<? super U>> Comparator comparing(Function<? super T,? extends U> keyExtractor)

```
public class Program {
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(new Apple(80,"green"),
                                             new Apple(155, "green"),
                                             new Apple(120, "red"));


        //inventoy.sort( (Apple a1, Apple a2) ->
a1.getWeight().compareTo(a2.getWeight()));
        //inventoy.sort( (a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
        //inventoy.sort( Comparator.comparing( apple -> apple.getWeight()) );
        inventoy.sort( comparing( apple -> apple.getWeight()) );
        //import static java.util.Comparator.comparing; is required

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

- Fourth Solution: using method reference

```
public class Program {
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(new Apple(80,"green"),
                                             new Apple(155, "green"),
                                             new Apple(120, "red"));


        inventoy.sort( comparing( Apple::getWeight ) );
        //inventoy.sort( comparing( Apple::getWeight ).reversed() );sort by decreasing
weight

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

- **Chaining Comparators**:
    - Two apples are compared based on their weight, we may want to sort them by color. The thenComparing method allows us to do just that.

```
public class Program {
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(new Apple(80,"green"),
                                             new Apple(155, "red"),
                                             new Apple(155, "green"));


        //inventoy.sort( comparing( Apple::getWeight ) );
        //[green 80, red 155, green 155]

        //inventoy.sort( comparing( Apple::getWeight ).reversed());
        //[red 155, green 155, green 80]

        inventoy.sort( comparing( Apple::getWeight
).reversed().thenComparing(Apple::getColor));
        //[green 155, red 155, green 80]

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

- **Composing Predicates**
    - Below are the default methods of java.util.function.Predicate interface:
        - default Predicate negate()
        - default Predicate and(Predicate<? super T> other)
        - default Predicate or(Predicate<? super T> other)
- Filter apple which are not green using negate() method.

```
public class Program {
    private static List<Apple> filter( List<Apple> list, Predicate<Apple> p){
        List<Apple> result = new ArrayList<>();
        for( Apple apple : list ){
            if( p.test(apple))
                result.add( apple );
        }
        return result;
    }
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(
                new Apple(150, "green"),
                new Apple(200,"red"),
                new Apple(180,"green"),
                new Apple(120,"yellow")
        );


        Predicate<Apple> isGreen = apple -> apple.getColor().equals("green");

        List<Apple> greenApples = Program.filter( inventoy, isGreen );
        System.out.println( greenApples );//[green 150, green 180]

        List<Apple> notGreenApples = Program.filter( inventoy, isGreen.negate() );
```

```
        System.out.println( notGreenApples );//[red 200, yellow 120]
    }
}
```

- We can combine the resulting predicate one step further to express apples that are green and heavy (above 150 g) or just red apples:

```
public class Program {
    private static List<Apple> filter( List<Apple> list, Predicate<Apple> p){
        List<Apple> result = new ArrayList<>();
        for( Apple apple : list ){
            if( p.test(apple))
                result.add( apple );
        }
        return result;
    }
    public static void main(String[] args) {
        List<Apple> inventoy = Arrays.asList(
                new Apple(150, "green"),
                new Apple(200,"red"),
                new Apple(180,"green"),
                new Apple(120,"yellow")
        );


        Predicate<Apple> isGreen = apple -> apple.getColor().equals("green");

        Predicate<Apple> greenAndHeavyOrRed;
        greenAndHeavyOrRed = isGreen.and( apple -> apple.getWeight() > 150 ).or(apple -
> apple.getColor().equals("red"));

        List<Apple> apples = Program.filter( inventoy, greenAndHeavyOrRed );

        System.out.println( apples );//[red 200, green 180]
    }
}
```

Behavior Parameterization

- In Programming, we can pass code/method/actions as a argument to the another method. It is called as Behavior Parameterization.
- Behavior Parameterization makes code more flexible.
- In C Programming, we use function pointer to achieve behavior Parameterization.
    - Function Pointer Example 1:

```
#include<stdio.h>
void send_email( const char *message ){
    printf("Sending Email: %s\n", message);
}

int main( void ){
    const char* message = "Hello, this is a test notification.";
```

```
    void (*notification)( const char* );
    //notification is a function pointer which can store address of any function
    //which acccept const char* type parameter
    //and whose return  type is void

    //notification = &send_email;   //or
    notification = send_email;

    //(*notification)( message );   //or
    notification( message );

    return 0;
}
```

- Function Pointer Example 2:

```
#include<stdio.h>
void send_email( const char *message ){
    printf("Sending Email: %s\n", message);
}

void send_sms( const char *message ){
    printf("Sending SMS: %s\n", message);
}

int main( void ){
    const char* message = "Hello, this is a test notification.";

    void (*notification)( const char* ) = NULL;

    notification = send_email;
    notification( message );

    notification = send_sms;
    notification( message );
    return 0;
}
```

- Function Pointer Example 3:

```
#include <stdio.h>
void send_email(const char *message) {
    printf("Sending Email: %s\n", message);
}

void send_sms(const char *message) {
    printf("Sending SMS: %s\n", message);
}

int main(void) {
    const char* email_message = "Hello, this is a test notification for Email.";
    const char* sms_message = "Hello, this is a test notification for SMS.";
```

```
    typedef void (*notification_t)(const char*);
    //typedef is used to give another name to the existing data type
    notification_t emailNotification = send_email;
    emailNotification(email_message);

    notification_t smsNotification = send_sms;
    smsNotification(sms_message);

    return 0;
}
```

- Function Pointer Example 4:

```
#include <stdio.h>
typedef void (*notification_t)(const char*);

void send_email(const char *message) {
    printf("Sending Email: %s\n", message);
}

void send_sms(const char *message) {
    printf("Sending SMS: %s\n", message);
}

//Behavior parameterization achieved using function pointer
void execute(notification_t event, const char *message) {
    printf("Business logic is executed\n");
    if (event != NULL) {
        event(message);
    }
}

int main(void) {
    const char* email_message = "Email notification sent!!.";
    const char* sms_message = "SMS notification sent!!";

    //send_email is a function passed as an argument: Behavior parameterization
    execute(send_email, email_message);

    //send_sms is a function passed as an argument: Behavior parameterization
    execute(send_sms, sms_message);

    execute(NULL, NULL);

    return 0;
}
```

- In Java Programming, we use functional interface and lambda expression/method reference to achieve behavior Parameterization.
    - Lets first define functional interface:

```
@FunctionalInterface
interface Notification {
    void notify(String message);
}
```

- Lambda Expression Example 1:

```
class Program{
    public static void main(String[] args){
        //Lambda Expression
        Notification emailNotification = message -> System.out.println( message);
        String message = "Hello, this is a test notification for Email";
        emailNotification.notify( message );

        //Lambda Expression
        Notification smsNotification = message -> System.out.println( message);
        message = "Hello, this is a test notification for SMS";
        emailNotification.notify( message );
    }
}
```

- Lambda Expression Example 2:

```
class Program{
    public static void execute( Notification event, String message ){
        System.out.println("Business logic is executed");
        if( event != null )
            event.notify( message );
    }
    public static void main(String[] args){
        //Lambda Expression
        Notification emailNotification = message -> System.out.println( message);
        String message = "Hello, this is a test notification for Email";
        execute( emailNotification, message );

        //Lambda Expression
        Notification smsNotification = message -> System.out.println( message);
        message = "Hello, this is a test notification for SMS";
        execute( smsNotification, message );
    }
}
```

- Lambda Expression Example 3:

```
class Program{
    //Behavior parameterization achieved using functional interface. Here
Notification
    public static void execute( Notification event, String message ){
        System.out.println("Business logic is executed");
        if( event != null )
```

```
            event.notify( message );
    }
    public static void main(String[] args){
        //message -> System.out.println( message) passed as an argument: Behavior
parameterization
        execute( message -> System.out.println( message), "Email notification
sent!!." );

        //message -> System.out.println( message) passed as an argument: Behavior
parameterization
        execute( message -> System.out.println( message), "SMS notification
sent!!." );
    }
}
```

- Let us first define NotificationImpl class

```
class NotificationImpl{
    public static void sendEmail( String message ){
        System.out.println(message);
    }
    public static void sendSms( String message ){
        System.out.println(message);
    }
}
```

- Method reference Example 1:

```
class Program{
    public static void main(String[] args){
        //Method Reference
        Notification emailNotification = NotificationImpl::sendEmail;
        String message = "Hello, this is a test notification for Email";
        emailNotification.notify( message );

        //Method Reference
        Notification smsNotification = NotificationImpl::sendSms;
        message = "Hello, this is a test notification for SMS";
        emailNotification.notify( message );
    }
}
```

- Method reference Example 2:

```
class Program{
    public static void execute( Notification event, String message ){
        System.out.println("Business logic is executed")
        if( event != null )
            event.notify( message );
    }
    public static void main(String[] args){
```

```
        //Method reference
        Notification emailNotification = NotificationImpl::sendEmail;
        String message = "Hello, this is a test notification for Email";
        execute( emailNotification, message );

        //Method reference
        Notification smsNotification = NotificationImpl::sendSms;
        message = "Hello, this is a test notification for SMS";
        execute( smsNotification, message );
    }
}
```

- Method reference Example 3:

```
class Program{
    //Behavior parameterization achieved using functional interface.
    public static void execute( Notification event, String message ){
        System.out.println("Business logic is executed");
        if( event != null )
            event.notify( message );
    }
    public static void main(String[] args){
        //NotificationImpl::sendEmail is an argument: Behavior parameterization
        execute( NotificationImpl::sendEmail, "Email notification sent!!." );

        //NotificationImpl::sendSms is an argument: Behavior parameterization
        execute( NotificationImpl::sendSms, "SMS notification sent!!." );
    }
}
```

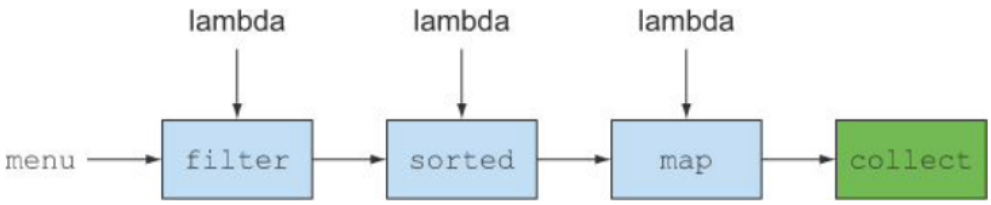
Java 8 Streams API

- A sequence of elements from a source that supports data processing operations is called as stream.
  - **Sequence of elements**:
    - A stream is similar to a collection, but it is not a data structure (like a list or array). Instead, it represents a sequence of values of a specific type (e.g., numbers, strings).
    - Collections are used to store data and streams are used to process data.
  - **Source**:
    - A stream gets its elements from a source like a collection, an array, or other I/O resources.
  - **Data processing operations**:
    - Streams support database-like operations and common operations from functional programming languages to manipulate data like:
      - **Filter**: Remove certain elements.
      - **Map**: Transform elements into something else.
      - **Sort**: Arrange elements in a specific order.
      - **Reduce**: Combine elements into a single result.
  - **Characteristics of Stream Operations**:
    - **Pipelining**:
      - Stream operations can be chained together. For example:

```
```

```
List<String> names = Arrays.asList("Prathamesh", "Soham", "Diyam",
"Sarvesh");

// Stream to filter names starting with "S", then convert them to
uppercase
List<String> result = names.stream()
                           .filter(name -> name.startsWith("S"))
                           .map(name -> name.toUpperCase)
                           .collect(Collectors.toList());

System.out.println(result);  // Output: [SOHAM, SARVESH]
```

- "names.stream().filter(..).map(..).collect(..)" is called as pipeline.The result of one operation becomes the input for the next, just like a chain.



- **Internal Iteration**:
  - In collections, we manually iterate through the data using loops or iterators. In streams, iteration is done automatically behind the scenes.
  - This makes the code more concise and easy to read.

```
for(Dish d : menus)//External Iteration
    System.out.println(d.toString());
```

```
menus.stream().forEach(System.out::println);//Internal Iteration
```

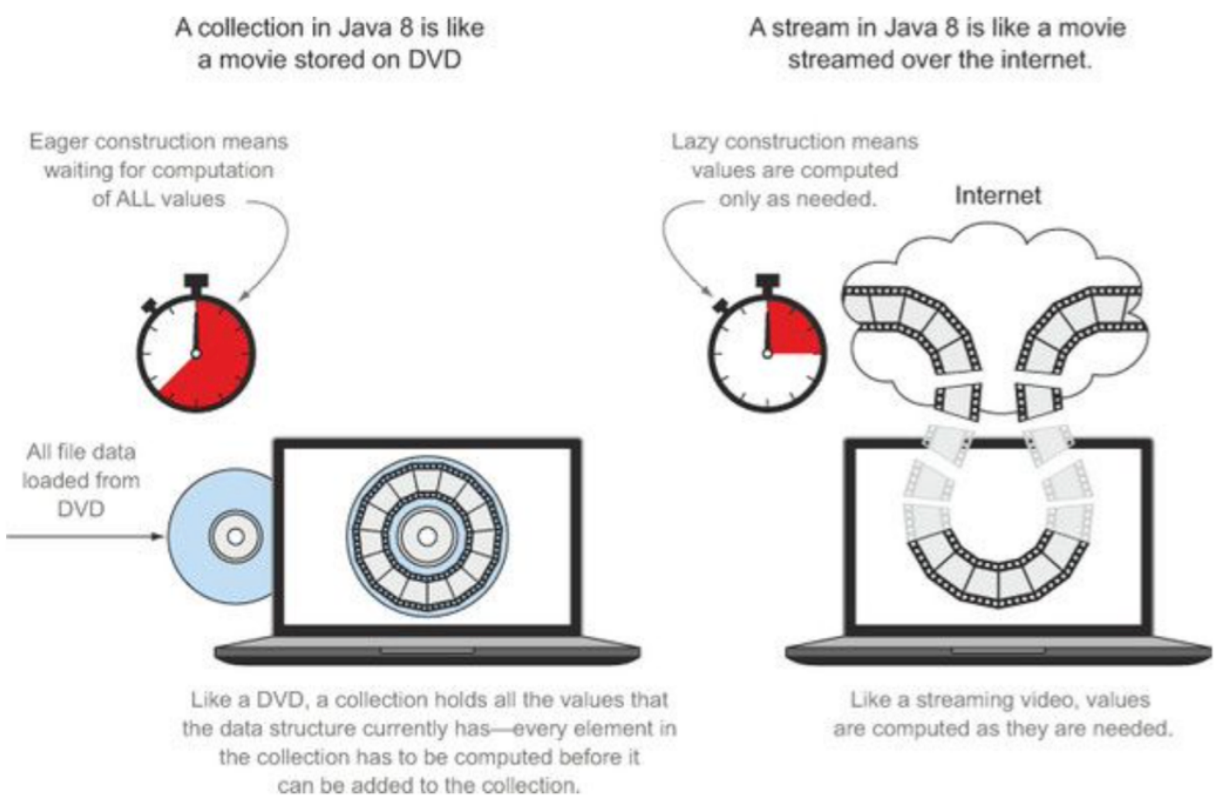
Streams vs. collections

- Consider class Dish to understand Streams.

```
package com.example;

import java.util.Arrays;
import java.util.List;

class Dish {
    //Nested Type
    public enum Type {
        MEAT, FISH, OTHER;
    }

    //Fields
```

```
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public Dish(String name, boolean vegetarian, int calories, Type type) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;
    }

    public String getName() {
        return this.name;
    }

    public boolean isVegetarian() {
        return this.vegetarian;
    }

    public int getCalories() {
        return this.calories;
    }

    public Type getType() {
        return this.type;
    }

    @Override
    public String toString() {
        return this.name;
    }

    public static final List<Dish> menu =
            Arrays.asList( new Dish("pork", false, 800, Dish.Type.MEAT),

                    new Dish("beef", false, 700, Dish.Type.MEAT),

                    new Dish("chicken", false, 400, Dish.Type.MEAT),

                    new Dish("french fries", true, 530, Dish.Type.OTHER),

                    new Dish("rice", true, 350, Dish.Type.OTHER),

                    new Dish("season fruit", true, 120, Dish.Type.OTHER),

                    new Dish("pizza", true, 550, Dish.Type.OTHER),

                    new Dish("prawns", false, 400, Dish.Type.FISH),

                    new Dish("salmon", false, 450, Dish.Type.FISH));
}
```

- Both streams and collections are ways to store and process sequences of data, but they work in very different ways.



- **Data Storage**:
  - Collection:
    - Think of a collection like a full container of data. For example, imagine we have a DVD that contains the entire movie (all the data is already there).
    - The collection keeps everything in memory. All the elements (values) are available, and we can access them at any time.
  - Stream:
    - A stream is like streaming a movie over the internet. The video is not fully available at once. we download and watch small parts of the movie at a time.
    - A stream does not store all the data in memory. It sends the data on demand (when we need it), and it may not even be available in its entirety
- **Data Computation**:
  - Collection
    - A collection holds all values in memory. We can access all the elements right away. If we want to add or remove elements, we do so immediately, and all elements are computed and stored before they can be used.
    - This is called as eager evluation.
  - Stream
    - A stream does not keep all data in memory. It computes values only when needed. For example, imagine we want to watch a movie; it doesn't download the entire movie at once. It downloads only the part we're currently watching, and the rest is downloaded as we go.
    - This is called lazy evaluation. The stream only produces values when we request them.
- **Changing Data**
  - Collection
    - We can add or remove elements from a collection any time. A collection is like a warehouse that we fill up with data and keep it there.
  - Stream
    - We cannot change a stream (no adding or removing elements). A stream is like a fixed pipeline of data: we get values in a specific order, and we can only consume them. If we want to process the next value, we have to ask for it.

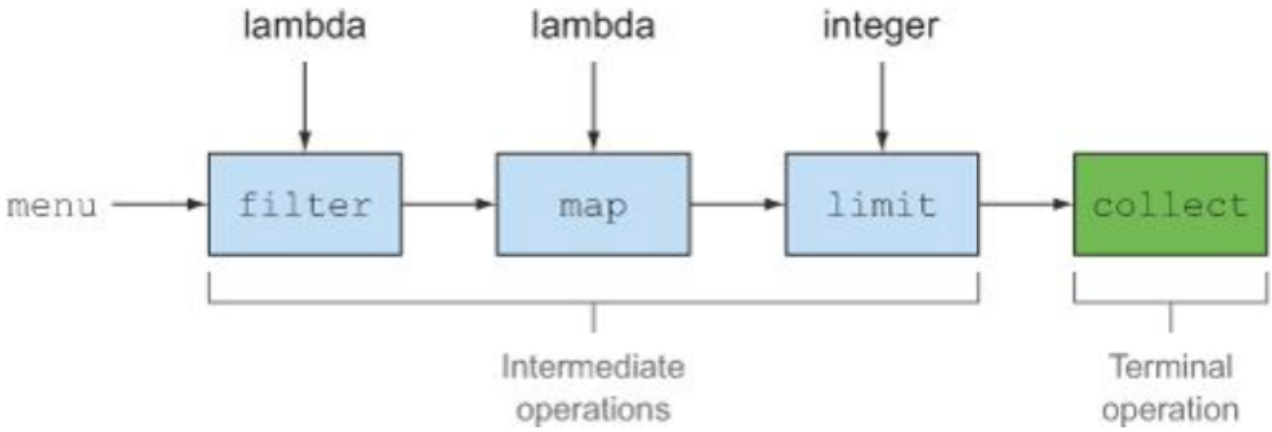
- **Performance**
  - Collection
    - A collection can take up more memory because it holds all the data at once. This might not be efficient for very large datasets.
  - Stream
    - A stream is more efficient when dealing with large or infinite datasets because it computes only what we need, one piece at a time.

Stream operations

- java.util.stream.Stream is a interface. It defines many operations. Click Here to get details of Stream API.
- Consider below example:

```
import static com.example.Dish.menu;
import java.util.stream.Collectors;
public class Program {
    public static void main(String[] args) {
        List<String> names = menu.stream()//Get Stream from collection
                .filter(d -> d.getCalories() > 300)//intermediate operation
                .map(Dish::getName)//intermediate operation
                .limit(3)//intermediate operation
                .collect(Collectors.toList());//terminal operation

        System.out.println( names );//[pork, beef, chicken]
    }
}
```

- Stream operations that can be connected are called intermediate operations, and operations that close a stream are called terminal operations.



Sr.No.	Operation	Type	Return Type	Argument of the Operation	Function Descriptor
1	filter	**Intermediate**	Stream<T>	Predicate<T>	T -> boolean
2	map	**Intermediate**	Stream<T>	Function<T,R>	T -> R
3	limit	**Intermediate**	Stream<T>	long	None
4	sorted	**Intermediate**	Stream<T>	Comparator<T>	(T,T) -> int
5	distinct	**Intermediate**	Stream<T>	None	None
6	forEach	**Terminal**	void	Consumer<T>	T -> void
7	count	**Terminal**	long	None	None
8	collect	**Terminal**	R	Collector<? super T,A,R>	

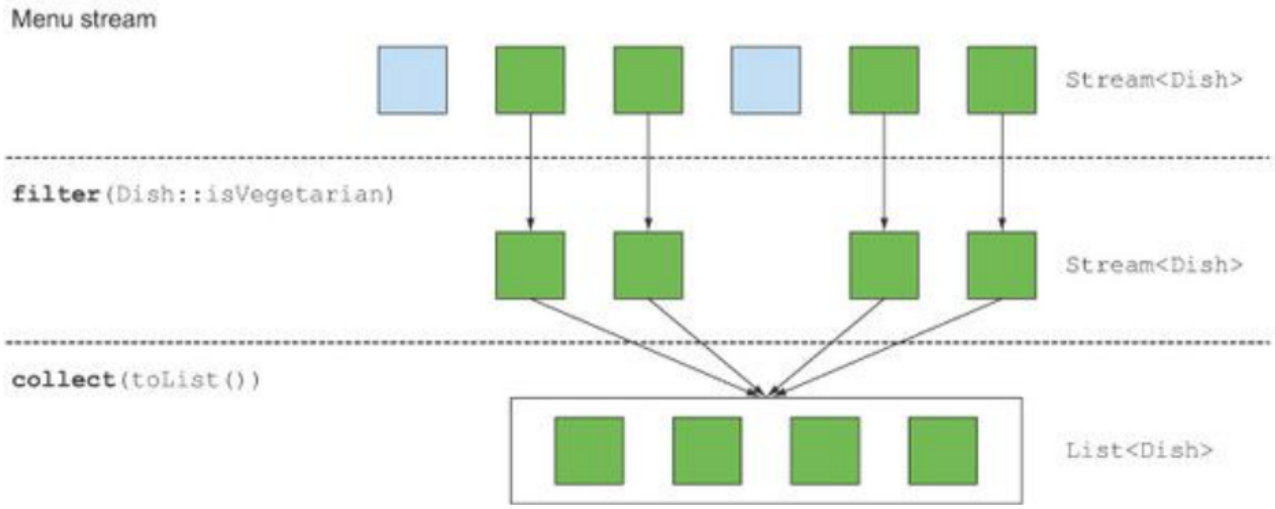
- **Filtering**
  - "Stream filter(Predicate<? super T> predicate)" is a method of java.util.stream.Stream interface.
- Create a vegetarian menu by filtering all vegetarian dishes.

```
import static com.example.Dish.menu;
import java.util.stream.Collectors;
public class Program {
    public static void main(String[] args) {
        Predicate<Dish> dishPredicate = d -> d.isVegetarian() == true;
        Collector<Dish, ?, List<Dish>> collector = Collectors.toList();

        List<Dish> vegetarianMenu = menu.stream()
                .filter(dishPredicate)
                .collect(collector);

        System.out.println( vegetarianMenu );
        //[french fries, rice, season fruit, pizza]
    }
}
```

```
import static com.example.Dish.menu;
import java.util.stream.Collectors;
public class Program {
    public static void main(String[] args) {
        List<Dish> vegetarianMenu = menu.stream()
                .filter(d -> d.isVegetarian() == true)
                .collect(Collectors.toList());

        System.out.println( vegetarianMenu );
        //[french fries, rice, season fruit, pizza]
    }
}
```

```
import static com.example.Dish.menu;
import java.util.stream.Collectors;
public class Program {
    public static void main(String[] args) {
        List<Dish> vegetarianMenu = menu.stream()
                .filter(Dish::isVegetarian)
                .peek(dish -> System.out.println("Filtered dish: "+dish+" isVegetarian-
"+dish.isVegetarian()))
                .collect(Collectors.toList());

        System.out.println( vegetarianMenu );
        //[french fries, rice, season fruit, pizza]
    }
}
```

- peek() allows us to inspect the elements as they are processed by the stream. we can see what's happening at each stage without interfering with the processing.This is perfect for logging intermediate values without altering the stream processing flow.



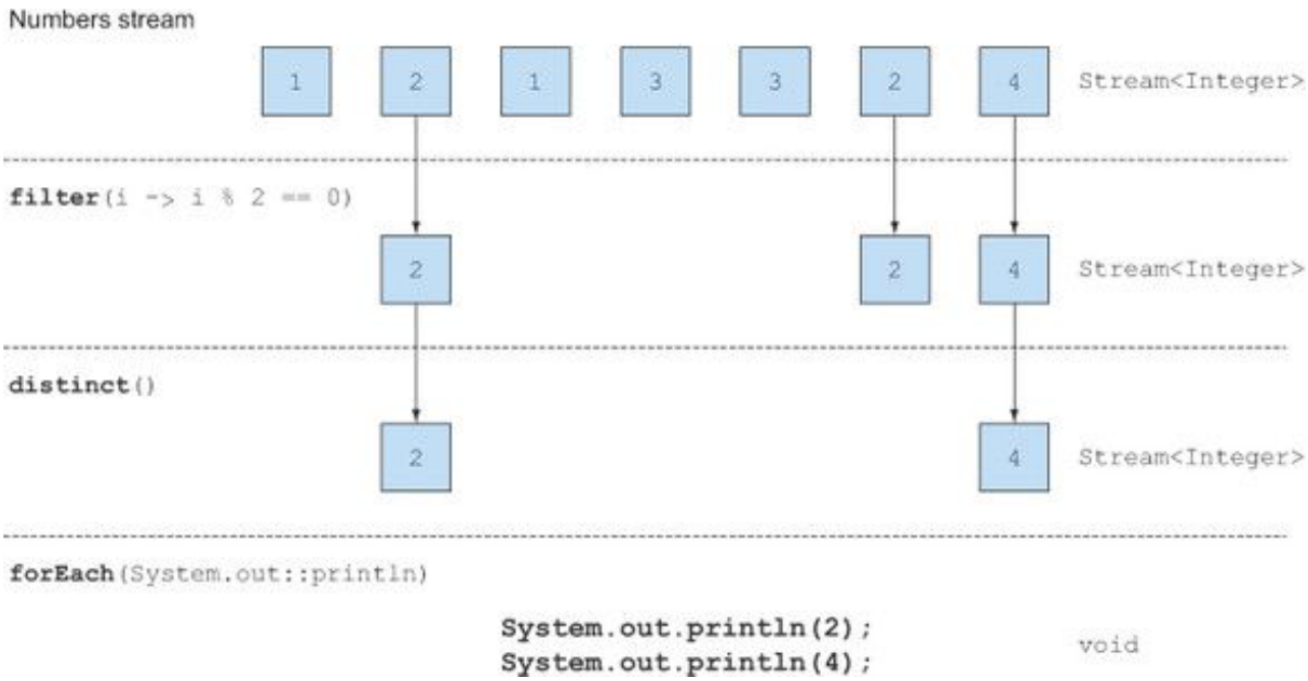
- Filter all even numbers from a list and makes sure that there are no duplicate.
    - "Stream distinct()" is a method of java.util.stream.Stream interface. It is used to get unique elements in stream.

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);

        List<Integer> evenNumbers = numbers.stream()
                    .filter(i -> i % 2 == 0)
                    .collect(Collectors.toList());

        System.out.println( evenNumbers );//[2, 2, 4]
    }
}
```

```
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);

        List<Integer> evenNumbers = numbers.stream()
                    .filter(i -> i % 2 == 0)
                    .distinct()
                    .collect(toList());

        System.out.println( evenNumbers );  //[2, 4]
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
        Consumer<Integer> consumer = number -> System.out.println( number );
        numbers.stream()
                    .filter(i -> i % 2 == 0)
                    .distinct()
                    .forEach(consumer);  //[2, 4]
    }
}
```

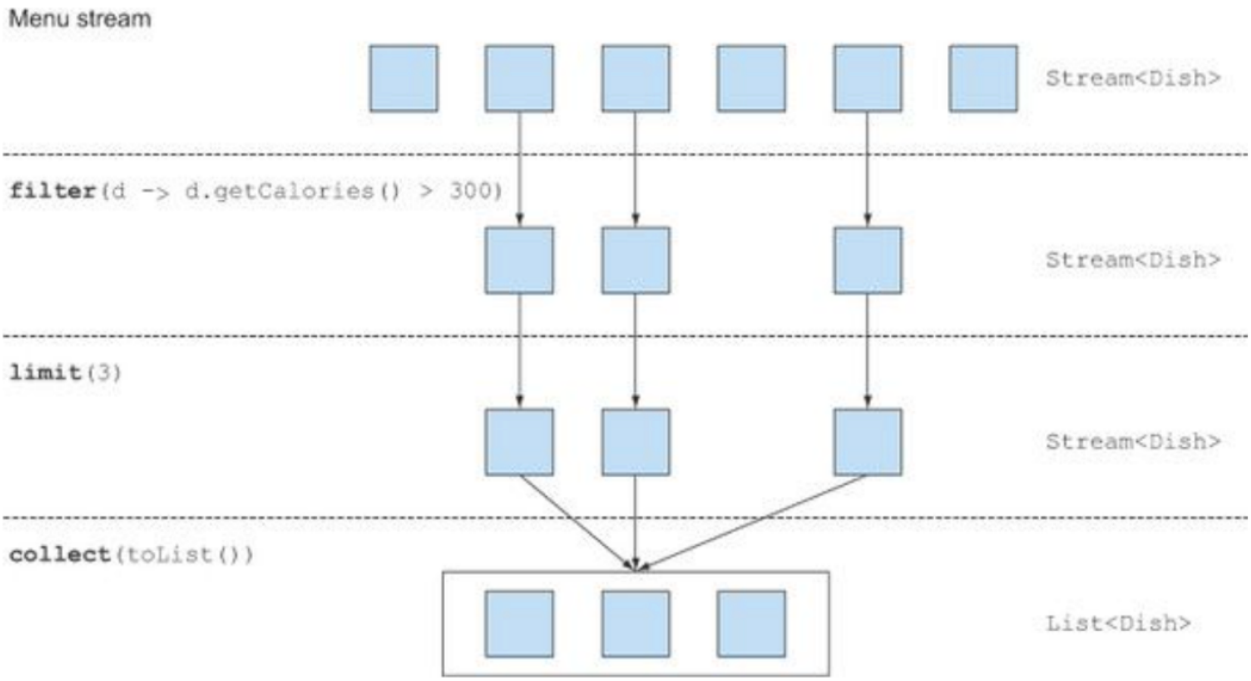
```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
        numbers.stream()
                    .filter(i -> i % 2 == 0)
                    .distinct()
                    .forEach(System.out::println);  //[2, 4]
    }
}
```



- Create a List by selecting the first three dishes that have more than 300 calories.
    - Stream limit(long maxSize) is a method of java.util.stream.Stream interface. It is used to truncate stream.

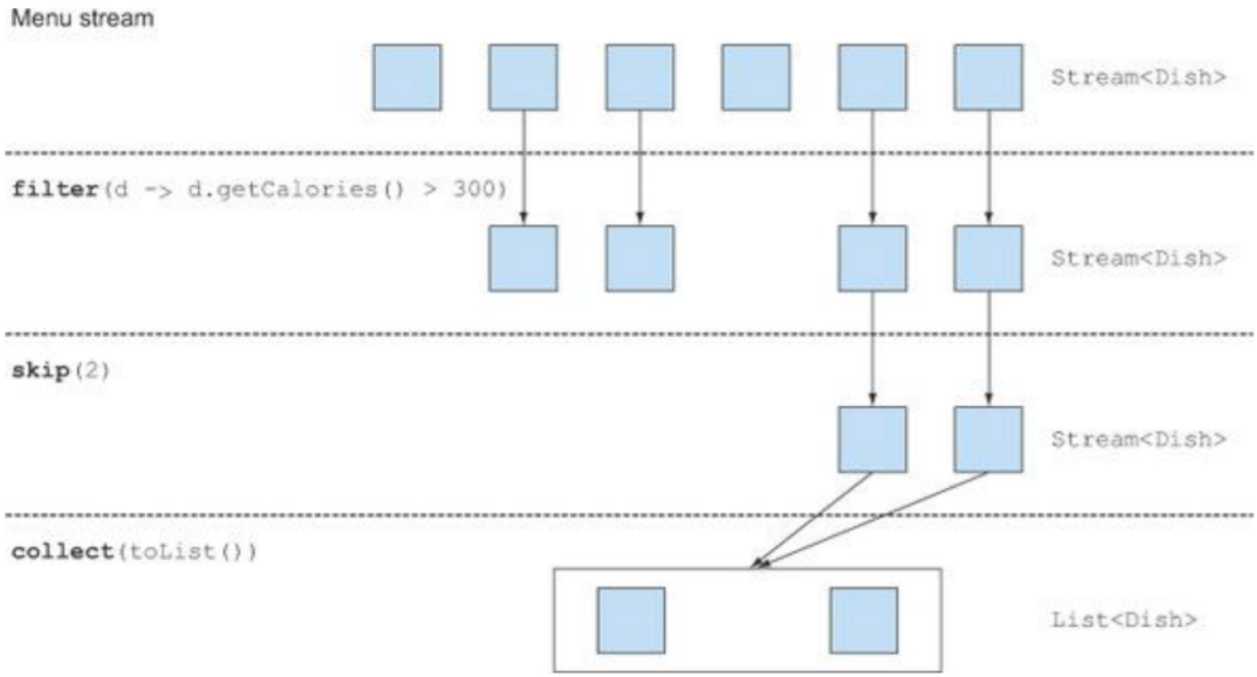
```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<Dish> dishes = menu.stream()
                .filter(d -> d.getCalories() > 300)
                .collect(toList());

        System.out.println(dishes);
        //[pork, beef, chicken, french fries, rice, pizza, prawns, salmon]
    }
}
```

```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<Dish> dishes = menu.stream()
                .filter(d -> d.getCalories() > 300)
                .limit(2)
                .collect(toList());

        System.out.println(dishes);
        //[pork, beef, chicken]
    }
}
```



- Skip the first two dishes that have more than 300 calories and returns the rest.
    - "Stream skip(long n)" is a method of java.util.stream.Stream interface. It returns a stream that discards the first n elements
    - If the stream has fewer elements than n, then an empty stream is returned. Note that limit(n) and skip(n) are complementary!

```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<Dish> dishes = menu.stream()
                .filter( d -> d.getCalories() > 300 )
                .skip( 2 )
                .collect(toList());

        System.out.println(dishes);
        //[chicken, french fries, rice, pizza, prawns, salmon]
    }
}
```



- How would you use streams to filter the first two meat dishes?

```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<Dish> dishes = menu.stream()
                .filter( d -> d.getType() == Dish.Type.MEAT )
                .limit( 2 )
                .collect(toList());

        System.out.println(dishes);
        //[pork, beef]
    }
}
```

Mapping

- Consider SQL to select a particular column from a table.

```
SELECT dish_name FROM menu;
```

- The Streams API provides similar facilities through the map and flatMap method to select information from certain objects.
- " Stream map(Function<? super T,? extends R> mapper)" is a method of java.util.stream.Stream interface.
- The term "mapping" is used because it refers to transforming an element into a new version, rather than simply modifying the original.
- Extract the names of the dishes in the stream:

```
```

```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        Function<Dish, String> function = dish -> dish.getName();
        List<String> names = menu.stream()
                .map(function)
                .collect(toList());

        System.out.println( names );
        //[pork, beef, chicken, french fries, rice, season fruit, pizza, prawns,
salmon]
    }
}
```

```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<String> names = menu.stream()
                .map(dish -> dish.getName())//Using Lambda Expression
                .collect(toList());

        System.out.println( names );
        //[pork, beef, chicken, french fries, rice, season fruit, pizza, prawns,
salmon]
    }
}
```

```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<String> names = menu.stream()
                .map(Dish::getName)//Using method reference
                .collect(toList());

        System.out.println( names );
        //[pork, beef, chicken, french fries, rice, season fruit, pizza, prawns,
salmon]
    }
}
```

- Given a list of words, we'd like to return a list of the number of characters for each word.

```
import java.util.Arrays;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
```

```
        List<String> words = Arrays.asList("Create", "the", "future", "don't", "just",
"change", "the", "past.");
        List<Integer> wordLengths = words.stream()
                //.map(word -> word.length())//or
                .map(String::length)
                .collect(toList());
        System.out.println( wordLengths);//[6, 3, 6, 5, 4, 6, 3, 5]
    }
}
```

- Find out the length of the name of each dish?

```
import static com.example.Dish.menu;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<Integer> dishNameLengths = menu.stream()
                //.filter( dish -> dish.getName())//or
                .map(Dish::getName)
                //.map( name -> name.length())//or
                .map(String::length)
                .collect(toList());

        System.out.println( dishNameLengths );
        //[4, 4, 7, 12, 4, 12, 5, 6, 6]
    }
}
```

Flattening streams

- " Stream flatMap(Function<? super T,? extends Stream<? extends R>> mapper)" is a method of java.util.stream.Stream interface.
- How flatMap works?
    - Transformation:
        - It transforms each element of the original stream into a new stream of elements.
    - Flattening:
        - The resulting streams from all elements are then flattened into a single stream of elements.
- Suppose we have a stream of lists, and we want to flatten these lists into a single stream of elements.

```
public class Program {
    public static void main(String[] args) {
        List<List<Integer>> listOfLists = Arrays.asList(
                Arrays.asList(1, 2, 3),
                Arrays.asList(4, 5),
                Arrays.asList(6, 7, 8)
        );

        listOfLists.stream()
                .forEach( System.out::println);
        //[1, 2, 3]
        //[4, 5]
```

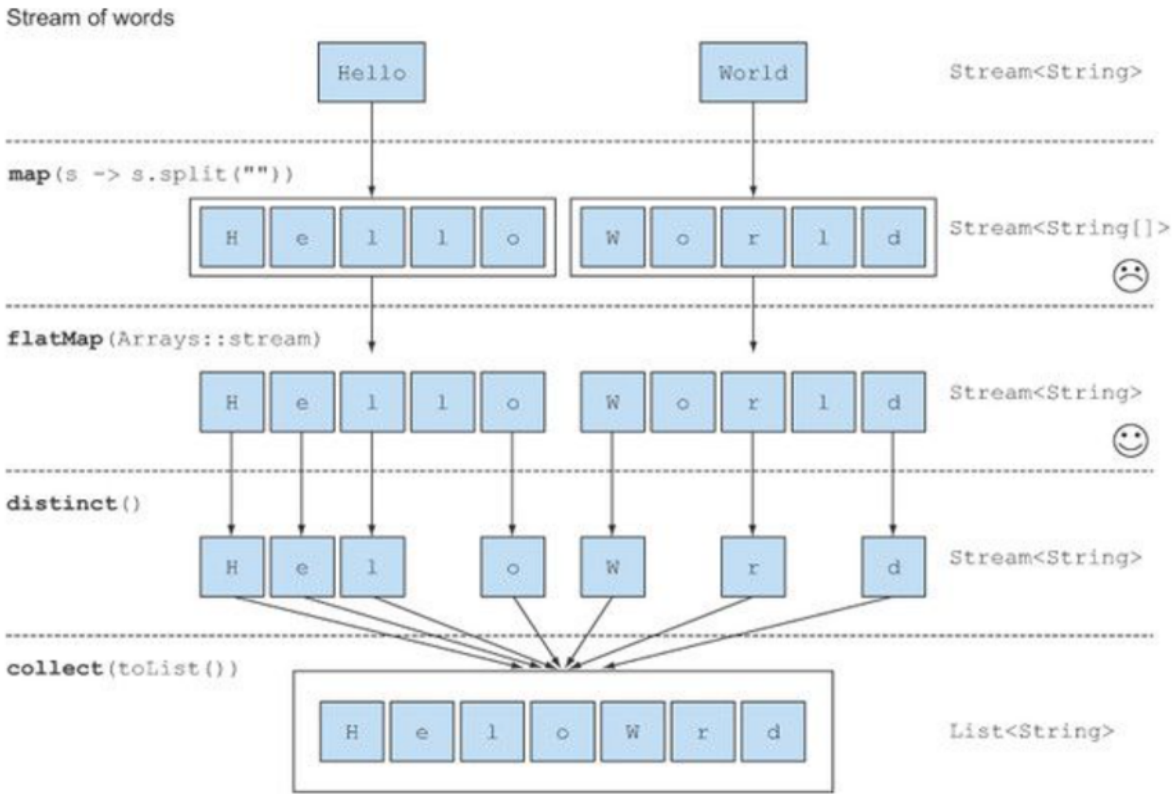
```
        //[6, 7, 8]
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        List<List<Integer>> listOfLists = Arrays.asList(
                Arrays.asList(1, 2, 3),
                Arrays.asList(4, 5),
                Arrays.asList(6, 7, 8)
        );

        listOfLists.stream()    //Stream<List<Integer>>
                //Convert each list into a stream and flatten
                .flatMap( list-> list.stream() )
                .forEach( System.out::print);
        //12345678
    }
}
```

- How could we return a list of all the unique characters for a list of words?
    - For example, given the list of words ["Hello", "World"] we'd like to return the list ["H", "e", "l", "o", "W", "r", "d"].

```
import java.util.Arrays;
public class Program {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("Hello", "World");
        words.stream()  //Stream<String>
                .map( s -> s.split(""))   //Stream<String[]>
                //.flatMap( arr -> Arrays.stream( arr ) )//or
                .flatMap( Arrays::stream )
                .distinct()
                .forEach(System.out::print);
        //HeloWrd
    }
}
```

- Using the flatMap method has the effect of mapping each array not with a stream but with the contents of that stream. All the separate streams that were generated when using map(Arrays::stream) get into a single stream.
- The flatMap method lets us replace each value of a stream with another stream and then concatenates all the generated streams into a single stream.



- Given a list of numbers, how would you return a list of the square of each number? For example, given [1, 2, 3, 4, 5] you should return [1, 4, 9, 16, 25].

```
import java.util.Arrays;
import static java.util.stream.Collectors.toList;
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        List<Integer> squares = numbers.stream()
                //.map( number -> number * number )//or
                .map(number -> (int) Math.pow(number, 2))
                .collect(toList());

        System.out.println(squares);    //[1, 4, 9, 16, 25]
    }
}
```

- Given two lists of numbers, how would you return all pairs of numbers? For example, given a list [1, 2, 3] and a list [3, 4] you should return [(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]. For simplicity, you can represent a pair as an array with two elements.

```
List<Integer> numberList1 = Arrays.asList(1, 2, 3);
List<Integer> numberList2 = Arrays.asList(3, 4);

List<Stream<int[]>> pairs = numberList1.stream()
        .map(i -> numberList2.stream().map(j -> new int[]{i, j}))
        .collect(toList());
```

- Here we are getting List<Stream<int[]>> pairs but we want List<int[]> pairs hence instead of map we need to use flatMap.

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numberList1 = Arrays.asList(1, 2, 3);
        List<Integer> numberList2 = Arrays.asList(3, 4);

        List<int[]> pairs = numberList1.stream()
                .flatMap(i -> numberList2.stream().map(j -> new int[]{i, j}))
                .collect(toList());

        pairs.forEach( arr -> System.out.println(Arrays.toString(arr)));
        /*
        [1, 3]
        [1, 4]
        [2, 3]
        [2, 4]
        [3, 3]
        [3, 4]*/
    }
}
```

- How would you extend the previous example to return only pairs whose sum is divisible by 3? For example, (2, 4) and (3, 3) are valid.

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numberList1 = Arrays.asList(1, 2, 3);
        List<Integer> numberList2 = Arrays.asList(3, 4);

        List<int[]> pairs = numberList1.stream()
                .flatMap(i -> numberList2.stream()
                        .filter( j -> (i + j ) % 3 == 0 ) //Look Here
                        .map(j -> new int[]{i, j}))
                .collect(toList());

        pairs.forEach( arr -> System.out.println(Arrays.toString(arr)));
        /*
        [2, 4]
        [3, 3]*/
    }
}
```

- Key Differences Between map and flatMap:
  - map: The map function transforms each element into another object (e.g., a list of elements).
  - flatMap: The flatMap function transforms each element into a stream and then flattens these streams into a single stream.

Finding and matching

- In data processing, it's often necessary to check whether elements in a dataset meet a specific condition or property. The Java Streams API provides several methods that allow us to perform such checks efficiently, without the need for complex loops.
- These methods are:
    - allMatch
    - anyMatch
    - noneMatch
    - findFirst
    - findAny
- Consider menu declared in Dish class:

```
public static final List<Dish> menu =
    Arrays.asList( new Dish("pork", false, 800, Dish.Type.MEAT),

            new Dish("beef", false, 700, Dish.Type.MEAT),

            new Dish("chicken", false, 400, Dish.Type.MEAT),

            new Dish("french fries", true, 530, Dish.Type.OTHER),

            new Dish("rice", true, 350, Dish.Type.OTHER),

            new Dish("season fruit", true, 120, Dish.Type.OTHER),

            new Dish("pizza", true, 550, Dish.Type.OTHER),

            new Dish("prawns", false, 400, Dish.Type.FISH),

            new Dish("salmon", false, 450, Dish.Type.FISH));
```

- **allMatch**: This method checks if all elements in the stream satisfy a given condition.
    - Example: Are all numbers greater than zero?
    - Method Signature: boolean allMatch(Predicate<? super T> predicate)
- Find out whether the menu is healthy (that is, all dishes are below 1000 calories)?

```
public class Program {
    public static void main(String[] args) {
        boolean isHealthy = menu.stream()
            .allMatch( dish -> dish.getCalories() < 1000 );

        System.out.println("Is Healthy :: "+isHealthy);
    }
}
```

- **anyMatch**: This method checks if any element in the stream satisfies the condition.
    - Example: Is there at least one negative number in the dataset?

- Method Signature: boolean anyMatch(Predicate<? super T> predicate)

- Find out whether the menu has a vegetarian option?

```
public class Program {
    public static void main(String[] args) {
        boolean isVegetarian = menu.stream()
                        //.anyMatch( dish -> dish.isVegetarian() );//or
                        .anyMatch( Dish::isVegetarian );

        System.out.println("Is Vegetarian :: "+isVegetarian);
    }
}
```

- **noneMatch**: This method checks if no elements in the stream satisfy the condition.
  - It is opposite of allMatch
  - Example: Are there no numbers above 100 in the list?
  - Method Signature: boolean noneMatch(Predicate<? super T> predicate)

- Find out whether the menu is healthy (that is, all dishes are below 1000 calories)?

```
public class Program {
    public static void main(String[] args) {
        boolean isHealthy = menu.stream()
                    //.allMatch( dish -> dish.getCalories() < 1000 );//or
                    .noneMatch( dish -> dish.getCalories() >= 1000 );

        System.out.println("Is Healthy :: "+isHealthy);
    }
}
```

- **findFirst**: This method returns the first element in the stream that satisfies the condition, or an empty result if none match.
  - Example: What is the first number greater than 50?
  - Method Signature: Optional findFirst()

- Given a list of numbers, finds the first square that's divisible by 3

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        Optional<Integer> result = numbers.stream()
                .map(number -> number * number)
                .filter(number -> number % 3 == 0)
                .findFirst();

        result.ifPresent(System.out::println);//9
    }
}
```

- Dont worry about Optional class for now.

- **findAny**: This method returns any element in the stream that satisfies the condition, without guaranteeing it's the first one. It is often used in parallel processing to optimize performance.
  - Example: "Give me any number less than 0."
  - Method Signature: Optional findAny()

- Find a dish that's vegetarian.

```
public class Program {
    public static void main(String[] args) {
        Optional<Dish> result = menu.stream()
                .filter(Dish::isVegetarian)
                .findAny();

        result.ifPresent(System.out::println);
    }
}
```

Optional as a better aleternative to null

- null is a special value in Java that represents the absence of an object or nothing. When a reference variable is set to null, it means that it does not point to any object/instance.
- Tony Hoare, a British computer scientist, who introduced the idea of null references in 1965 while working on a programming language called **ALGOL W**. He said he did this because it was "easy to implement". But over time, he regretted this decision, calling it his "billion-dollar mistake." This is because null causes a lot of problems in programming, like bugs and errors.

```
String str = null;
int length = str.length(); //NullPointerException
```

- Over the last 50 years, millions of developers have spent a lot of time fixing bugs caused by null references. Even though many modern programming languages (including Java) still use null for simplicity, it creates a lot of bugs.
- **How do we model the absence of a value?**
  - Imagine we have the following nested object structure for a person owning a car and having car insurance.

```
class Insurance {
    private String name;    //Default value is null
    public String getName() {
        return this.name;
    }
}
```

```
class Car {
    private Insurance insurance;    //Default value is null
```

```
    public Insurance getInsurance() {
        return this.insurance;
    }
}
```

```
class Person {
    private Car car;    //Default value is null
    public Car getCar() {
        return this.car;
    }
}
```

```
public class Program {
    public static String getCarInsuranceName(Person person) {
        return person.getCar().getInsurance().getName();
    }

    public static void main(String[] args) {
        Person person = new Person();
        String carInsuranceName = Program.getCarInsuranceName( person );
//NullPointerException
        //return value of Person.getCar() is null
        System.out.println("Car Insurance Name:: "+carInsuranceName);
    }
}
```

- In this code, if a person doesn't own a car, the method getCar() might return null to indicate that there's no car. However, if getCar() returns null, calling getInsurance() on that null value will cause a NullPointerException. This will stop our program from running correctly.
- Things can get even worse:
  - What if the person itself is null?
  - What if getInsurance() also returns null?
  - In these cases, we'll run into more errors, making the code unstable.

- **Reducing NullPointerException with defensive checking**
  - First attempt: Using deeply nested if blocks

```
public class Program {
    public static String getCarInsuranceName(Person person) {
        if (person != null) {
            Car car = person.getCar();
            if( car != null ){
                Insurance insurance = car.getInsurance();
                if( insurance != null ){
                    return insurance.getName();
                }
            }
        }
        return "Unknown";
    }
```

```
    public static void main(String[] args) {
        Person person = new Person();
        String carInsuranceName = Program.getCarInsuranceName( person ); //OK
        System.out.println("Car Insurance Name:: "+carInsuranceName);//Unknown
    }
}
```

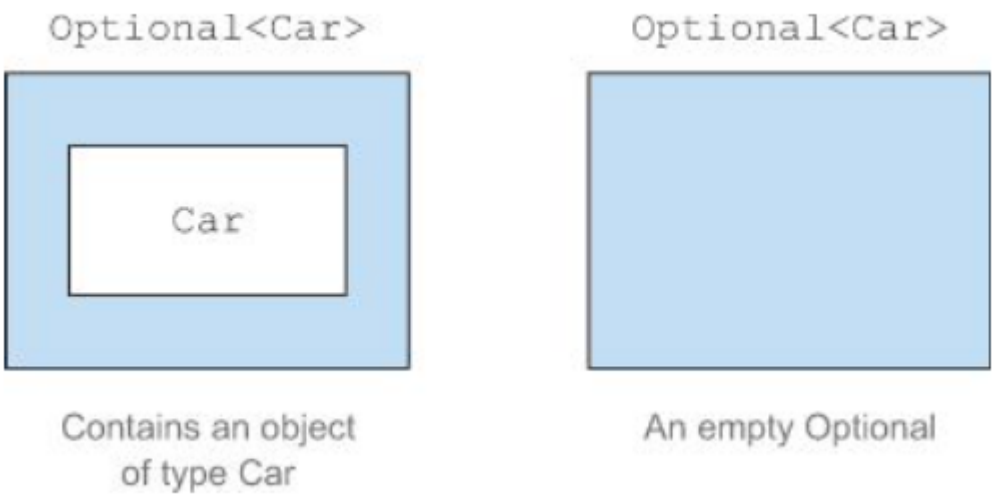
- Second attempt: Without deeply nested if blocks

```
public class Program {
    public static String getCarInsuranceName(Person person) {
        if (person == null) {
            return "Unknown";
        }
        Car car = person.getCar();
        if( car == null ) {
            return "Unknown";
        }
        Insurance insurance = car.getInsurance();
        if( insurance == null ) {
            return "Unknown";
        }
        return insurance.getName();
    }

    public static void main(String[] args) {
        Person person = new Person();
        String carInsuranceName = Program.getCarInsuranceName( person ); //OK
        System.out.println("Car Insurance Name:: "+carInsuranceName);//Unknown
    }
}
```

- What if we forget to check that one property could be null? What we need is a better way to model the absence and presence of a value.

- **Optional class**
  - Java 8 introduces a new class called java.util.Optional. It is a container class to represent the existance or absence of a value.
  - For example, if we know a person might or might not have a car then person declaration should be

```
import java.util.Optional;
public class Person {
    private Optional<Car> car;
    //TODO
}
```



- Methods of java.util.Optional class
  - public static Optional empty();
  - public static Optional of(T);
  - public static Optional ofNullable(T);
  - public T get();
  - public boolean isPresent();
  - public boolean isEmpty();
  - public void ifPresent(Consumer<? super T>);
  - public void ifPresentOrElse(Consumer<? super T>, Runnable);
  - public Optional filter(Predicate<? super T>);
  - public <U> Optional<U> map(Function<? super T, ? extends U>);
  - public <U> Optional<U> flatMap(Function<? super T, ? extends Optional<? extends U>>);
  - public Optional or(Supplier<? extends Optional<? extends T>>);
  - public Stream stream();
  - public T orElse(T);
  - public T orElseGet(Supplier<? extends T>);
  - public T orElseThrow();
  - public T orElseThrow(Supplier<? extends X>) throws X;
- How to create Optional objects?:
  - We can get hold of an empty optional object using the static factory method Optional.empty():

```
public static void main(String[] args) {
    Optional<Car> optionalCar = Optional.empty();
}
```

  - We can also create an optional from a non-null value with the static factory method Optional.of():

```
public static void main(String[] args) {
    Car car = null;
    //Optional<Car> optionalCar = Optional.of( car );  //NullPointerException
```

```
    car = new Car( );
    Optional<Car> optionalCar = Optional.of( car ); //OK
}
```

- Finally, by using the static factory method Optional.ofNullable, we can create an Optional object that may hold a null value:

```
public static void main(String[] args) {
    Car car = null;
    //Optional<Car> optionalCar = Optional.ofNullable( car );  //OK
    //Optional object would be empty

    car = new Car();
    Optional<Car> optionalCar = Optional.ofNullable( car );  //OK
}
```

- Extracting and transforming values from optionals with map

```
class Insurance {
    private String name;
    public String getName() {
        return this.name;
    }
}
```

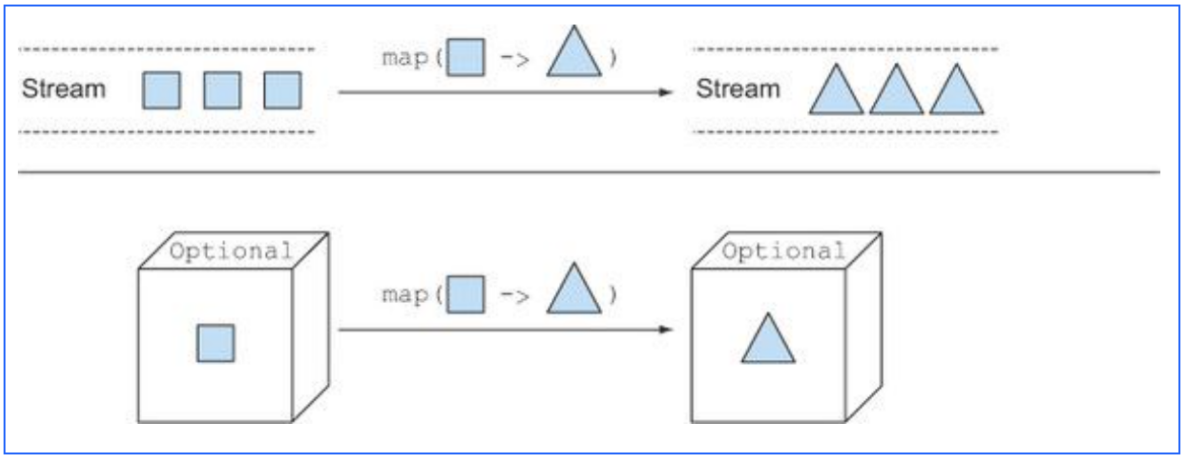
```
class Car {
    private Optional<Insurance> insurance;
    public Optional<Insurance> getInsurance() {
        return this.insurance;
    }
}
```

```
class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() {
        return this.car;
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        Insurance insurance = new Insurance( );
        Optional<Insurance> optionalInsurance = Optional.ofNullable(insurance);
        Optional<String> optionalName =
optionalInsurance.map(Insurance::getName);
        optionalName.ifPresent(System.out::println);
```

```
    }
}
```

- It's conceptually similar to the stream's map method.The map operation applies the provided function to each element of a stream.We could also think of an Optional object as a particular collection of data, containing at most a single element.If the Optional contains a value, then the function passed as argument to map transforms that value. If the Optional is empty, then nothing happens.
- Comparing the map method of Stream and Optional



- Chaining Optional objects with flatMap:
  - Let us imagine we have an Optional and we want to safely get the name of the insurance of the car owned by the person.

```
public class Program {
    public static void main(String[] args) {
        Person person = new Person();
            Optional<Person> optionalPerson = Optional.of(person);
            Optional<String> carInsuranceName =
optionalPerson.map(Person::getCar)
                    .map(Car::getInsurance)//getCar returns
Optional<Optional<Car>>
                    .map(Insurance::getName);
    }
}
```

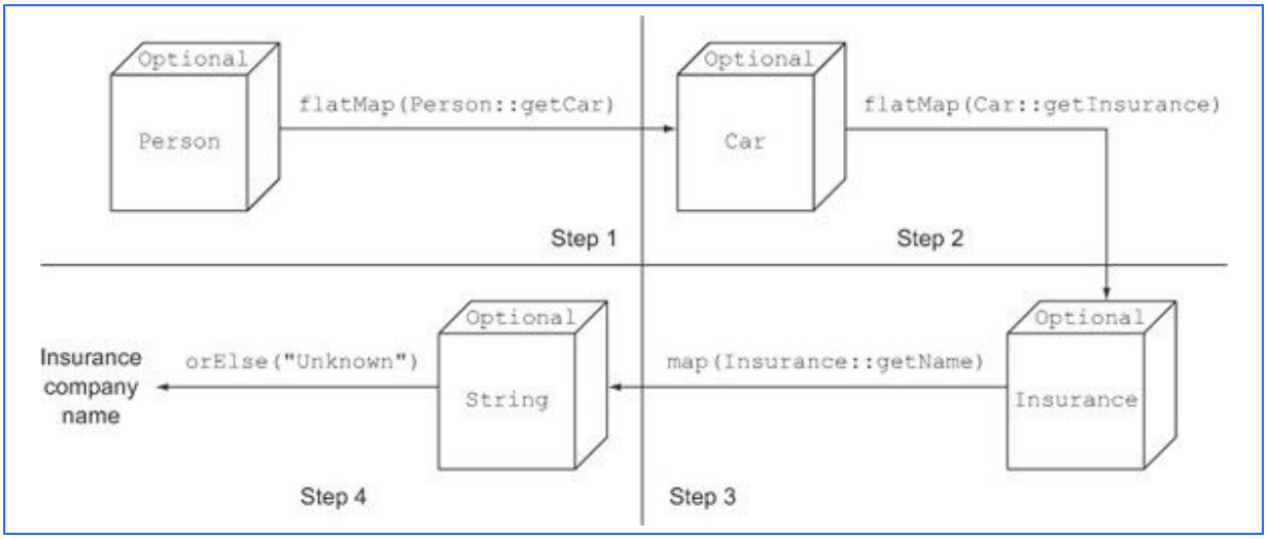
- The problem is with the getCar() method of the Person class. It doesn't return a simple Car object, it returns an Optional. So when we use map(Person::getCar), it actually gives you an Optional<Optional> instead of a plain Optional.
- To fix this, we need to flatten the nested Optional. We can do this using flatMap(), which is similar to map().

```
public class Program {

    public static void main(String[] args) {
        Person person = new Person();
        Optional<Person> optionalPerson = Optional.of(person);
        Optional<String> carInsuranceName =
```

```
optionalPerson.flatMap(Person::getCar)
                .flatMap(Car::getInsurance)
                .map(Insurance::getName);
    }
}
```

- Finding a car's insurance company name with optionals

```
public class Program {
    private static String getCarInsuranceName( Optional<Person> optionalPerson){
        return optionalPerson.flatMap(Person::getCar)
                .flatMap(Car::getInsurance)
                .map(Insurance::getName)
                .orElse("Unknown");
    }
    public static void main(String[] args) {
        Person person = new Person();
        Optional<Person> optionalPerson = Optional.of(person);
        String carInsuranceName = getCarInsuranceName( optionalPerson );
        System.out.println("Car Insurance Name:: "+carInsuranceName);
    }
}
```



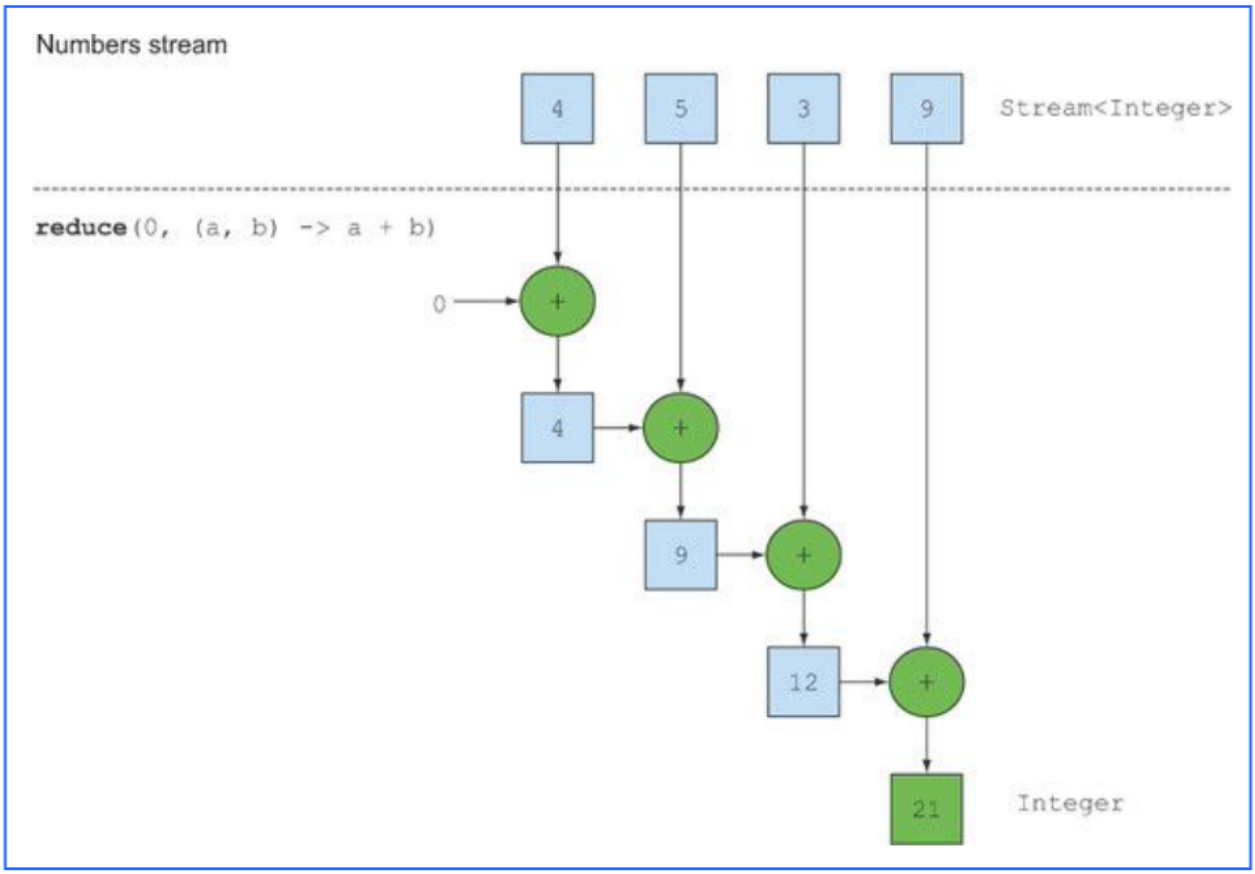
Reducing

- In Java Streams, reducing is an operation that combines the elements of a stream into a single result by repeatedly applying a binary operator. Essentially, it takes a sequence of elements and combines them into a single summary value, such as a sum, average, maximum, minimum, or concatenation.
    - Optional reduce(BinaryOperator accumulator) //BinaryOperator is Functional I/F
    - T reduce(T identity, BinaryOperator accumulator)
    - <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
    - **Summing the elements**
        - First see how we'd sum the elements of a list of numbers using a for-each loop

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4,5,3,9);
        int sum = 0;
        for( int number : numbers )
            sum += number;
        System.out.println("Sum:: "+sum); //21
    }
}
```

- We can sum all the elements of a stream as follows:

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4,5,3,9);

        BinaryOperator<Integer> integerBinaryOperator = (num1, num2) -> num1 +
num2;
        int sum = numbers.stream() .reduce( 0, integerBinaryOperator );
        System.out.println("Sum:: "+sum);   //21

        sum = numbers.stream() .reduce( 0, (num1, num2) -> num1 + num2 );
        System.out.println("Sum:: "+sum);   //21

        sum = numbers.stream() .reduce( 0, Integer::sum );
        System.out.println("Sum:: "+sum);   //21

    }
}
```

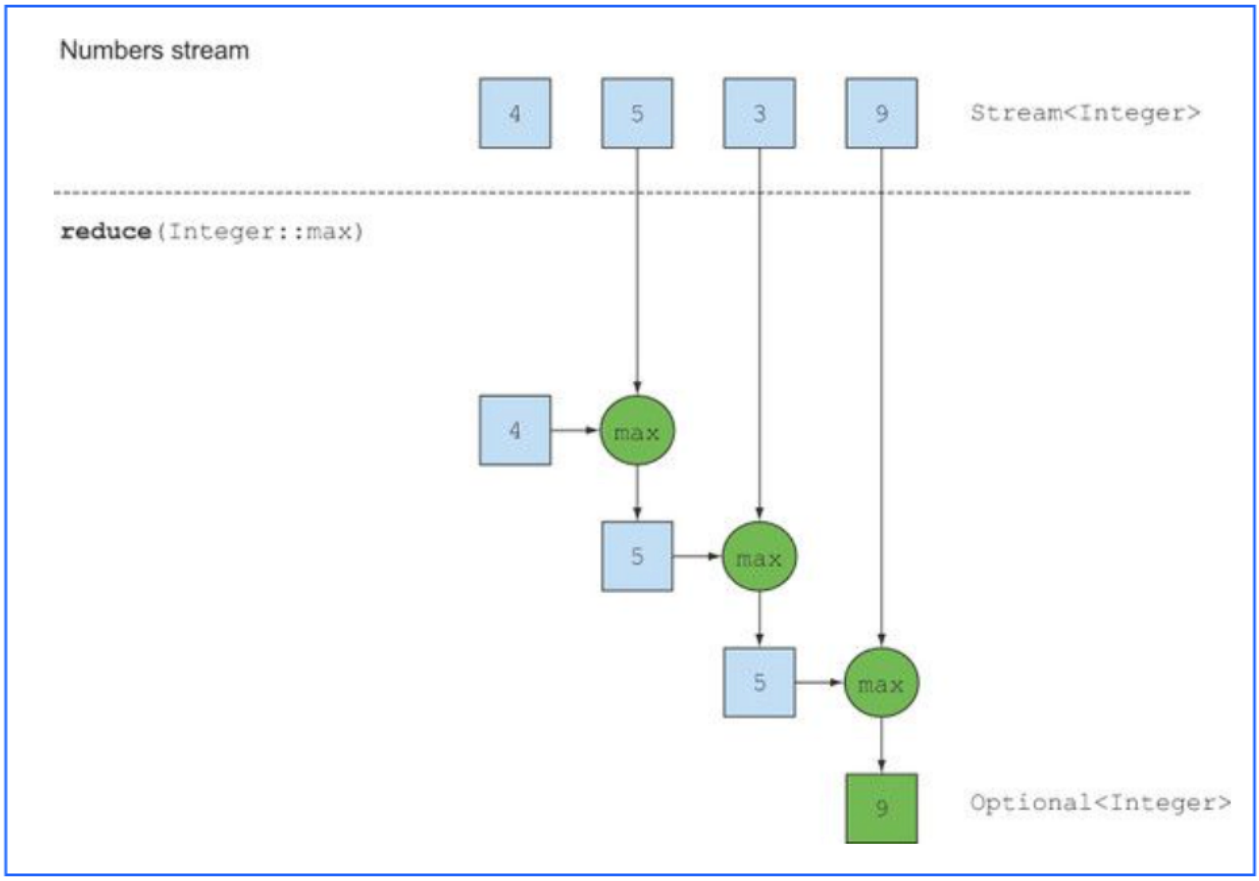


- Here's how it works step by step:
  - Initial Value: We start with the value 0 as the starting point. This is the first parameter of the operation.
  - First Step: The first number in the stream is 4. So, we add 0 + 4, which gives us 4. Now, 4 becomes the new accumulated/current total value.
  - Second Step: Next, we take the accumulated value (4) and add the next number from the stream, which is 5. So, 4 + 5 = 9. Now, 9 is the new accumulated value.
  - Third Step: We then take the accumulated value (9) and add the next number from the stream, which is 3. So, 9 + 3 = 12. Now, 12 is the accumulated value.
  - Final Step: Finally, we take the accumulated value (12) and add the last number in the stream, which is 9. So, 12 + 9 = 21.
- Using overloaded reduce method(No initial value):

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4,5,3,9);

        Optional<Integer> optionalSum = numbers.stream().reduce(Integer::sum);

        optionalSum.ifPresentOrElse( sum -> System.out.println("Sum    :"+sum),
                                     () -> System.out.println("List is empty"));
    }
}
```

- **Maximum and minimum**
  - The reduce operation can also be used to find the maximum or minimum value in a stream, just like it can be used to sum numbers.

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4,5,3,9);

        //Optional<Integer> max = numbers.stream().reduce((num1, num2) ->
Math.max( num1, num2 ) );

        //Optional<Integer> max = numbers.stream().reduce(Math::max );

        Optional<Integer> max = numbers.stream().reduce(Integer::max );

        max.ifPresentOrElse( number -> System.out.println("Max Number is
:"+number),
                                    () -> System.out.println("List is empty"));
    }
}
```



- To calculate the minimum, we need to pass Integer.min to the reduce operation instead of Integer.max:

```
public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4,5,3,9);

        //Optional<Integer> min = numbers.stream().reduce((num1, num2) ->
Math.min( num1, num2 ) );

        //Optional<Integer> min = numbers.stream().reduce(Math::min );
```

```
        Optional<Integer> min = numbers.stream().reduce(Integer::min );

        min.ifPresentOrElse( number -> System.out.println("Min Number is
:"+number),
                                    () -> System.out.println("List is empty"));
    }
}
```

- A chain of map and reduce is commonly known as the map-reduce pattern.
  - How would you count the number of dishes in a stream using the map and reduce methods?
    - We can solve this problem by mapping each element of a stream into the number 1 and then summing them using reduce! This is equivalent to counting in order the number of elements in the stream.

```
public class Program {
    public static void main(String[] args) {
        //long count = menu.stream().count();

        long count = menu.stream()
                .map( dish -> 1 )
                .reduce( 0, Integer::sum);

        System.out.println("Dish Count:: "+ count);
    }
}
```

Putting it all into practice

```
class Trader{
    private String name;
    private String city;

    public Trader(String n, String c){
        this.name = n;
        this.city = c;
    }

    public String getName(){
        return this.name;
    }

    public String getCity(){
        return this.city;
    }

    public void setCity(String newCity){
        this.city = newCity;
    }

    public String toString(){
        return "Trader:"+this.name + " in " + this.city;
```

```
    }
}
```

```
class Transaction{
    private Trader trader;
    private int year;
    private int value;

    public Transaction(Trader trader, int year, int value){
        this.trader = trader;
        this.year = year;
        this.value = value;
    }

    public Trader getTrader(){
        return this.trader;
    }

    public int getYear(){
        return this.year;
    }

    public int getValue(){
        return this.value;
    }

    public String toString(){
        return "{" + this.trader + ", " +"year: "+this.year+", " +"value:" + this.value
+"}";
    }
}
```

```
class Program{
    private static List<Transaction> getTransactions(){
        Trader raoul = new Trader("Raoul", "Cambridge");
        Trader mario = new Trader("Mario","Milan");
        Trader alan = new Trader("Alan","Cambridge");
        Trader brian = new Trader("Brian","Cambridge");

        List<Transaction> transactions = Arrays.asList(
            new Transaction(brian, 2011, 300),
            new Transaction(raoul, 2012, 1000),
            new Transaction(raoul, 2011, 400),
            new Transaction(mario, 2012, 710),
            new Transaction(mario, 2012, 700),
            new Transaction(alan, 2012, 950)
        );
        return transactions;
    }
}
```

- Find all transactions in the year 2011 and sort them by value (small to high).

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
class Program{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();
        List<Transaction> transaction2011 = transactions.stream()
                .filter(transaction -> transaction.getYear() == 2011)
                .sorted(comparing(Transaction::getValue))
                .collect(toList());

        transaction2011.stream().forEach(System.out::println);
    }
}
```

- What are all the unique cities where the traders work?

```
import static java.util.stream.Collectors.toList;
class Program{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();

        List<String> cities = transactions.stream()
                .map(Transaction::getTrader)
                .map(Trader::getCity)
                .distinct()
                .collect(toList());

        cities.stream().forEach(System.out::println);
    }
}
```

- Find all traders from Cambridge and sort them by name.

```
class Program{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();

        List<Trader> traders = transactions.stream()
                .map(Transaction::getTrader)
                .filter(trader -> trader.getCity().equals("Cambridge"))
                .distinct()
                .sorted(comparing(Trader::getName))
                .collect(toList());

        traders.forEach(System.out::println);
    }
}
```

- Return a string of all trader's names sorted alphabetically.

```
class Program{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();

        String traderNames = transactions.stream()
                .map(Transaction::getTrader)
                .map(Trader::getName)
                .distinct()
                .sorted()
                .reduce("", (s1, s2) -> s1.concat(s2));

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

- Are any traders based in Milan?

```
class Program{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();

        boolean milanBased = transactions.stream()
                .map(Transaction::getTrader)
                .anyMatch( trader -> trader.getCity().equals("Milan"));

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

- Print all transactions values from the traders living in Cambridge?

```
class Proagram{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();

        transactions.stream()
            .filter( transaction ->
transaction.getTrader().getCity().equals("Cambridge"))
            .map( Transaction::getValue)
            .forEach(System.out::println);
    }
}
```

- What's the highest value of all the transactions?

```
class Program{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();
```

```
        Optional<Integer> highestTransactionsValue = transactions.stream()
                .map(Transaction::getValue)
                .reduce(Math::max);

        highestTransactionsValue.ifPresentOrElse(System.out::println, ()->
System.out.println("Data not found."));
    }
}
```

- Find the transaction with the smallest value.

```
class Proagram{
    public static void main(String[] args) {
        List<Transaction> transactions = Program.getTransactions();

        Optional<Transaction> smallestTransactions = transactions.stream()
            .reduce((t1, t2) -> t1.getValue() < t2.getValue() ? t1 : t2);

        //Optional<Transaction> smallestTransactions =
transactions.stream().min(comparing(Transaction::getValue));

        smallestTransactions.ifPresent(System.out::println);
    }
}
```

Numeric Streams