

OBJECT-ORIENTED PROGRAMMING USING JAVA

JAVA FUNCTIONAL INTERFACES

QUAN THAI HA

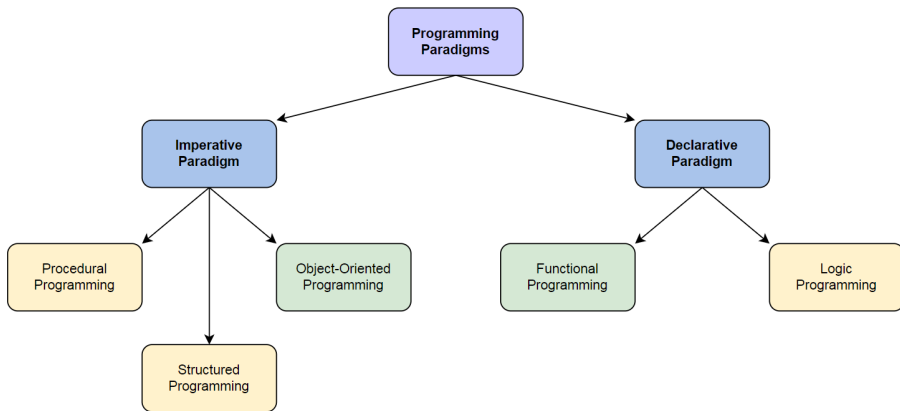
2022



- 1** Java Functional Programming
- 2 Anonymous Class
- 3 Lambda Expressions
- 4 Use Standard Functional Interfaces
- 5 Java Standard Functional Interfaces

- Programming paradigms are approaches used to categorize or classify programming languages based on techniques and features they support.

Types of programming paradigms



Broadly speaking, programming styles can be categorized into **imperative** and **declarative** programming paradigms.

- **Imperative programming is programming how to do something.** It defines a program as a sequence of statements that change the program's state until it reaches the final state.
- **Procedural programming** is a type of imperative programming where we construct programs using procedures or subroutines. One of the popular programming paradigms known as **Object-Oriented Programming (OOP)** extends procedural programming concepts.
- In contrast, **The declarative approach's focus is to define what the program has to achieve rather than how it should achieve it.** **Functional programming** is a subset of the declarative programming languages, it expresses the logic of a computation without describing its control flow in terms of a sequence of statements.

- Traditional programming languages, such as C, FORTRAN, Pascal, are based on procedural programming.
- This programming paradigm uses procedure calls, where each procedure (e.g. function or subroutine) is a set of computational steps to follow.
- It is accessible and easy to understand, but when code gets longer or more complex, making changes to one function can cause a cascade of bugs that can be a headache to trace back.
- To avoid this spaghetti code, computer architects formulated two programming paradigms: **object-oriented programming (OOP)** and **functional programming**.

- Object-oriented programming (OOP) is an imperative paradigm. This means that the code describes how the program should achieve a result in a step-by-step process. These statements procedurally change the program's state.
- Object-oriented programming groups related functions and their variables into objects. In an object, functions are referred to as methods and variables are referred to as properties.
- The four pillars of object-oriented programming are **encapsulation**, **abstraction**, **inheritance**, and **polymorphism**. It is important to recognize these tenets when studying object-oriented programming as they are all interconnected.
- Each pillar of OOP is important in its own right, but the pillars are dependent on one another. Encapsulation is a necessary feature for abstraction and inheritance to be possible. Additionally, polymorphism cannot exist without inheritance. Each of the four pillars is integral to the paradigm. OOP as a whole functions through objects with self-contained properties and methods, and their relationships with other objects.

- Basically, functional programming is a style of writing computer programs that treat computations as evaluating mathematical functions. In mathematics, a function is an expression that relates an input set to an output set. Importantly, the output of a function depends only on its input. More interestingly, we can compose two or more functions together to get a new function.
- **Functional programming is centered around building software composed of functions,** similar to procedural programming.
- In functional programming, functions are treated as primitives. Primitives serve as the simplest elements of a programming language. Thus, a function can be:
 - ▶ stored in a variable
 - ▶ passed as an argument
 - ▶ returned from a function
- Functional programming is a declarative programming model that communicates what the program does without specifying the flow of control. It uses pure functions, immutable variables, and tail-call recursion.

- In the 1930s, mathematician Alonzo Church developed a formal system to express computations based on function abstraction. This universal model of computation came to be known as **lambda calculus**.
- **Lambda calculus had a tremendous impact on developing the theory of programming languages, particularly functional programming languages.** Typically, functional programming languages implement lambda calculus.
- Since lambda calculus focuses on function composition, functional programming languages provide expressive ways to compose software in function composition, be known as **Lambda Expression**.
- Lambda Expression and functional interface in Java is introduced so that functional programming can be more easily adopted in Java. Using Lambda expression, we can mimic the functional programming syntax of assigning a method to a variable or passing it to a parameter.

Object-Oriented Programming

- OOP is so popular because it allows keeping things secure from unwanted external use. It hides variables within the class and thus prevents outside access. Besides that, OOP allows for modularity (possibility to separate the functionality of a program into independent modules) and the management of shared states.
- Objects can be easily reused in another application. It is easy to create new objects for the same class and it is easy to maintain and alter the code.
- OOP has memory management. **It offers a great benefit when it comes to creating large programs** since it allows for easily dividing things into smaller parts and helps to distinguish the components that need to be executed in a certain way.

Functional Programming

- FP offers advantages like lazy evaluation, bug-free code, nested functions, parallel programming. It uses a more declarative approach that concentrates on what needs to be done and less on how to do it, with an emphasis on **efficiency and optimization**.
- In functional programming, it's much easier to know what changes were made since the object itself is now a new object with a different name. It's effective when you have a fixed set of operations, and as your code evolves, you add new operations on existing things.
- In functional programming, it's easier to simulate real-world processes than objects. Its mathematical origins make it suitable for cases that require calculations or anything that includes transformation and processing. OOP in such cases will be inefficient.

- Functional programming and OOP are two of the most popular and followed (even if just partly) programming paradigms. Despite having different approaches, both were designed to help developers create efficient and top-quality applications. These paradigms simply have different ways - and by "ways," we mean strategies, principles, and rules - to get there.
- In OOP, data is stored in objects since the data and respective behaviors (meaning, what a program can do to or with data) should belong to a single location.
- In functional programming, data is passed and collected by functions. However, it does not store data in objects because that could compromise clarity, considering that, for functional programming, data and behavior are different.
- Both programming paradigms have their pros and cons, which is why many developers actually prefer to implement hybrid solutions according to each project's requirements and goals.

Now we'll try to understand how programming languages are divided based on their support for functional programming.

- Pure functional languages, such as Haskell, only allow pure functional programs.
- Other languages allow both functional and procedural programs and are considered impure functional languages. Many languages fall into this category, including C++, Scala, Kotlin and Java.
- It's important to understand that most of the popular programming languages today are general-purpose languages, so they tend to support multiple programming paradigms.

- A typical problem in 2000: model a company employees, departments, and salaries!



```
1 public static List<String> getNames(List<Employee> employees) {  
    List<String> nameList = new ArrayList<>();  
3    for (int i = 0; i < employees.size(); i++) {  
        String name = nameList.get(i).getName();  
5        if (name.length() > 5) {  
            nameList.add(name.toUpperCase());  
7        }  
    }  
9    return nameList;  
}
```

We had to manually implement a way to iterate the list (using for loop), check each name's length, capitalize and add it to the new list. In other words, we iterated each element and

- ▶ filtered the elements longer than 5 characters
- ▶ converted the elements to uppercase, and
- ▶ added the elements to a list.

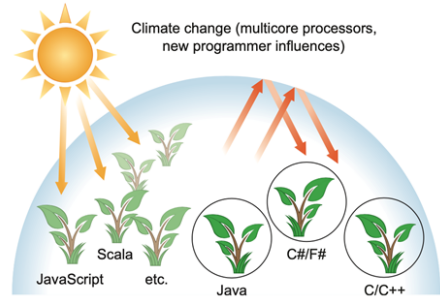
- A typical problem in 2022: analyse a massive 20TB dataset for finding lazy employees!



```
public static List<String> getNames(List<Employee> employees) {  
2   return nameList  
    .stream()  
4   .filter(employee -> employee.getName().length() > 5)  
    .map(employee -> employee.getName().toUpperCase())  
6   .collect(Collectors.toList());  
}
```

- Java provides Stream API as a functional approach to process collections. A stream can iterate itself, so we don't need to implement how to iterate the list. It works like a pipeline that can take in a number of operations.
 - Instead of implementing how to filter, convert, and add the elements in imperative programming, we can now use methods provided by the Stream API such as filter(), map(), and collect() and customize them by passing parameters.

- The rise of big-data, and affordable multi-core architectures made functional programming concepts popular again.
- A number of languages (C++, Java, Python, Scala) are introducing ways for better supporting functional programming.
- Haskell is a recent purely functional language.
- Languages need to evolve to track changing hardware or programmers expectations. Otherwise, they die (COBOL, LISP, ...)



- Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria.
- The members of this social networking application are represented by *Person* class.
- The members of your social networking application are stored in a *List<Person>* instance.



```
1 public class Person {  
    public enum Sex {  
3         MALE, FEMALE  
    }  
5  
    private String name;  
7    private LocalDate birthday;  
    private Sex gender;  
9    private String emailAddress;  
11  
    public int getAge() {  
        /* ... */  
13    }  
15  
    public void printPerson() {  
        /* ... */  
17    }  
}
```

- One simplistic approach is to create several methods; each method searches for members that match one characteristic, such as gender or age. The following method prints members that are older than a specified age:



```
public static void printPersonsOlderThan(  
2    List<Person> roster, int age) {  
    for (Person p : roster) {  
4        if (p.getAge() >= age) {  
            p.printPerson();  
6        }  
    }  
8 }
```


- After a while, requirements change and you need to **print members within a specified range of ages**.
- You can add an alternative `printPersonsWithinAgeRange()` method. Trying to create a separate method for each possible search query can still lead to brittle code. Furthermore, this approach breaks the **DRY (don't repeat yourself)** principle of software engineering. The two methods vary only in one line: **the highlighted condition inside the if construct**.



```
public static void printPersonsWithinAgeRange(  
2     List<Person> roster, int low, int high) {  
    for (Person p : roster) {  
4        if (low <= p.getAge() && p.getAge() < high) {  
            p.printPerson();  
6        }  
    }  
8 }
```

- You can instead separate the code that specifies the criteria for which you want to search in a different class



```
public static void printPersons(  
2    List<Person> roster, CheckPerson tester) {  
    for (Person p : roster) {  
4        if (tester.test(p)) {  
            p.printPerson();  
6        }  
    }  
8 }
```

- This method checks each *Person* instance contained in the *List* parameter *roster* whether it satisfies the search criteria specified in the *CheckPerson* parameter *tester* by invoking the method *tester.test*. If the method *tester.test* returns a *true* value, then the method *printPersons* is invoked on the *Person* instance.

- We can define a *CheckPerson* interface encapsulating the **selection strategy**. Different classes can then implement different strategies.
- This method filters members that are eligible for selective service: it returns a true value if its Person parameter is male and between the ages of 18 and 25:



```
1 interface CheckPerson {  
2     boolean test(Person s);  
3 }
```



```
1 class CheckPersonForSelectiveService  
    implements CheckPerson {  
3     @Override  
    public boolean test(Person p) {  
5         return p.gender == Person.Sex.MALE  
            && p.getAge() >= 18  
7            && p.getAge() <= 25;  
            }  
9 }
```

- This code is much **more flexible** than our first attempt, and at the same time it's easy to read and to use!
- **Behaviour parameterization** is great because it enables you to separate the logic of iterating the collection and the behaviour to apply on each element of that collection.



```
1 public static void printPersons(  
    List<Person> roster, CheckPerson tester) {  
3     for (Person p : roster) {  
        if (tester.test(p)) {  
5         p.printPerson();  
        }  
7     }  
    }  
9  
    /* main */  
11 printPersons(roster, new CheckPersonForSelectiveService());
```

- However, when you want to pass new behaviour to your *printPersons* method, **you're forced to declare several classes** that implement the *CheckPerson* interface and then instantiate the needed objects. It's a **time-consuming and verbose process!**
- **Verbosity in general is bad**; it discourages the use of that language feature because it takes a long time to write and maintain.



```
1 interface CheckPerson {  
    boolean test(Person s);  
3 }  
  
5 class CheckPersonForSelectiveService implements CheckPerson {  
    public boolean test(Person p) {  
7        return p.gender == Person.Sex.MALE  
            && p.getAge() >= 18  
9            && p.getAge() <= 25;  
        }  
11 }
```

- 1 Java Functional Programming
- 2 Anonymous Class**
- 3 Lambda Expressions
- 4 Use Standard Functional Interfaces
- 5 Java Standard Functional Interfaces

- Because *checkPersonForSelectiveService* implements an interface, you can use an **anonymous class** instead of a local class and bypass the need to declare a new class for each search.



```
1 printPersons (  
    roster ,  
3    new CheckPerson() {  
        public boolean test(Person p) {  
5            return p.getGender() == Person.Sex.MALE  
                && p.getAge() >= 18  
7            && p.getAge() <= 25;  
        }  
9    }  
);
```

- This approach **reduces the amount of code** required because you don't have to create a new class for each search that you want to perform.

- Anonymous classes are like local classes except that they do not have a name.
- They enable you to declare and instantiate a class at the same time.
- Use them if you need to use a local class only once. In short, they allow you to create ad hoc implementations.
- Anonymous classes enable you to make your code more concise.

- 1 Java Functional Programming
- 2 Anonymous Class
- 3 Lambda Expressions**
- 4 Use Standard Functional Interfaces
- 5 Java Standard Functional Interfaces

- The *CheckPerson* interface is a interface that contains only one abstract method. Because it contains only one abstract method, you can omit the name of that method when you implement it.
- To do this, instead of using an anonymous class expression, you use a *lambda expression*, which is highlighted in the following method invocation



```

1 printPersons (
2     roster ,
3     (Person p) -> p.getGender() == Person.Sex.MALE
4         && p.getAge() >= 18
5         && p.getAge() <= 25
6 );
    
```

- In Java, an interface that contains only one abstract method is a *functional interface*. (A *functional interface* may contain one or more *default methods* or *static methods*).



```
1 interface CheckPerson {  
2     boolean test(Person s);  
3 }  
4  
5 public static void printPersons(  
6     List<Person> roster, CheckPerson tester) {  
7     for (Person p : roster) {  
8         if (tester.test(p)) {  
9             p.printPerson();  
10        }  
11    }  
12 }  
13  
14 printPersons(  
15     roster,  
16     (Person p) -> p.getGender() == Person.Sex.MALE  
17         && p.getAge() >= 18  
18         && p.getAge() <= 25  
19 );
```

- To know how a function can be legally used, you need to know **its name**, **how many parameters** it requires, **their types**, and the **return type** of the function. A **functional interface** specifies this information about one function.
- A functional interface is similar to a class, and it can be defined in a .java file, just like a class. However, its content is just a specification for a single function



```
1 interface CheckPerson {  
    boolean test(Person s);  
3 }
```

- Java comes with many standard functional interfaces. One of the most important is a very simple one named Runnable, which is already defined in Java as



```
1 public interface Runnable {  
    public void run();  
3 }
```

- A lambda expression can be understood as a concise representation of an anonymous function, that is, one without a name. It doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown. The full syntax is:

(parameter-list) -> { statements }

That's one big definition, let's break it down:

- ▶ **Anonymous** - We say anonymous because it doesn't have an explicit name like a method would normally have.
- ▶ **Function** - We say function because a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- ▶ **Passed around** - A lambda expression can be passed as argument to a method or stored in a variable.
- ▶ **Concise** - You don't need to write a lot of boilerplate like you do for anonymous classes.

- The **parameter-list** can be empty, or it can be a list of parameter declarations, separated by commas, where each declaration consists of a type followed by a parameter name.
- However, the syntax can often be simplified.
 - ▶ The parameter types can be omitted, as long as they can be deduced from the context. For example, if the lambda expression is known to be of type **CheckPerson**, then the parameter type must be **Person**, so it is unnecessary to specify the parameter type in the lambda expression.
 - ▶ If there is exactly one parameter and if its type is not specified, then the parentheses around the parameter list can be omitted.
 - ▶ On the right-hand side of the \rightarrow , if the only thing between the braces, { and }, is a single function call statement, then the braces can be omitted.
 - ▶ If the right-hand side has the form `return expression;`, then you can omit everything except the **expression**.

- Suppose that we want a lambda expression to represent a function that computes the square of a *double* value. The type of such a function can be the *SquareFunction* interface.
- If *square* is a variable of type *SquareFunction*, then the value of the function can be a lambda expression, which can be written in any of the following forms



```
1 // The full lambda expression syntax.  
square = (double x) -> { return x*x; };  
3 // Parameter types can be omitted.  
square = (x) -> { return x*x; };  
5 // The parentheses around the parameter list can be omitted.  
square = x -> { return x*x; };  
7 // The braces can be omitted.  
square = x -> x*x;  
9 // The parameters in a lambda expression are dummy parameters;  
// their names are irrelevant.  
11 square = (double fred) -> fred*fred;  
square = (z) -> z*z;
```



```
// 1. A boolean expression
2 (List<String> list) -> list.isEmpty()

4 // 2. Creating objects
  () -> new Apple(10)

6

8 // 3. Consuming from an object
  (Apple a) -> { System.out.println(a.getWeight()); }

10 // 4. Select/extract a field from an object
   (String s) -> s.length()

12

14 // 5. Multiply two ints
   (int a, int b) -> a * b

16 // 6. Compare two objects
   (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
```


- **Functional interfaces** or interfaces defining only one function (one method) are ideal candidates for making use of **lambda expressions**.
- Instead of using **anonymous classes** (still verbose!), lambda expressions can be used for providing the implementation of their single method!
- Java 8 has defined a lot of functional interfaces in this package `java.util.function`. Some of the useful java 8 functional interfaces are **Consumer**, **Supplier**, **Function** and **Predicate**.

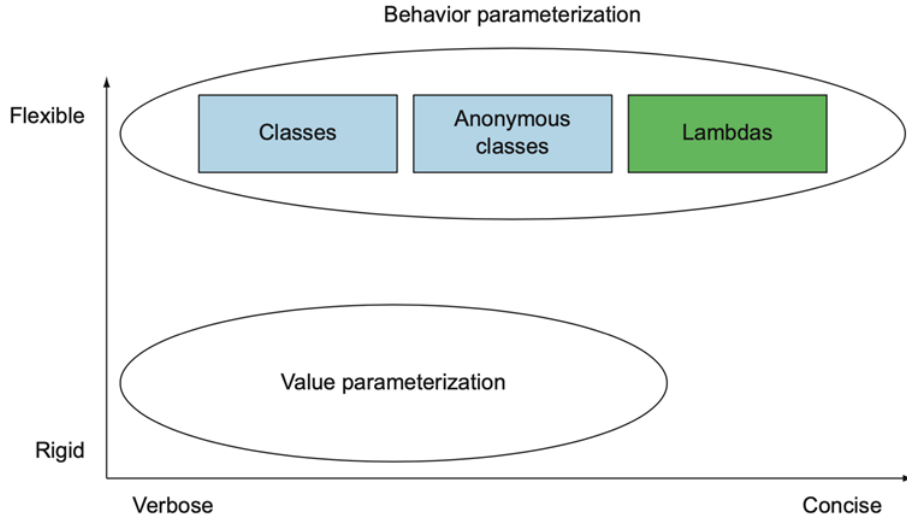


```
1 public interface Comparator<T> {  
    int compare(T o1, T o2);  
3 }  
  
5 interface Predicate<T> {  
    boolean test(T t);  
7 }  
  
9 public interface Consumer<T> {  
    void accept(T t);  
11 }  
  
13 public interface Supplier<T> {  
    T get();  
15 }  
  
17 public interface Function<T, R> {  
    R apply(T t);  
19 }
```



```
1 // Using anonymous classes
  Comparator<Apple> byWeight =
3   new Comparator<Apple>() {
      public int compare(Apple a1, Apple a2) {
5         return a1.getWeight().compareTo(a2.getWeight());
      }
7   };

9
11 // Using lambda expressions
    Comparator<Apple> byWeight =
      (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```



- 1 Java Functional Programming
- 2 Anonymous Class
- 3 Lambda Expressions
- 4 Use Standard Functional Interfaces**
- 5 Java Standard Functional Interfaces

- Reconsider the *CheckPerson* interface. This is a very simple interface. **It's a functional interface because it contains only one abstract method.** The method is so simple that it might not be worth it to define one in your application.



```
1 interface CheckPerson {  
2     boolean test(Person p);  
3 }
```

- Consequently, the JDK defines several standard functional interfaces, which you can find in the package `java.util.function`, such as *Predicate*<*T*>.



```
1 interface Predicate<T> {  
2     boolean test(T t);  
3 }
```

- The interface *Predicate<T>* is an example of a **generic interface**. Generic types (such as generic interfaces) specify one or more type parameters within angle brackets (<>). This interface contains only one type parameter, T.
- When you declare or instantiate a generic type with actual type arguments, you have a **parameterized type**.
- For example, the parameterized type *Predicate<Person>* is the following



```
1 interface Predicate<Person> {  
    boolean test(Person t);  
3 }
```

- You can use *Predicate<T>* in place of *CheckPerson* as the following:



```
1 public static void printPersonsWithPredicate (  
    List<Person> roster , Predicate<Person> tester) {  
3     for (Person p : roster) {  
        if (tester.test(p)) {  
5         p.printPerson();  
        }  
7     }  
9 }  
  
10 printPersonsWithPredicate (  
11     roster ,  
    p -> p.getGender() == Person.Sex.MALE  
13     && p.getAge() >= 18  
    && p.getAge() <= 25  
15 );
```

- Reconsider the method *printPersonsWithPredicate* to see where else you could use lambda expressions:



```
1 public static void printPersonsWithPredicate(  
    List<Person> roster, Predicate<Person> tester) {  
3     for (Person p : roster) {  
        if (tester.test(p)) {  
5         p.printPerson();  
        }  
7     }  
}
```

- This method checks each *Person* instance contained in the *List* parameter *roster* whether it satisfies the criteria specified in the *Predicate* parameter *tester*. If the *Person* instance does satisfy the criteria specified by *tester*, the method *printPerson* is invoked on the *Person* instance.

- Instead of invoking the method *printPerson*, you can specify a different action to perform on those *Person* instances that satisfy the criteria specified by *tester*. You can specify this action with a lambda expression.
- Suppose you want a lambda expression similar to *printPerson*, one that takes one argument (an object of type *Person*) and returns *void*.
- To use a lambda expression, you need to implement a functional interface.
- You need a functional interface that contains an abstract method that can take one argument and returns *void*.
- The *Consumer<T>* interface contains the method *void accept(T t)*, which has these characteristics.



```
public static void processPersons (
2     List<Person> roster ,
    Predicate<Person> tester ,
4     Consumer<Person> block) {
    for (Person p : roster) {
6         if (tester.test(p)) {
            block.accept(p);
8         }
    }
10 }

12 processPersons (
    roster ,
14     p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
16     && p.getAge() <= 25 ,
    p -> p.printPerson()
18 );
```

- What if you want to do more with your members' profiles than printing them out. Suppose that you want to validate the members' profiles or retrieve their contact information?
- You need a functional interface that contains an abstract method that returns a value.
- The `Function<T,R>` interface contains the method `R apply(T t)` which has these characteristics.
- The following method retrieves the data specified by the parameter *mapper*, and then performs an action on it specified by the parameter *block*:



```

1 public static void processWithFunction (
2     List<Person> roster ,
3     Predicate<Person> tester ,
4     Function<Person, String> mapper ,
5     Consumer<String> block) {
6     for (Person p : roster) {
7         if (tester.test(p)) {
8             String data = mapper.apply(p);
9             block.accept(data);
10        }
11    }
12 }

14 processWithFunction (
15     roster ,
16     p -> p.getGender() == Person.Sex.MALE
17         && p.getAge() >= 18
18         && p.getAge() <= 25,
19     p -> p.getEmailAddress(),
20     email -> System.out.println(email)
21 );
    
```

- The *processElements* is a **generic version** of *processWithFunction* that accepts, as a parameter, a collection that contains elements of any data type.
- This method invocation performs the following actions:
 - ▶ Obtains a source of objects from the collection source, *roster*.
 - ▶ Filters objects that match the **Predicate** object *tester*.
 - ▶ Maps each filtered object to a value as specified by the **Function** object *mapper*.
 - ▶ Performs an action on each mapped object as specified by the **Consumer** object *block*.



```
1 public static <X,Y> void processElements(  
    Iterable<X> source ,  
3    Predicate<X> tester ,  
    Function<X,Y> mapper ,  
5    Consumer<Y> block ) {  
    for (X p : source) {  
7        if ( tester.test(p) ) {  
            Y data = mapper.apply(p);  
9            block.accept(data);  
        }  
11    }  
12 }  
13  
14 processElements(  
15     roster ,  
    p -> p.getGender() == Person.Sex.MALE  
17     && p.getAge() >= 18  
    && p.getAge() <= 25 ,  
19     p -> p.getEmailAddress() ,  
    email -> System.out.println(email)  
21 );
```

- You can use **aggregate operations** that accept lambda expressions as parameters



```
1 roster
   .stream()
3   .filter(
       p -> p.getGender() == Person.Sex.MALE
5       && p.getAge() >= 18
       && p.getAge() <= 25)
7   .map(p -> p.getEmailAddress())
   .forEach(email -> System.out.println(email));
```

The operations filter, map, and forEach are **aggregate operations**. **Aggregate operations** process elements from a **stream**, not directly from a collection. A **stream** is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source, such as collection, through a **pipeline**. A **pipeline** is a sequence of stream operations. In addition, aggregate operations typically accept lambda expressions as parameters, enabling you to customize how they behave.

- 1 Java Functional Programming
- 2 Anonymous Class
- 3 Lambda Expressions
- 4 Use Standard Functional Interfaces
- 5 Java Standard Functional Interfaces**

- JDK 8 introduces a new package *java.util.function*, which provides a number of **Standard Functional Interfaces**. Besides declaring one abstract method, these interfaces also heavily use default and static methods (with implementation) to enhance their functionality.
- There are four basic patterns (suppose that *t* is an instance of *T*):

Pattern	Functional Interface	Lambda Expression	Explanation
Predicate	Predicate<T> BiPredicate<T, U>	t -> boolean (t, u) -> boolean	Apply boolean test on the given element t
Function	Function<T, R> BiFunction<T, U, R> UnaryOperator<T> BinaryOperator<T>	t -> r (t, u) -> r t -> t (t, t) -> t	Transform (Map) from t to r
Consumer	Consumer<T> BiConsumer<T, U>	t -> void (t, u) -> void	Consume t with side effect such as printing
Supplier	Supplier<T> BooleanSupplier	() -> t () -> boolean	Supply an instance t

- For greater efficiency, specialized functional interfaces are defined for primitive types `int`, `long` and `double`, as summarized below:

Pattern	Functional Interface	Lambda Expression
Predicate	<code>IntPredicate</code>	<code>int -> boolean</code>
Function	<code>IntFunction<R></code> <code>IntToDoubleFunction</code> <code>IntToLongFunction</code> <code>ToIntFunction<T></code> <code>ToIntBiFunction<T, U></code> <code>IntUnaryOperator</code> <code>IntBinaryOperator</code>	<code>int -> r</code> <code>int -> double</code> <code>int -> long</code> <code>t -> int</code> <code>(t, u) -> int</code> <code>int -> int</code> <code>(int, int) -> int</code>
Consumer	<code>IntConsumer</code> <code>ObjIntConsumer<T></code>	<code>int -> void</code> <code>(t, int) -> void</code>
Supplier	<code>IntSupplier</code>	<code>() -> int</code>

Notes: Same Functional Interfaces as `int` are also defined for primitives `long` and `double`.

- Java **Predicate** in general meaning is a statement about something that is either true or false. In programming, **predicates represent single argument functions that return a boolean value.**
- In Java, *Predicate*<T> is a generic functional interface that represents a single argument function that returns a **boolean** value (**true** or **false**). This interface available in *java.util.function* package and contains a *test(T t)* method that evaluates the predicate of a given argument.



```
@FunctionalInterface
2 public interface Predicate<T> {
    boolean test(T t);
4 }
```

- It is used to filter elements from the **java stream**.

■ How it works?

1. A **Predicate instance** implements **Predicate interface** to provide implementation for the *test()* abstract method.
2. Methods such as *aStream.filter(aPredicate)* takes *aPredicate* as its argument. It then invokes *aPredicate.test(e)* on each of its elements *e*, which returns **true/false**, to decide if the element is to be filtered in/out.
3. **Lambda expression** hides the name of the abstract method and the **Predicate**, e.g., *aStream.filter(p -> p.getAge() >= 21)*. You simply apply the method body to each of the elements *p*.

■ Some of the Stream methods where **Predicate** specializations are used are:

Syntax
<code>Stream<T> filter(Predicate<? super T> predicate)</code>
<code>boolean anyMatch(Predicate<? super T> predicate)</code>
<code>boolean allMatch(Predicate<? super T> predicate)</code>
<code>boolean noneMatch(Predicate<? super T> predicate)</code>



```
import java.util.List;
2 import java.util.function.Predicate;

4 class BiggerThanFive<E> implements Predicate<Integer> {
    @Override
6     public boolean test(Integer v) {
        Integer five = 5;
8         return v > five;
    }
10 }

12 public class PredicateTest {
    public static void main(String[] args) {
14         List<Integer> numbers = List.of(2, 3, 1, 5, 6, 7, 8, 9, 12);
        BiggerThanFive<Integer> biggerThanfive = new BiggerThanFive<>();
16         numbers.stream().filter(biggerThanfive).forEach(System.out::println);
    }
18 }
```

- Java **Function** represents a function that takes one type of argument and returns another type of argument. `Function<T, R>` is the generic form where `T` is the type of the input to the function and `R` is the type of the result of the function.



```
@FunctionalInterface
2 public interface Function<T, R> {
    /**
4     * Applies this function to the given argument.
    *
6     * @param t the function argument
    * @return the function result
8     */
    R apply(T t);
10 }
```

- Some of the Stream methods where Function or it's primitive specialization is used are:

Syntax
<code><R> Stream<R> map(Function<? super T, ? extends R> mapper)</code>
<code>IntStream mapToInt(ToIntFunction<? super T> mapper)</code>
<code>IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper)</code>
<code><A> A[] toArray(IntFunction<A[]> generator)</code>
<code><U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)</code>



```
import java.util.function.Function;

2
class SquareFunction implements Function<Integer, String> {
4
    @Override
6    public String apply(Integer t) {
        return Integer.toString(t*t);
8    }
}

10
public class FunctionTest {
12    public static void main(String[] args) {
        Function<Integer, String> squareFunction = new SquareFunction();
14        System.out.println(squareFunction.apply(2));
    }
16 }
```

- Java **Consumer** is a functional interface which represents an operation that **accepts a single input argument and returns no result.**



```
@FunctionalInterface
2 public interface Consumer<T> {
    /**
4     * Performs this operation on the given argument.
    *
6     * @param t the input argument
    */
8     void accept(T t);
}
```

- Unlike most other functional interfaces, Consumer is expected to operate via side-effects.
- It can be used to perform some action on all the elements of the java stream.

- Some of the Java 8 Stream methods where Consumer specialization interfaces are used are:

Syntax

```
Stream<T> peek(Consumer<? super T> action)
```

```
void forEach(Consumer<? super T> action)
```

```
void forEachOrdered(Consumer<? super T> action)
```



```
1 import java.util.function.Consumer;
3 public class ConsumerTest {
5     public static void main(String[] args) {
6         // Create consumer
7         Consumer<String> consumer = new Consumer<String>() {
8             @Override
9             public void accept(String name) {
10                 System.out.println("Hello, " + name);
11             }
12         };
13
14         // Calling Consumer method
15         consumer.accept("Java"); // Hello, Java
16     }
17 }
```


- Java **Supplier** is a functional interface which represents an operation that returns a result. Supplier does not take any arguments.



```
1 @FunctionalInterface
  public interface Supplier<T> {
3  /**
   * Gets a result.
5  *
   * @return a result
7  */
   T get();
9 }
```

- Some of the methods in Stream that takes Supplier argument are:

Syntax

```
<T> Stream<T> generate(Supplier<T> s)
```

```
<R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)
```







```
1 import java.util.function.Supplier;  
  
3 public class SupplierTest {  
  
5     public static void main(String[] args) {  
  
7         // Create random supplier  
        Supplier<Double> randomSupplier = () -> Math.random();  
  
9  
        System.out.println("Random value: " + randomSupplier.get());  
11       System.out.println("Random value: " + randomSupplier.get());  
        System.out.println("Random value: " + randomSupplier.get());  
13     }  
}
```

- Java **Optional** is a container object which may or may not contain a non-null value. If a value is present, `isPresent()` will return true and `get()` will return the value. Stream terminal operations return Optional object.
- Some of these methods are:

Syntax
<code>Optional<T> reduce(BinaryOperator<T> accumulator)</code>
<code>Optional<T> min(Comparator<? super T> comparator)</code>
<code>Optional<T> max(Comparator<? super T> comparator)</code>
<code>Optional<T> findFirst()</code>
<code>Optional<T> findAny()</code>

- For supporting parallel execution in Java 8 Stream API, [Spliterator](#) interface is used. Spliterator trySplit method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

-  ALLEN B. DOWNEY, CHRIS MAYFIELD, ***THINK JAVA*** (2016).
-  MARTIN FOWLER, ***UML DISTILLED - A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGUAGE*** (2004).
-  RICHARD WARBURTON, ***OBJECT-ORIENTED VS. FUNCTIONAL PROGRAMMING - BRIDGING THE DIVIDE BETWEEN OPPOSING PARADIGMS*** (2016).
-  JAMES GOSLING, BILL JOY, GUY STEELE, GILAD BRACHA, ALEX BUCKLEY, ***THE JAVA LANGUAGE SPECIFICATION - JAVA SE 8 EDITION*** (2015).

THANK YOU!