Lambda expressions basic

Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.
From (https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.htmll)

A lambda expression is an anonymous function. A function that doesn't have a name and doesn't belong to any class. The concept of lambda expression was first introduced in LISP programming language.

Lambda Expressions enable you to encapsulate a single unit of behavior and pass it to other code. You can use a lambda expressions if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
New and Enhanced APIs That Take Advantage of Lambda Expressions and Streams in Java SE 8 describe new and enhanced classes that take advantage of lambda expressions and streams.

Lambda expressions let you express instances of single-method classes more compactly
A functional interface is any interface that contains only one abstract method
Because a functional interface 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:

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

JDK defines several standard functional interfaces, which you can find in the package java.util.function see: https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html:

Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
Function<T, R>	Represents a function that accepts one argument and produces a result.
Predicate<T>	Represents a predicate (boolean-valued function) of one argument.
Supplier<T>	Represents a supplier of results.
UnaryOperator<T>	Represents an operation on a single operand that produces a result of the same type as its operand.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands

Using Lambda examples
→ standard way

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
```

```
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

→ general

```
public static <X, Y> void processElements(
    Iterable<X> source,
    Predicate<X> tester,
    Function <X, Y> mapper,
    Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

→ agregat

roster

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

The major difference is, that an anonymous interface implementation can have state (member variables) whereas a lambda expression cannot. Look at this interface: from (http://tutorials.jenkov.com/java/lambda-expressions.html)

A Java **lambda** expression is essentially an **object**. You can assign a lambda expression to a variable and pass it around, like you do with any other object.

Note: The semWlambdas differ from anonymous implementations of interfaces. An anonymous interface implementation can have its own instance variables which are referenced via the this reference. However, an lambda cannot have its own instance variables, so this always points to the enclosing object.

See: https://stackoverflow.com/search?q=java+lambda