Lambda Expression

Advanced Programming

Anonymous class - example

```
interface HelloWorld {
    public void greet();
    public void greetSomeone(String someone);
}


HelloWorld frenchGreeting = new HelloWorld() {
     String name = "tout le monde";
     public void greet() {
         greetSomeone("tout le monde");
     }
     public void greetSomeone(String someone) {
         name = someone;
         System.out.println("Salut " + name);
     }

};
```

Anonymous classes

- They do not have a name
- To be used only once
- They are expressions: the class is defined in another expression
- The constructor is default (empty parenthesis): in an interface, there is no constructor
- Anonymous classes are often used in graphical user interface (GUI) applications

In GUI applications

```
public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");

        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
```

EventHandler<ActionEvent> is an interface

Lambda expression

- Because the `EventHandler<ActionEvent>` interface contains only one method, we can use a **lambda expression** instead of an anonymous class expression
- Anonymous classes are instead ideal for implementing an interface that contains two or more methods

Exercise

- From the Oracle documentation
  - Create a feature that enables an administrator to perform any kind of action (e.g., sending a message) on members of the social networking application that satisfy certain criteria

Example

Field	Description
Name	Perform action on selected members
Primary Actor	Administrator
Preconditions	Administrator is logged in to the system.
Postconditions	Action is performed only on members that fit the specified criteria.
Main Success Scenario	1 Administrator specifies criteria of members on which to perform a certain action. 2 Administrator specifies an action to perform on those selected members. 3 Administrator selects the **Submit** button. 4 The system finds all members that match the specified criteria. 5 The system performs the specified action on all matching members.
Extensions	1a. Administrator has an option to preview those members who match the specified criteria before he or she specifies the action to be performed or before selecting the **Submit** button.
Frequency of Occurrence	Many times during the day.

The member of the social network

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

The full class description is
in the code sample of this lecture

List<Person> are the members of the social network

Traditional

```
import java.util.List;


//the main class

public class RosterTest {


    public static void printPersons(
    List<Person> roster, int low, int high) {
        for (Person p : roster) {
            if (p.getGender() == Person.Sex.MALE && low <=
            p.getAge() && p.getAge() < high) {
                p.printPerson();
            }
        }
    }
```


main

```
   public static void main(String... args) {

        // populating the list; the method createRoster instantiate a
        // list with some Person objects

        List<Person> roster = Person.createRoster();


        printPersons(roster, 18, 25);

   }
}
```

With anonymous class

```
import java.util.List;


//the main class

public class RosterTest {
     interface CheckPerson {
        boolean test(Person p);
     }
     public static void printPersons(
             List<Person> roster, CheckPerson tester) {
             for (Person p : roster) {
                 if (tester.test(p)) {
                     p.printPerson();
                 }
             }
         }
```


With anonymous class

```
  public static void main(String... args) {

     // populating the list; the method createRoster instantiate a
     // list with some Person objects

        List<Person> roster = Person.createRoster();

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

Better, but still hard to read considering that the interface has only one method

With lambda expression

```
import java.util.List;


//the main class

public class RosterTest {
     interface CheckPerson {
         boolean test(Person p);
     }
     public static void printPersons(
             List<Person> roster, CheckPerson tester) {
             for (Person p : roster) {
                 if (tester.test(p)) {
                     p.printPerson();
                 }
             }
         }
```


With lambda expression

```
   public static void main(String... args) {

     // populating the list; the method createRoster instantiate a
     // list with some Person objects

        List<Person> roster = Person.createRoster();

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

Here, we do not need to have a new object . There only one method in CheckTest thus we do need to declare a name for it

much leaner!

java.util.function package

- We can further extend the simplicity of our code using parametrised generics with default classes in the function package. For example Predicate

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

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

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

The interface Predicate has a method boolean test.

The interface can be used instead of the specific interface CheckPerson

With Standard Functional Interfaces

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

//the main class

public class RosterTest {

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

With Standard Functional Interfaces

```
  public static void main(String... args) {

     // populating the list; the method createRoster instantiate a
     // list with some Person objects

        List<Person> roster = Person.createRoster();



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

   }                                                    and lambda expression
}
```


Lambda expression

- Methods without a name
- Single parametrised functionality

Lambda expression

1. **A comma-separated list of formal parameters enclosed in parentheses**

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

- You can omit the data **type** of the parameters in a lambda expression or the **parentheses**

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


Lambda expression

2. The arrow token, ->

3. **A body**, which consists of a **single expression or a statement block**.

```
p.getGender() == Person.Sex.MALE
&& p.getAge() >= 18
&& p.getAge() <= 25
```

- As a single expression or with a return statement:

```
p -> {  return p.getGender() == Person.Sex.MALE
          && p.getAge() >= 18
          && p.getAge() <= 25;
     }
```

- Void methods invocation does not need braces:

```
email -> System.out.println(email)
```

More formal parameters

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```


What is the output?

```
interface MyNumber{

    double getValue();

}

MyNumber myNum = () ->123.45;

System.out.println(myNum.getValue());
```

123.45

Maker Interfaces

- Are a tag.
- DEscriptive function
- E.g. Clonable
- Functional interfaces
- one single method with empty body

What is the output?

```
interface MyNumber{
    double getValue();
}
MyNumber myNum;
myNum = () ->Math.random()*100;
System.out.println(myNum.getValue());
```

88.89448331

What is the output?

```
interface NumericTest{
   boolean test(int n);
}
NumericTest isEven = (n) -> (n % 2) ==0;
if(isEven.test(10)) System.out.println("10 is
even");
if(isEven.test(9)) System.out.println("9 is not
even");
```

10 is even
9 is not even

Note

- Parameters must be of the same type
- return value must be of the same type