INFO1113 Object-Oriented Programming

Week 10B: Lambdas, Streams and Method References

Copyright Warning

Topics

- Passing methods (Callback) (s. 4)
- Method References (s. 22)
- Java Functional Interfaces (s. 33)
- Stream API (s. 42)

Callbacks

Within the java language we have are able to effectively wrap methods within an class and attach it to a variable.

Although Java is a very OO language we are able to perform the same through these wrapped objects.

Callbacks

What is a callback?

A callback is where a method is parameterised and invoked from a context which holds the object or method.

Simply: We are able to pass a method or a method wrapped in an object to another method and it can invoke it. This property allows us to provide dynamic method invocation support without know the explicit method.

Callbacks

Let’s break down what is being transferred.

```
interface MyCallback {
    public void callingYou();
}

public class Callback {

    public static void a() {
        MyCallback m1 = new MyCallback() {
            public void callingYou() { System.out.println("Ring ring!"); }
        };
        b(m1);
    }

    public static void b(MyCallback callback) {
        System.out.println("Start of callback");
        callback.callingYou();
        System.out.println("End of callback");
    }

    public static void main(String[] args) {
        a();
    }
}
```

Callbacks

Let's break down what is being transferred.

```
interface MyCallback {
    public void callingYou();
}

public class Callback {

    public static void a() {
        MyCallback m1 = new MyCallback() {
            public void callingYou() { System.out.println("Ring ring!"); }
        };
        b(m1);
    }

    public static void b(MyCallback callback) {
        System.out.println("Start of callback");
        callback.callingYou();
        System.out.println("End of callback");
    }

    public static void main(String[] args) {
        a();
    }
}
```

We create a class that contains a callback method that will be utilised within method **b.**

Callbacks

Let’s break down what is being transferred.

```
interface MyCallback {
    public void callingYou();
}

public class Callback {

    public static void a() {
        MyCallback m1 = new MyCallback() {
            public void callingYou() { System.out.println("Ring ring!"); }
        };
        b(m1);
    }

    public static void b(MyCallback callback) {
        System.out.println("Start of callback");
        callback.callingYou();
        System.out.println("End of callback");
    }

    public static void main(String[] args) {
        a();
    }
}
```

We pass **m1** to the method **b** which can be utilised by this method

Callbacks

Let's break down what is being transferred.

```
interface MyCallback {
    public void callingYou();
}

public class Callback {

    public static void a() {
        MyCallback m1 = new MyCallback() {
            public void callingYou() { System.out.println("Ring ring!"); }
        };
        b(m1);
    }

    public static void b(MyCallback callback) {
        System.out.println("Start of callback");
        callback.callingYou();
        System.out.println("End of callback");
    }

    public static void main(String[] args) {
        a();
    }
}
```

This is represented through the variable **callback** and since it is a **MyCallback** object, we can therefore call it.

Callbacks

What we are doing here is nothing different from what we have done before.

We have been passing objects between methods since the beginning of the course. However we are generalising what kind of objects we can pass and we get the guarantee that a method of a certain signature will exist.

Where are callbacks used?

Callbacks

Callbacks can be utilised within a synchronous and **asynchronous** context.

In the previous example we saw a synchronous usage of a callback that passed an object and allow us to utilise it in a method without that method directly calling a method by its identifier.

Here’s a real world example

Callbacks

Welcome back collections!

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyComparatorProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        Collections.sort(people, new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        });

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

Callbacks

Welcome back collections!

We may encounter a case where we have a collection of objects that we want to sort. In the case of Person, we may have different ways of sorting a list of Person objects.

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyComparatorProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        Collections.sort(people, new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        });

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

Callbacks

Welcome back collections!

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyComparatorProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        Collections.sort(people, new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        });

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

Our class definition for **Person**.

Callbacks

Welcome back collections!

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyComparatorProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        Collections.sort(people, new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        });

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

Create a list with Person objects in it

Callbacks

Welcome back collections!

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyComparatorProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        Collections.sort(people, new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        });

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

We will want to sort a Collection of Person objects by age (or even name!). This requires a creation of a comparator object.

Callbacks

Welcome back collections!

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyComparatorProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        Collections.sort(people, new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        });

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

We will want to sort a Collection of Person objects by age (or even name!). This requires a creation of a comparator object.

Since Collections.sort expects a Comparator object to be passed to it and expects to utilise the compare method.

Okay, what if I wanted to sort by name?

Command-Query Separation

Command-Query separation is the idea that a method or function should only do:

- Command (change state)

You will have written a command already via a set method. This will change or mutate an element. However the method does not return anything.

- Query (Get an answer)

Similarly, a get method, this will retrieve an answer and should always return the same result.

Method References

Java 8 introduced references alongside lambdas. For very simple lambdas which are in essence just invoking another method (such as println).

Since developers are lazy and writing a few more characters of code is a chore, we instead build a system which allows us to utilise a reference to a method.

Refer to Java Language Specification, 15.27. Method Reference Expressions, (https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27)

Method References

Method references allow us to assign and pass methods to other methods. They adhere to the same rules as lambdas in that they require a **functional interface** to be assigned to.

Method References

We are able to get a reference to a method by using **::**
This pair of symbols is used in conjunction with a method identifier.
When assigning a method reference to a variable, the variable must be a functional interface type and contain the same method signature.

Syntax:

`<ClassOrInstance>::MethodIdentifier`

Example:

```
SayHi h = System.out::println;
```

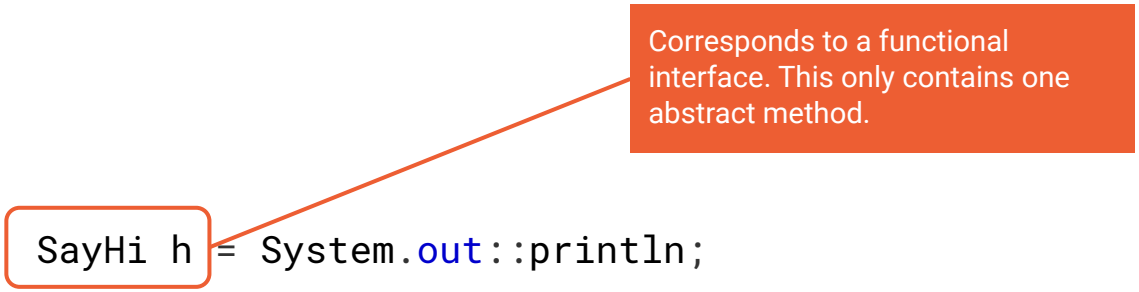
Method References

We are able to get a reference to a method by using **::**
This pair of symbols is used in conjunction with a method identifier.
When assigning a method reference to a variable, the variable must be a functional interface type and contain the same method signature.

Syntax:

```
<ClassOrInstance>::MethodIdentifier
```

Example:



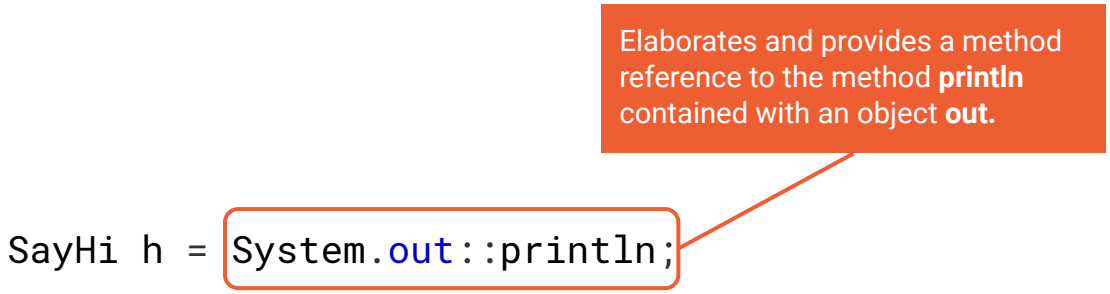
Method References

We are able to get a reference to a method by using **::**
This pair of symbols is used in conjunction with a method identifier.
When assigning a method reference to a variable, the variable must be a functional interface type and contain the same method signature.

Syntax:

```
<ClassOrInstance>::MethodIdentifier
```

Example:



Method References

We are able to get a reference to a method by using **::**
This pair of symbols is used in conjunction with a method identifier.
When assigning a method reference to a variable, the variable must be a functional interface type and contain the same method signature.

Syntax:

```
<ClassOrInstance>::MethodIdentifier
```

Example:

```
SayHi h = System.out::println;
```

Elaborates and provides a method reference to the method **println** contained with an object **out.**

Instead of writing a lambda that is just using another method ie
`SayHi h = (arg) -> System.out.println(arg)`
We would utilise a method reference instead.

Method References

Let’s consider the following:

```
interface MyReference {
    public int ref(int x, int y);
}

public class MethodReferences {

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

    public static int mask(int a, int b) {
        return a & b;
    }

    public static void main(String[] args) {
        MyReference m1 = MethodReferences::add;
        MyReference m2 = MethodReferences::mask;

        System.out.println(m1.ref(1, 1)); //2
        System.out.println(m2.ref(7, 3)); //3
    }
}
```

Method References

Let's consider the following:

```
interface MyReference {
    public int ref(int x, int y);
}

public class MethodReferences {

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

    public static int mask(int a, int b) {
        return a & b;
    }

    public static void main(String[] args) {
        MyReference m1 = MethodReferences::add;
        MyReference m2 = MethodReferences::mask;

        System.out.println(m1.ref(1, 1)); //2
        System.out.println(m2.ref(7, 3)); //3
    }
}
```

For each MyReference type we have given them a method reference.

Method References

Let's consider the following:

```
interface MyReference {
    public int ref(int x, int y);
}

public class MethodReferences {

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

    public static int mask(int a, int b) {
        return a & b;
    }

    public static void main(String[] args) {
        MyReference m1 = MethodReferences::add;
        MyReference m2 = MethodReferences::mask;

        System.out.println(m1.ref(1, 1)); //2
        System.out.println(m2.ref(7, 3)); //3
    }
}
```

We can see from the interface type, it determines what methods can be placed here. We must choose only methods that adhere to this signature.

What would happen if we were to throw an incorrect signature?

Why do we need an object to use a method reference correctly for instance methods?

Functional Interfaces

Java provides general use functional interface library. This is used in conjunction with the **stream api** that collection classes utilise.

The **function** package within java provides a few common interfaces that we can attach lambdas and method references to.

Refer to Java Language Specification, 18.5.3. Functional Interface Parameterization Inference, (https://docs.oracle.com/javase/specs/jls/se8/html/jls-18.html#jls-18.5.3)

Functional Interfaces

Common classes that are utilised by the stream object:

- **Predicate**

A predicate function is where given an argument, the function will return either true or false based on predicate definition.

- **Consumer**

A consumer function is where given an argument, the function will perform an operation onto it. This has no return value

- **Supplier**

A supplier function produces values without any arguments.

- **Function**

The function interface is where given an argument, the definition will provide an output.

Functional Interfaces

Common classes that are utilised by the stream object:

- **Predicate<T>**

A predicate function is where given an argument, the function will return either true or false based on predicate definition.

- **Consumer<T>**

A consumer function is where given an argument, the function will perform an operation onto it. This has no return value

- **Supplier<T>**

A supplier function produces values without any arguments.

- **Function<T, R>**

The function interface is where given an argument, the definition will provide an output.

Functional Interfaces

So we'll go through each functional interface

```
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.Random;

public class FunctionInterfaces {
    public static void main(String[] args) {
        Random random = new Random();
        Predicate<String> p = (String s) -> s.length() > 5;
        Function<String, Integer> f = (String s) -> Integer.valueOf(s.length());
        Consumer<String> i = (String s) -> System.out.println(s);
        Supplier<Integer> s = () ->  Integer.valueOf(random.nextInt()  % 10);

        System.out.println(p.test("This is a string longer than 5"));
        System.out.println(f.apply("Give me the length"));
        i.accept("I'm printing this!");
        System.out.println("This is a randomly outputted integer: " + s.get());
    }
}
```

Functional Interfaces

So we'll go through each functional interface

```
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.Random;

public class FunctionInterfaces {
    public static void main(String[] args) {
        Random random = new Random();
        Predicate<String> p = (String s) -> s.length() > 5;
        Function<String, Integer> f = (String s) -> Integer.valueOf(s.length());
        Consumer<String> i = (String s) -> System.out.println(s);
        Supplier<Integer> s = () ->  Integer.valueOf(random.nextInt()  % 10);

        System.out.println(p.test("This is a string longer than 5"));
        System.out.println(f.apply("Give me the length"));
        i.accept("I'm printing this!");
        System.out.println("This is a randomly outputted integer: " + s.get());
    }
}
```

We expose the function interface classes.

Functional Interfaces

So we'll go through each functional interface

```
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.Random;

public class FunctionInterfaces {
    public static void main(String[] args) {
        Random random = new Random();
        Predicate<String> p = (String s) -> s.length() > 5;
        Function<String, Integer> f = (String s) -> Integer.valueOf(s.length());
        Consumer<String> i = (String s) -> System.out.println(s);
        Supplier<Integer> s = () ->  Integer.valueOf(random.nextInt()  % 10);

        System.out.println(p.test("This is a string longer than 5"));
        System.out.println(f.apply("Give me the length"));
        i.accept("I'm printing this!");
        System.out.println("This is a randomly outputted integer: " + s.get());
    }
}
```

With a Predicate we specify the input type with String and define the lambda expression

Functional Interfaces

So we'll go through each functional interface

```
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.Random;

public class FunctionInterfaces {
    public static void main(String[] args) {
        Random random = new Random();
        Predicate<String> p = (String s) -> s.length() > 5;
        Function<String, Integer> f = (String s) -> Integer.valueOf(s.length());
        Consumer<String> i = (String s) -> System.out.println(s);
        Supplier<Integer> s = () ->  Integer.valueOf(random.nextInt()  % 10);

        System.out.println(p.test("This is a string longer than 5"));
        System.out.println(f.apply("Give me the length"));
        i.accept("I'm printing this!");
        System.out.println("This is a randomly outputted integer: " + s.get());
    }
}
```

Function requires two type arguments, In this case we are using String as input and Integer as the return type

Functional Interfaces

So we'll go through each functional interface

```java
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.Random;

public class FunctionInterfaces {
    public static void main(String[] args) {
        Random random = new Random();
        Predicate<String> p = (String s) -> s.length() > 5;
        Function<String, Integer> f = (String s) -> Integer.valueOf(s.length());
        Consumer<String> i = (String s) -> System.out.println(s);
        Supplier<Integer> s = () ->  Integer.valueOf(random.nextInt()  % 10);

        System.out.println(p.test("This is a string longer than 5"));
        System.out.println(f.apply("Give me the length"));
        i.accept("I'm printing this!");
        System.out.println("This is a randomly outputted integer: " + s.get());
    }
}
```

Consumer only requires an input type and does not return any value. In this case we are inputting a string and using it with **println**

Functional Interfaces

So we'll go through each functional interface

```
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.Random;

public class FunctionInterfaces {
    public static void main(String[] args) {
        Random random = new Random();
        Predicate<String> p = (String s) -> s.length() > 5;
        Function<String, Integer> f = (String s) -> Integer.valueOf(s.length());
        Consumer<String> i = (String s) -> System.out.println(s);
        Supplier<Integer> s = () ->  Integer.valueOf(random.nextInt()  % 10);

        System.out.println(p.test("This is a string longer than 5"));
        System.out.println(f.apply("Give me the length"));
        i.accept("I'm printing this!");
        System.out.println("This is a randomly outputted integer: " + s.get());
    }
}
```

Supplier function that is typically a method (or could be an object) that generates/**supplies** values of type **T.**

Stream API

Each collection within java 8 implements a stream interface where we are able to employ functional interfaces in some of the following methods.

- **map(Function<T, R>)**
  Will return a **Stream**<R> object where it will map elements to the same or different type.

- **filter(Predicate<T>)**
  Will remove elements based on the predicate that is given.

- **forEach(Consumer<T>)**
  Allows you to utilise a **Consumer** method to apply an operations on the stream of items.

Creating a stream

Once we have some collection created, we are able to utilise the stream() method call and get ahold of a **Stream** object.

Syntax:

```
<Collection Instance>.stream()
```

Example:

```
List<Integer> list = new ArrayList<Integer>();
…
int n = list.stream().filter((x) -> x % 2 == 0).count();
Stream<Integer> s = list.stream();
```

Creating a stream

Once we have some collection created, we are able to utilise the stream() method call and get ahold of a **Stream** object.

Syntax:

```
<Collection Instance>.stream()
```

Example:

Instantiate some collection type and add some elements.

```
List<Integer> list = new ArrayList<Integer>();
…
int n = list.stream().filter((x) -> x % 2 == 0).count();
Stream<Integer> s = list.stream();
```

Creating a stream

Once we have some collection created, we are able to utilise the stream() method call and get ahold of a **Stream** object.

Syntax:

```
<Collection Instance>.stream()
```

Example:

Invoke the stream() method to return a Stream object which we can build a query.

```
List<Integer> list = new ArrayList<Integer>();
...
int n = list.stream().filter((x) -> x % 2 == 0).count();
Stream<Integer> s = list.stream();
```

Creating a stream

Once we have some collection created, we are able to utilise the stream() method call and get ahold of a **Stream** object.

Syntax:

```
<Collection Instance>.stream()
```

Example:

Since **stream()**, **map()**, and **filter()** return the **Stream** object back and therefore able to chain methods

```
List<Integer> list = new ArrayList<Integer>();

int n = list.stream().filter((x) -> x % 2 == 0).count();
Stream<Integer> s = list.stream();
```

Creating a stream

Once we have some collection created, we are able to utilise the stream() method call and get ahold of a **Stream** object.

Syntax:

```
<Collection Instance>.stream()
```

Example:

We finalise the stream by using an aggregate method **count()** which will return an integer.

```
List<Integer> list = new ArrayList<Integer>();
…
int n = list.stream().filter((x) -> x % 2 == 0).count();
Stream<Integer> s = list.stream();
```

Creating a stream

Once we have some collection created, we are able to utilise the stream() method call and get ahold of a **Stream** object.

Syntax:

```
<Collection Instance>.stream()
```

Example:

```
List<Integer> list = new ArrayList<Integer>();
…
int n = list.stream().filter((x) -> x % 2 == 0).count();
Stream<Integer> s = list.stream();
```

Without chaining the methods, we are able to grab the Stream object assign it to a variable.

Let’s query the list of people again

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyStreamProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        List<Person> over25 = people.stream().filter((Person p) -> p.getAge() > 25)
            .collect(Collectors.toList());

        System.out.println((over25));
    }
}
```

Let's query the list of people again

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyStreamProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        List<Person> over25 = people.stream().filter((Person p) -> p.getAge() > 25)
            .collect(Collectors.toList());

        System.out.println((over25));
    }
}
```

We can extract out all the Person objects which are over the age of 25.

Let's query the list of people again

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyStreamProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        List<Person> over25 = people.stream().filter((Person p) -> p.getAge() > 25)
            .collect(Collectors.toList());

        System.out.println((over25));
    }
}
```

We use a **Predicate** function with a Person object that allows us to accept only Person objects that adhere to the criteria.

Let's query the list of people again

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyStreamProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        List<Person> over25 = people.stream().filter((Person p) -> p.getAge() > 25)
            .collect(Collectors.toList());

        System.out.println((over25));
    }
}
```

We are able to simply collect all the objects within this list into a separate list using collect.

But what if I just want to iterate through them?

Let's query the list of people again

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() {return age; }
    public String toString() { return "[Name: " + name + " Age:" + age + "]"; }
}
public class MyStreamProgram {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add(new Person("Jim", 28));
        people.add(new Person("Fred", 18));
        people.add(new Person("Veronica", 23));
        people.add(new Person("Lisa", 35));
        people.add(new Person("Alice", 21));

        people.stream().filter((Person p) -> p.getAge() > 25)
            .forEach(System.out::println);


    }
}
```

We can provide the **println** method reference to the **forEach** method. This achieve the same idea to writing a for each loop and checking.

See you next time!