INFO1113 Object-Oriented Programming

Week 10A: Lambda Methods and Anonymous Classes

Slides developed by Tyson Thomas

Copyright Warning

Topics

Anonymous Classes

We are used to writing classes for reusability and type inheritance. However we will visit anonymous classes so we have an understanding of the process behind an assembly of a class and how lambda methods are created.

Refer to Java Language Specification, 15.9.5. Anonymous Class Declarations, (https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.9.5)

Anonymous Classes

An anonymous class is <u>immediately</u> constructed and an instance is returned to the caller.

Syntax:

```
new Type() {
     [fields]
     [methods]
}
```

```
interface SayHello{
          public void hello();
}

SayHello hi = new SayHello() {
          public void hello() {
                    System.out.println("Hello!");
          }
};
```

An anonymous class is <u>immediately</u> constructed and an instance is returned to the caller.

Syntax:

```
new Type() {
     [fields]
     [methods]
}
```

There is a **SayHello** type within our code that we are able to utilise. An anonymous type would implicitly inherit from **SayHello**.

```
interface SayHello{
        public void hello();
}

SayHello hi = new SayHello() {
        public void hello() {
                System.out.println("Hello!");
        }
};
```

Anonymous Classes

An anonymous class is <u>immediately</u> constructed and an instance is returned to the caller.

Syntax:

```
new Type() {
     [fields]
     [methods]
}
```

Within the braces, we are defining the anonymous type. Simply just overriding the method that is required by **SayHello**.

```
interface SayHello{
        public void hello();
}

SayHello hi = new SayHello() {
        public void hello() {
                System.out.println("Hello!");
        }
};
```

Why would we use anonymous classes?

Anonymous Classes

The idea can be considered contrary to the idea of classes and reusability of code.

An anonymous class has the following properties:

- Only one instance of an anonymous class exists
- It is typically declared within a method

So, when would this situation come up?

Anonymous Classes

Let’s consider the following:

```java
interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class Calculator {
    public static void main(String[] args) {
        IntegerBinaryOperation add = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        };
        IntegerBinaryOperation subtract = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        };
        IntegerBinaryOperation multiply = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        };

        System.out.println(add.apply(1, 1)); //2
        System.out.println(subtract.apply(3, 5)); //-2
        System.out.println(add.apply(3, subtract.apply(3, multiply.apply(2, 6)))); //-6
    }
}
```

Anonymous Classes

Let's consider the following:

```
interface IntegerBinaryOperation {
    int apply(int x, int y);
}
```

Define our interface. We want to define some binary integer operation objects. This will allow a simple method (**apply**) to be implemented.

```
public class Calculator {
    public static void main(String[] args) {
        IntegerBinaryOperation add = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        };
        IntegerBinaryOperation subtract = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        };
        IntegerBinaryOperation multiply = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        };

        System.out.println(add.apply(1, 1)); //2
        System.out.println(subtract.apply(3, 5)); //-2
        System.out.println(add.apply(3, subtract.apply(3, multiply.apply(2, 6)))); //-6
    }
}
```

Anonymous Classes

Let's consider the following:

```
interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class Calculator {
    public static void main(String[] args) {
        IntegerBinaryOperation add = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        };
        IntegerBinaryOperation subtract = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        };
        IntegerBinaryOperation multiply = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        };

        System.out.println(add.apply(1, 1)); //2
        System.out.println(subtract.apply(3, 5)); //-2
        System.out.println(add.apply(3, subtract.apply(3, multiply.apply(2, 6)))); //-6
    }
}
```

Instantiate and we will be creating a new object from an implemented.

Anonymous Classes

Let's consider the following:

```
interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class Calculator {
    public static void main(String[] args) {
        IntegerBinaryOperation add = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        };
        IntegerBinaryOperation subtract = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        };
        IntegerBinaryOperation multiply = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        };

        System.out.println(add.apply(1, 1)); //2
        System.out.println(subtract.apply(3, 5)); //-2
        System.out.println(add.apply(3, subtract.apply(3, multiply.apply(2, 6)))); //-6
    }
}
```

Define the method within the type.

At this point we are writing an anonymous class and instantiating it.

Anonymous Classes

Let’s consider the following:

```
interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class Calculator {
    public static void main(String[] args) {
        IntegerBinaryOperation add = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        };
        IntegerBinaryOperation subtract = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        };
        IntegerBinaryOperation multiply = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        };

        System.out.println(add.apply(1, 1)); //2
        System.out.println(subtract.apply(3, 5)); //-2
        System.out.println(add.apply(3, subtract.apply(3, multiply.apply(2, 6)))); //-6
    }
}
```

We have multiple anonymous classes that have a differing implementation for the **apply** method.

Let's consider the following:

```
interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class Calculator {
    public static void main(String[] args) {
        IntegerBinaryOperation add = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        };
        IntegerBinaryOperation subtract = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        };
        IntegerBinaryOperation multiply = new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        };

        System.out.println(add.apply(1, 1)); //2
        System.out.println(subtract.apply(3, 5)); //-2
        System.out.println(add.apply(3, subtract.apply(3, multiply.apply(2, 6)))); //-6
    }
}
```

Since each type implements the methods within the interface, we are able to treat it as the interface type and therefore utilise the apply method with each.

This seems like a long and convoluted way to do something very simple!

Anonymous Classes

Yes! But there is an advantage to anonymous classes.

For example, within a GUI, a button's event may never be used by any other button.

We may want to hold a collection of commands and each command contains a unique implementation of a method.

We are identifying a pattern with a method and its usage.

Let’s take have a look at the following modifications:

```java
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}
public class Calculator {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations = new HashMap<>();
        operations.put("ADD", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        });
        operations.put("SUB", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        });
        operations.put("MUL", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        });
        System.out.println(operations.get("ADD").apply(1, 1)); //2
        System.out.println(operations.get("SUB").apply(3, 5)); //-2
        System.out.println(operations.get("ADD").apply(3, operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6)))); //-6
    }
}
```

Let’s take have a look at the following modifications:

We are able to specify a type that the anonymous class will implement.

```java
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}
public class Calculator {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations = new HashMap<>();
        operations.put("ADD", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        });
        operations.put("SUB", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        });
        operations.put("MUL", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        });
        System.out.println(operations.get("ADD").apply(1, 1)); //2
        System.out.println(operations.get("SUB").apply(3, 5)); //-2
        System.out.println(operations.get("ADD").apply(3, operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6)))); //-6
    }
}
```

Let's take have a look at the following modifications:

We are able to store the operations within a collection and refer to them from a string.

```java
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}
public class Calculator {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations = new HashMap<>();
        operations.put("ADD", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        });
        operations.put("SUB", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        });
        operations.put("MUL", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        });
        System.out.println(operations.get("ADD").apply(1, 1)); //2
        System.out.println(operations.get("SUB").apply(3, 5)); //-2
        System.out.println(operations.get("ADD").apply(3, operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6)))); //-6
    }
}
```

Anonymous Classes

Let's take have a look at the following modifications:

```
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}
public class Calculator {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations = new HashMap<>();
        operations.put("ADD", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x + y;
            }
        });
        operations.put("SUB", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x - y;
            }
        });
        operations.put("MUL", new IntegerBinaryOperation() {
            public int apply(int x, int y) {
                return x * y;
            }
        });
        System.out.println(operations.get("ADD").apply(1, 1)); //2
        System.out.println(operations.get("SUB").apply(3, 5)); //-2
        System.out.println(operations.get("ADD").apply(3, operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6)))); //-6
    }
}
```

Using the key for the element, we are able to extract the method and execute it.

So let’s extend our program to support this

Lambdas

Lambda methods require an interface that declares only one method.

After an interface has been defined and only contains one abstract method, it can adhere allow the usage of lambda methods.

Syntax:

```
(arg1[, arg2…]) -> functionBody
```

```
SayHello hi = () -> System.out.println("Hello!");
NumericOperation add = (x, y) -> x + y
NumericOperation add = (int x, int y) -> x + y
```

Prior to Java 8, lambdas does not exist.

Refer to Java Language Specification, 15.13. Lambda Expressions, (https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.13)

Lambdas

Lambda methods require an interface that declares only one method.

After an interface has been defined and only contains one abstract method, it can adhere allow the usage of lambda methods.

Syntax:

```
(arg1[, arg2…]) -> functionBody
```

We define the expression using the paranethesis and -> arrow.

```
SayHello hi = () -> System.out.println("Hello!");
NumericOperation add = (x, y) -> x + y
NumericOperation add = (int x, int y) -> x + y
```

Lambdas

Lambda methods require an interface that declares **only one method.**

After an interface has been defined and only contains one **abstract** method, it can adhere allow the usage of lambda methods.

Syntax:

```
(arg1[, arg2…]) -> functionBody
```

Afterwards is our expression for our lambda

```
SayHello hi = () -> System.out.println("Hello!");
NumericOperation add = (x, y) -> x + y
NumericOperation add = (int x, int y) -> x + y
```

Lambdas

This looks similar to our previous but with lambdas!

```
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class CalculatorLambdas {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations = new HashMap<>();
        operations.put("ADD", (x, y) -> x + y);
        operations.put("SUB", (int x, int y) -> x - y);
        operations.put("MUL", (x, y) -> x * y);

        System.out.println(operations.get("ADD").apply(1, 1)); //2
        System.out.println(operations.get("SUB").apply(3, 5)); //-2
        System.out.println(operations.get("ADD").apply(3,
            operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6)))); //-6
    }
}
```

This looks similar to our previous but with lambdas!

```java
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class CalculatorLambdas {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations = new HashMap<>();
        operations.put("ADD", (x, y) -> x + y);
        operations.put("SUB", (int x, int y) -> x - y);
        operations.put("MUL", (x, y) -> x * y);

        System.out.println(operations.get("ADD").apply(1, 1)); //2
        System.out.println(operations.get("SUB").apply(3, 5)); //-2
        System.out.println(operations.get("ADD").apply(3,
            operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6)))); //-6
    }
}
```

We still have the hashmap storing the operations, however we are using lambda expressions instead

This looks similar to our previous but with lambdas!

```
import java.util.HashMap;

interface IntegerBinaryOperation {
    int apply(int x, int y);
}

public class CalculatorLambdas {
    public static void main(String[] args) {
        HashMap<String, IntegerBinaryOperation> operations = new HashMap<>();
        operations.put("ADD", (x, y) -> x + y);
        operations.put("SUB", (int x, int y) -> x - y);
        operations.put("MUL", (x, y) -> x * y);

        System.out.println(operations.get("ADD").apply(1, 1)); //2
        System.out.println(operations.get("SUB").apply(3, 5)); //-2
        System.out.println(operations.get("ADD").apply(3,
            operations.get("SUB").apply(3,
            operations.get("MUL").apply(2, 6)))); //-6
    }
}
```

Since the interface adheres to a functional interface, we are able to write a method that resembles the only abstract method signature.

Can lambdas have multiple lines?

Lambdas

YES!

Syntax:

```
(arg1[, arg2…]) -> { functionBody }
```

Example:

We are able to specify multiple lines in a lambda method by utilising the curly brace.

```
SayHello hi = () -> {
            System.out.println("Hello!");
            System.out.println("Yo!");
        };
```

What about default methods?

Excellent question!

Referring to the java language specification of what is considered a **Functional Interface**:

“A functional interface is an interface that has just one **abstract** method (aside from the methods of Object), and thus represents a single function contract.”

https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.8

So we can use default methods in lambda expressions?

Expanding on the JLS definition:

“Practically speaking, it is unusual for a lambda expression to need to talk about itself (either to call itself recursively or to invoke its other methods), while it is more common to want to use names to refer to things in the enclosing class that would otherwise be shadowed `(this, toString())`.

If it is necessary for a lambda expression to refer to itself (as if via this), a method reference or an anonymous inner class should be used instead. “

https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27.2

Default Methods

So! We can have default methods within an interface and also allow that interface to be a **functional interface** but we cannot use them within lambda expressions.

However! **We can use the lambda expression within our default methods!**

Let's consider the following example:

```java
interface SayHello {
    public default void howAreYou() { hello(); System.out.println("How are you today?"); }
    public void hello();
}

public class Hello {
    public static void main(String[] args) {
        SayHello hi = () -> {
            System.out.println("Hello!");
        };
        hi.howAreYou();
    }

}
```

Let's consider the following example:

```
interface SayHello {
    public default void howAreYou() { hello(); System.out.println("How are you today?"); }
    public void hello();
}

public class Hello {
    public static void main(String[] args) {
        SayHello hi = () -> {
            System.out.println("Hello!");
        };
        hi.howAreYou();
    }

}
```

We specify a default method that utilises the eventually defined abstract method.

So what’s the difference?

Anonymous Classes

Beyond the syntax and brevity it may seem like that there is no difference between an anonymous class and a lambda.

However we are only scratching the surface between them. Specifically we are able to do more with anonymous classes such as:

- Create instance variables
- Multiple methods
- Encapsulation of fields

See you next time!