INFO1113 / COMP9003
Object-Oriented Programming

Lecture 10

THE UNIVERSITY OF SYDNEY

Contents developed by Tyson Thomas

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Topics

- **Anonymous Classes**
- **Java Lambdas**
- **Difference between these two**

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 immediately 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!");
        }
};
```

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

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

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

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
  }
}
```

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

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

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.

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
   }
}
```

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.

Why would we use anonymous classes?

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.

Anonymous Classes

Let's 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
   }
}
```

Anonymous Classes

Let's 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
   }
}
```

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

Anonymous Classes

Let's 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
   }
}
```

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

Anonymous Classes

Let's 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…]) -> methodBody

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…]) -> methodBody
```

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

IntegerBinaryOperation add = (x, y) -> x + y;

IntegerBinaryOperation add = (int x, int y) -> x + y;
```

We define the expression using the parenethesis and **->** arrow.

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
   }
}
```

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
   }
}
```

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

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
   }
}
```

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:

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

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

What about default methods?

Lambdas

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?

Default Methods

Yes! 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**!

Lambdas

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();
   }

}
```

Lambdas

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.

Demo

Difference between Anonymous class and Lambda Expression

Difference

Anonymous Class	Lambda Expression
It is a class without name.	It is a method without name. (anonymous function)
It is the best choice if we want to handle interface with multiple methods.	It is the best choice if we want to handle functional interface.
At the time of compilation, a separate .class file will be generated.	At the time of compilation, no separate .class file will be generated. It simply convert it into private method of the outer class.
Memory allocation is on demand, whenever we are creating an object.	It resides in a permanent memory of JVM.

See you next time!

THE UNIVERSITY OF SYDNEY