Department of Computer Science
University of Bristol

COMSM0103 Object Oriented Programming with Java

LAMBDAS & STREAMS

Sion Hannuna | sh1670@bris.ac.uk

Let's start at the end

Live code demo

Where does this new syntax come from?

```
String[] names = {"Sebastian", "Mutalib", "Tom", "Tilo" …
Arrays.stream(names)
        .filter(x -> x.startsWith("S"))
        .sorted()
        .forEach(System.out::println);
```

Stream<T>	filter(Predicate<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.

Interface Predicate<T> has a single abstract method, test:

boolean	test(T t) Evaluates this predicate on the given argument.

How do we reconcile the arguments to `filter` with this?

Predicate four ways, One

```
public class Main implements Predicate<String> {
    //@Override
    public boolean test(String s) {
        return s.startsWith("S");
    }

    private void predicateOne(){
        String[] names = {"Sebastian", "Mutalib”…
        Arrays.stream(names)
                .filter(this)
                .sorted()
                .forEach(System.out::println);
    }
…
```

Predicate four ways, Two

```
private void predicateTwo(){
    String[] names = {"Sebastian", "Mutalib", …
    Predicate <String> predicate= new Predicate <String>(){
        public boolean test(String s) {
            return s.startsWith("S");
        }
    };

    Arrays.stream(names)
            .filter(predicate)
            .sorted()
            .forEach(System.out::println);
}
```

Predicate four ways, Three

```
private void predicateThree(){
    String[] names = {"Sebastian", "Mutalib…
    Predicate <String> predicate=(s) ->  s.startsWith("S");

    Arrays.stream(names)
            .filter(predicate)
            .sorted()
            .forEach(System.out::println);
}
```

Predicate four ways, Four – back to where we started

```
private void predicateFour(){
    String[] names = {"Sebastian", "Mutalib", …
    Arrays.stream(names)
            .filter(x -> x.startsWith("S"))
            .sorted()
            .forEach(System.out::println);
}
```

"…whereas some declarative programmers only pay lip service to equational reasoning, users of functional languages exploit them every time they run a compiler, whether they notice it or not...."

--- *Philip Wadler*

Recap: Strategy Pattern

Recap: Strategy Pattern

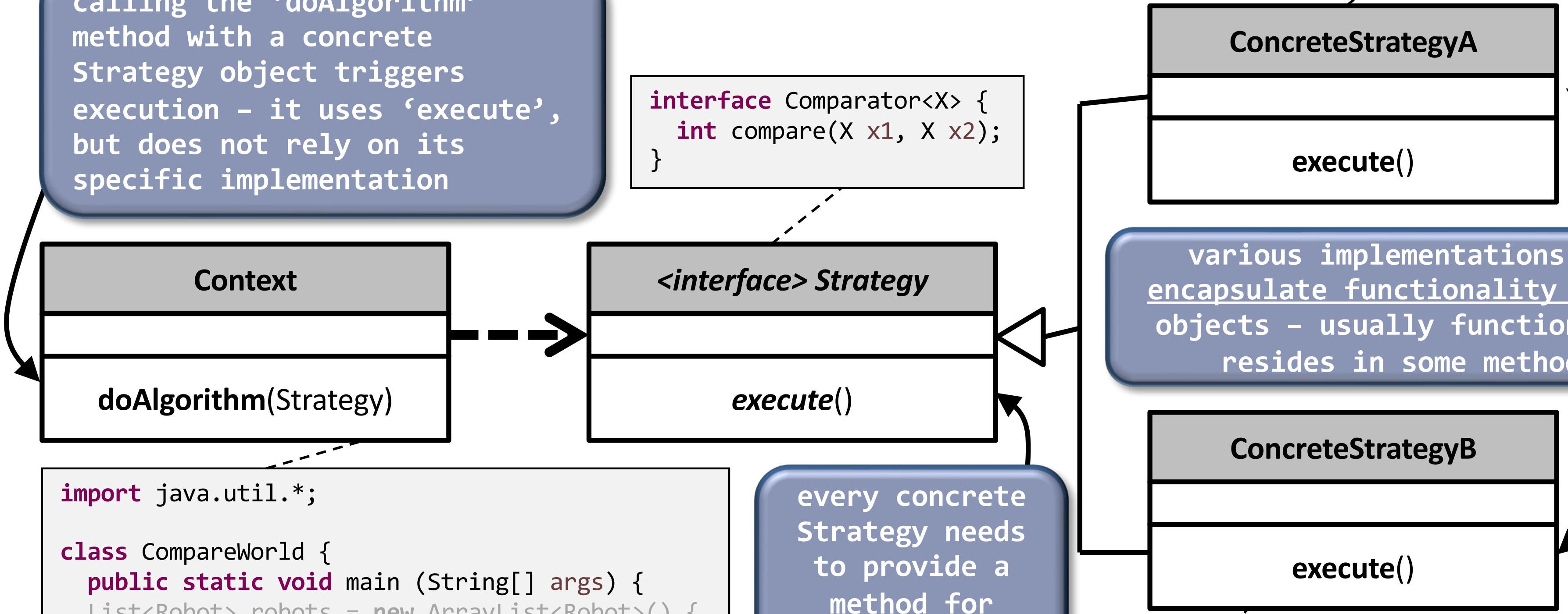
SYNOPSIS | UML | code | comments

The Strategy Pattern defines a set of encapsulated algorithms that can be swapped to carry out a specific behaviour. [GoF]

```java
import java.util.Comparator;
public class RobotLegsComparator implements Comparator<Robot> {
  public int compare(Robot robotA, Robot robotB) {
    return (robotA.numLegs - robotB.numLegs);
} }
```

calling the 'doAlgorithm' method with a concrete Strategy object triggers execution – it uses 'execute', but does not rely on its specific implementation

```java
interface Comparator<X> {
  int compare(X x1, X x2);
}
```



various implementations can encapsulate functionality within objects – usually functionality resides in some methods

every concrete Strategy needs to provide a method for execution

```java
import java.util.*;

class CompareWorld {
  public static void main (String[] args) {
  List<Robot> robots = new ArrayList<Robot>() {
    { add(new CarrierRobot());
      add(new Robot("C3PO"));
    } };
  robots.get(0).charge(10);
  robots.sort(new RobotPowerComparator());
  robots.sort(new RobotLegsComparator());
} }
```

```java
import java.util.Comparator;
class RobotPowerComparator implements Comparator<Robot> {
  public int compare(Robot robotA, Robot robotB) {
    return (Math.round(robotA.powerLevel - robotB.powerLevel));
} }
```

Recap:
Anonymous Inner Classes

Recap: Anonymous Instantiation of Inner Classes

- inner classes are defined within another class
- anonymous (inner) classes are defined and instantiated in a single place using **new**, where the anonymous class definition itself is actually an expression
- inner classes are often local helper classes, whilst anonymous classes are often use-once helper classes without an explicit handle to the code that defines it

```
import java.util.Comparator;
class RobotPowerComparator implements Comparator<Robot> {
  public int compare(Robot a, Robot b) {
    return (Math.round(a.powerLevel - b.powerLevel));
} }
```

```
import java.util.*;
class CompareWorld {
  public static void main (String[] args) {
    SortedSet<Robot> robots =
      new TreeSet<>(
        new Comparator<Robot>() {
          public int compare(Robot a, Robot b) {
            return (Math.round(a.powerLevel - b.powerLevel));
        } }
    );
    Robot c3po = new Robot("C3PO");
    c3po.charge(10);
    robots.add(c3po);
    robots.add(new CarrierRobot());
    System.out.println(robots);
} }
```

instead of defining a new class in a new file, we can create and define a class ‘in-situ’ – this removes a lot of overhead, yet provides no handle for using the definition again for another object

That’s a lot of code!

If Java is the answer, it must have been a really **verbose question**.

A First Motivation for 'Code as Data'

- thus, sometimes we use message parameters to hand over objects to the receiver in order to **provide** the object's **method capabilities**

```
...
class CompareWorld {
  public static void main (String[] args) {
    SortedSet<Robot> robots =
      new TreeSet<Robot>(new Comparator<Robot>() {
        public int compare(Robot a, Robot b) {
           return (Math.round(a.powerLevel - b.powerLevel));
        }
      });
  ...
} }
```

the anonymous inner class (in red) serves as a parameter to supply the TreeSet instance with the functionality for comparing robots

- however, we still have to **write a whole class** to supply just the functionality of a single method to the receiver
- it would be handy to allow just the code (i.e. a method body with its parameters) as arguments in method calls
- more generally, we would like to **reference computational functionality** (other OO languages use function pointers etc)

Lambdas

Lambdas and Single Abstract Method (SAM) Interfaces

- for single-method interfaces, Java allows to replace an anonymous inner class with just 'the essence' of its only method: 1) the input parameters, 2) the **->** arrow symbol, and 3) an expression or code block that produces the result

```
...
class CompareWorld {
  public static void main (String[] args) {
    SortedSet<Robot> robots =
      new TreeSet<Robot>(new Comparator<Robot>() {
        public int compare(Robot robotA, Robot robotB) {
           return (Math.round(robotA.powerLevel - robotB.powerLevel));
        }
      });
    ...
} }
```

```
interface Comparator<X> {
  int compare(X x1, X x2);
}
```

...the single abstract method (SAM) interface...

```
...
class CompareWorld {
  public static void main (String[] args) {
   SortedSet<Robot> robots =
      new TreeSet<Robot>((robotA, robotB) ->
        Math.round(robotA.powerLevel - robotB.powerLevel));
    ...
} }
```

1) the type of the input variables has to match the interface signature (here: type inference from type parameter is used)

type inference allows to connect our Lambda to the interface it implements (which is never mentioned!)

3) the result of this expression is automatically mapped to be the return value of the compare method – the return type has to match up with the interface definition of the only method

2) the arrow symbol indicates the use of a Lambda expression

Basic Concepts around Lambdas

- conceptually, a lambda expression is an unnamed function, a piece of reusable code that can be treated as functionality data that is passed around (used as arguments etc)
- it has a type signature (from the interface it is encapsulated within) and a body (the provided code block), but no name
- yet, a Lambda can be referenced just as objects can be:

```
...
class CompareWorld {
  public static void main (String[] args) {
     Comparator<Robot> comp = (robotA, robotB) ->
       Math.round(robotA.powerLevel - robotB.powerLevel);
    SortedSet<Robot> robots = new TreeSet<>(comp);
  ...
} }
```

here the lambda expression is held in a reference ‘comp’ of type ‘Comparator<...>’, in this sense it still is an object with memory address etc

the reference can be used in the same way as an object of the same type (a.k.a. the lambda object), in fact both are conceptually identical

- in contrast to some functional languages such as Haskell, in Java a Lambda may or may not be pure, i.e., may or may not have any side effects

Impure Lambdas and Side Effects

- since a Lambda can contain a code block, all objects or state in scope and accessible **may be mutated** – as a result such Lambdas are not pure anymore and have **side effects**:

```
...
class CompareWorld {
  public static void main (String[] args) {
    Comparator<Robot> comp = (robotA, robotB) -> {
      robotA.charge(10);
      return Math.round(robotA.powerLevel - robotB.powerLevel);
    };
    SortedSet<Robot> robots =
      new TreeSet<>(comp);
    ...
} }
```

this line manipulates state outside the local scope of the function – the full effects are often difficult to forecast (therefore: minimize side effects as much as possible for clearer, usually better programs)

```
...
class CompareWorld {
  public static void main (String[] args) {
    final Robot robot = new Robot();
    Comparator<Robot> comp = (robotA, robotB) -> {
      robot.charge(10);
      return Math.round(robotA.powerLevel - robotB.powerLevel);
    };
    SortedSet<Robot> robots =
      new TreeSet<>(comp);
    ...
} }
```

potentially even more problematic, in Java objects outside the set of input arguments may be manipulated; here a robot object is ‘charged’, which is not one of the input arguments

Functional Interfaces

Using Lambdas with Functional Interfaces

- as mentioned before, Lambdas can be seen as implementations of **`interface`**s with one single method
- thus, often programmers provide an **`interface`** of the suitable signature, and then instantiate an object that implements this **`interface`** via a Lambda expression
- the encapsulating object is then typed by the **`interface`** and can reference the Lambda
- this way, the Lambda can be passed around like an object

```
@FunctionalInterface
public interface MyStringTransform {
  String transform(String string);
}
```

```
public class FunctionalWorld {

  static public String doTransform(
    String string,
    MyStringTransform f) {
      return f.transform(string);
  }

  public static void main(String args[]) {
    MyStringTransform transform =
      input -> input + " transformed";
    String result = doTransform("message",
                        transform);
    System.out.println(result);
} }
```

Artefacts of Functional Interfaces

- as mentioned before, a functional interface is an interface which contains **exactly one method**
- as we have seen with **`Comparator`**, such interfaces existed already before Lambdas were introduced in Java 8, e.g. **`Runnable, Readable, Callable, Iterable, Closeable, Flushable, Formattable, Comparable,`** or **`FileFilter`**
- the Java designers decided that the compiler converts each lambda expression to a matching functional interface type – this is possible in every case
- there are some **artefacts** created by this notion: for instance, there may now be various, incompatible interfaces of identical type

```
...
interface Executable {
   public void execute();
}
...
class CompareWorld {
  public static void main (String[] args) {
   ...
    Runnable runMe = () -> {};
    Executable executeMe = () -> {};
    //runMe = executeMe; //DOES NOT WORK WITHOUT CAST!
   ...
} }
```

More on Functional Interfaces

- all functional interfaces are recommended to have an informative **`@FunctionalInterface`** annotation in their code
- **`default`** methods are not abstract; thus, a functional interface may still have multiple **`default`** methods
- Java also provides a simple, generic functional interface that receives one value and returns another; the **`interface Function<T,R>`** represents it
- many other functional interfaces are shipped with Java; we will cover some of them in the future

```java
import java.util.function.Function;
class FunctionalWorld {

  static public String doTransform(
    String string,
    Function<String,String> f) {
            return f.apply(string);
  }

  public static void main(String args[]) {
    Function<String,String> transform =
      input -> input + " transformed";
    String result = doTransform("message",
                transform);
    System.out.println(result);
} }
```

Example: Overloading with Functional Interfaces

```
import java.util.function.Consumer;
import java.util.function.Function;

public interface OverloadInterface {
  String doTransform(Function<String, String> f);
  void doTransform(Consumer<Integer> f);
}
```

```
import java.util.function.Consumer;
import java.util.function.Function;

public class OverloadWorld implements OverloadInterface {
  @Override
  public  String doTransform(Function<String, String> f) {
    return f.apply("Something ");
  }
  @Override
  public void doTransform(Consumer<Integer> f) {}

  public static void main(String args[]) {
    OverloadWorld myInstance = new OverloadWorld();
    String result = myInstance.doTransform(a -> a + " transformed");
    System.out.println(result);
} }
```

tricky type inference that allows to match the Lambda to one of two overloaded abstract methods – try to avoid overloading in this case

Bulk Operations

Motivation for Bulk Operations

- problem of replacing serial, single-threaded execution by parallel, multi-threaded one: we need an interface for this!
- **`Iterators`** explicitly step-through a **`Collection`** serially
- Idea: the user supplies an operation (via a Lambda) and can apply it to an entire **`Collection`** automatically
  → element-wise code (the Lambda) and application to elements (possibly parallelisation) are now separated
- this notion of **internal iteration** (handing over your functionality to a method that applies it to all elements) replaces **external iteration** (getting each element and applying the functionality to it in an explicit loop)

forEach

- instead of using an **Iterator**, one can use a method that is attached to every **Collection** for applying functionality:

```
public void forEach(Consumer<? super T> consumer);
```

- the associated **Consumer** interface (you will remember it from your courseworks) is called by for each for all elements:

```
public interface Consumer<T> { public void accept(T t); }
```

- **Consumer** is a functional SAM interface, thus we supply **forEach**'s parameter as a Lambda instead of a **Consumer** object:

```
class RobotWorld {
  public static void main (String[] args) {
    List<Robot> robots = new ArrayList<Robot>() {
      { add(new Robot("C3PO"));
        add(new Robot("C4PO"));
        add(new Robot("C5PO"));
      }
    };
    robots.forEach(robot -> robot.charge(10)); ...
} }
```

this is an alternative to an external for-loop that steps through each element and calls the 'charge' method on it, instead we supply a Lambda as functionality to be executed 'forEach' element

Streams

- in the case of `forEach()`, the `Collection` is responsibility for supplying the means to access all elements and how to apply the specified functionality to them, the user is responsible for supplying functionality to be applied to each of the sequence elements
- Java 8 introduces a new interface `Stream<E>` and a method `stream()` for Collections to turn the latter into the former – this offers a subset-view to a Collection
- Java supplies many more bulk operations for streams including `forEach`, `filter`, `map`, `reduce` ... to name a few
- some of these methods produce a `Stream` as a return type, thus one can easily produce **chains of processing** (this style is also known as **fluent programming**)

Concept of Fluent Programming

- resulting **method cascading** (i.e. chaining) produces linearly readable code, which is often easy to understand

```
import java.util.*;

class RobotWorld {
  public static void main (String[] args) {
    List<Robot> robots = new ArrayList<Robot>() {
      { add(new Robot("C3PO"));
        add(new Robot("C4PO"));
        add(new Robot("C5PO"));
      }
    };
    robots
      .stream()
      .filter(r -> !r.name.equals("C3PO"))
      .map(r -> r.name)
      .forEach(s -> System.out.println(s));
} }
```

FLUENT STYLE

get a Stream view of the List of Robot objects

'filter' takes a Stream and returns a Stream retaining only those elements for which the Lambda expression is true; here: two elements C4PO and C5PO are in the result

'map' takes a Stream and returns a Stream replacing every element with the result of the Lambda expression; here: the Stream of two Robot objects is turned into a Stream of two String objects containing the String "C4PO" and the String "C5PO"

finally, we iterate through all members of the Stream using internal iteration, printing out each String

- This **fluent programming style** is a departure from classical object-orientation! – (avoid in your coursework if possible)

To Do

- recap content and check out the unit website
- write, compile, run and understand **ALL** the tiny programs from the lectures so far

→Remember that we currently recommend 9 hours of time to go into this unit per week overall – work in your team of two as often as possible. *Ask yourself: Would you be ready for a theory exam on OO tomorrow? Am I far enough through the project? Is my team working efficiently?*