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*

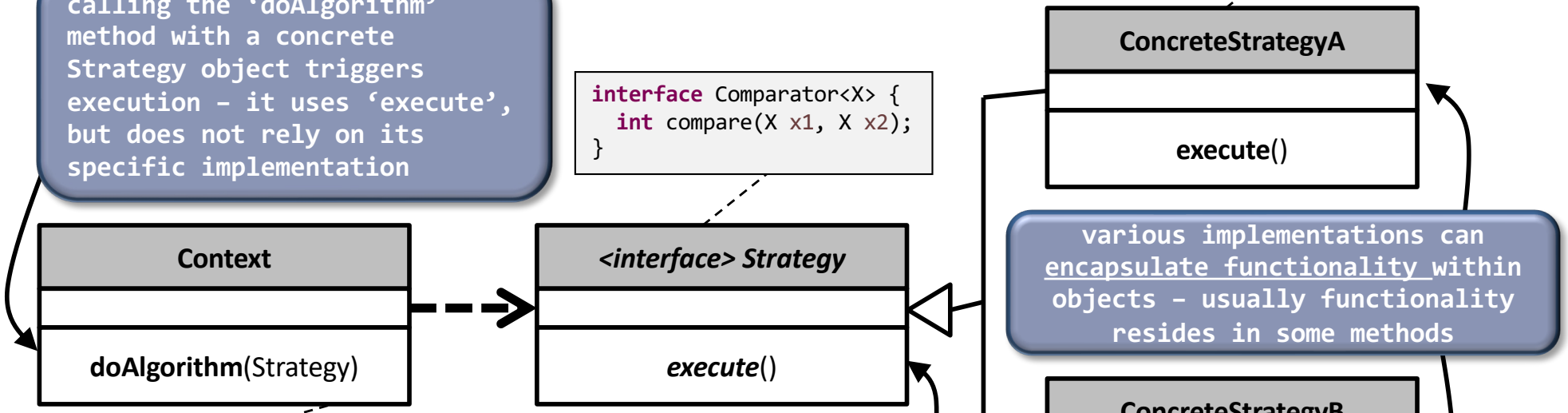
Recap: Strategy Pattern

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]

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



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

every concrete Strategy needs to provide a method for execution

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

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

```
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());
} }
```

```
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));
        }
      });
  ...
} }
```

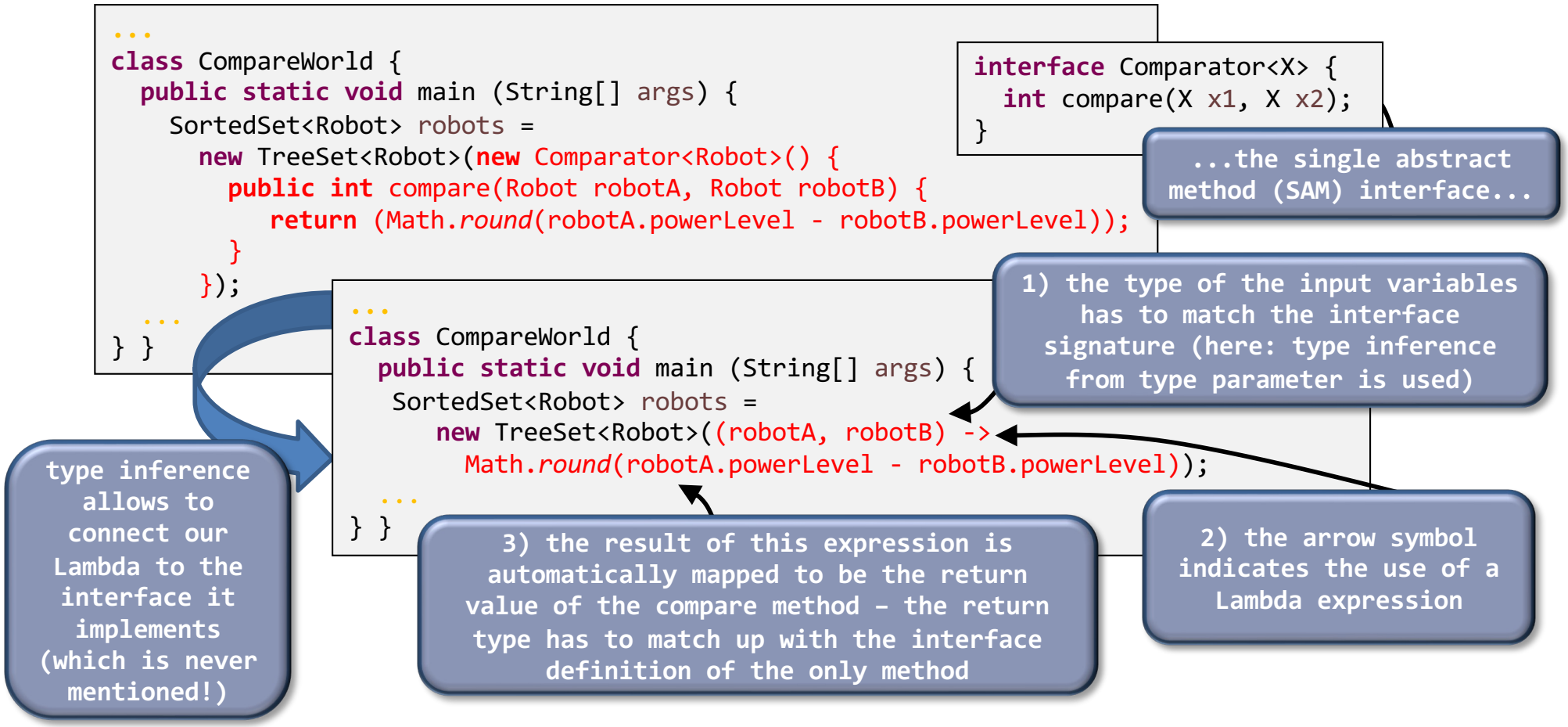
the anonymous inner class (in red) serves as a parameter to supply the TreeSet instance with the functionality for comparing robots

- 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



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