CSEC/SWEN-124
Software Development & Problem Solving

8.2: Anonymous Classes

RIT | Golisano College of Computing and Information Sciences
Department of Computing Security

Lambdas

```
public interface Shape {
   double area (double length);
}

public static void main (String[] args) {
   int x = 5;
   int height = 10;

   Shape rectangle =
                (width) -> width * height;

   System.out.println (rectangle.area (x));
}
```

Lambda expressions are a condensed version of anonymous classes for functional interfaces.

- In this unit we will learn about a different way to create and use classes.
  - Using classes from inside other classes.
  - Creating on the fly classes
  - Simplifying the class declaration syntax with lambdas
  - Working with streams
- We will be exploring several different ways to generate and use classes, including:
  - Inner Classes ✓
  - Anonymous Classes ✓
  - Lambdas ➤
  - Streams ➤
- Today we will focusing on a condensed notation for anonymous classes, known as lambdas. We'll also explore some Java streams which traditionally make heavy use of lambdas.

- Java allows the creation of class ***instances*** without ever formally declaring the class
- This is called an ***anonymous*** class
- Start by creating a new instance of a base ***class*** or ***interface***
- After the () add ***{}*** and include any implementation details
- It is advised to only use anonymous classes when only a couple methods must be implemented

Review: Anonymous Class

```
public interface Shape {
    double area (double length);
}

public static void main (String[] args) {
    int x = 5;
    Shape square = new Shape () {
        public double area (double length){
            return length * length;
        }
    };
    System.out.print (square.area(x));
}
```

Passing the length to area is a little kludgy, but it will make demonstrating lambdas in the coming slides easier.

Functional Interface

```
public interface Shape {
    double area (double length);
}
```

A ***functional interface*** is an interface that only contains a single method.

- Interfaces that have only ***one method***, are known as ***functional interfaces***
- They are effectively the closest thing you can get to a traditional function in a fully object oriented language like Java
  - Traditional meaning a non-method function, like the ones we wrote in python
- What are some functional interfaces that have been used in the class so far?
  - `Comparable`
  - `Comparator`
  - `Iterable`
- They are often the target of ***anonymous*** classes because they are so small

Lambda I

- Since functional interfaces only have a singular method there is a lot of information that can be ***inferred*** from them
- This is exploited by **lambda** expressions which can be used in the place of anonymous classes
- The easiest information to infer is the ***name of the method*** in the functional interface
- When using lambda expressions the inferred information will not be written
- The general lambda syntax is the function's ***parameter list*** followed by a `->` then the body of the function

```
public interface Shape {
   double area (double length);
}

public static void main (String[] args) {
   int x = 5;

   Shape circle = (double radius) -> {
      return Math.PI * Math.pow (radius, 2);
   };

   System.out.println (circle.area (x));
}
```

The arrow (`->`) is used to separate a functional interface's parameters and implementation in a ***lambda expression***.

Lambda II

```
public interface Shape {
   double area (double length);
}

public static void main (String[] args) {
   int x = 5;
   int height = 10;

   Shape rectangle =
                  width -> width * height;

   System.out.println (rectangle.area (x));
}
```

Notice the height is specified as a ***local variable*** since it cannot be passed in without changing the interface.

- What other information can be inferred from the interface?
  - parameter types
  - return (type)
- With Java being a typed language, we know the ***types of the parameters***, so that information can be removed from the lambda expression
- Since we know the function **returns**, we can remove that as well
- The ***curly braces*** can be removed if the statement is only one line
- If there is only one parameter, you may also omit the ***parenthesis*** surrounding it
- *Reminder*: Since lambda are a form of anonymous class they can also access local variables directly (as seen in the example)

Lambda III

- Methods, just like everything else in Java, have a memory address associated with them
- For ***static methods***, we can refer to that memory address using a ***method reference***
  - ***Interface methods*** can also be accessed via a method reference
- A method reference is accessed using the class/interface name a double colon (**::**) and the method's name
  - Example: `Iterable::iterator`
- Do not include any ***parameters***
- When available, you can simplify a lambda even further by using a method reference instead of a local implementation

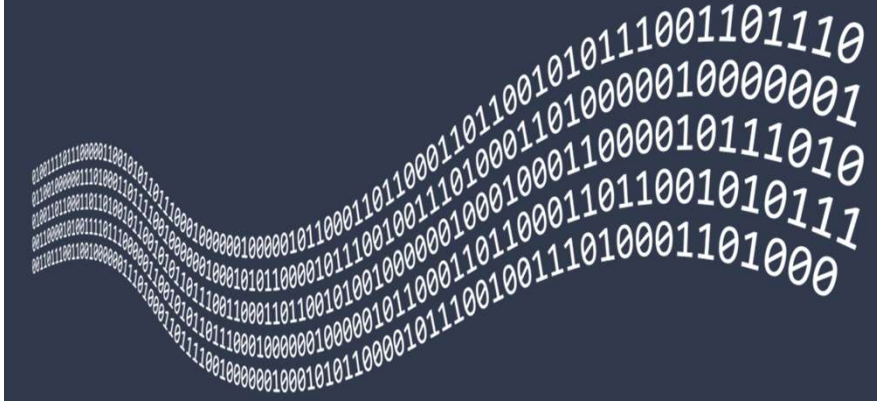
```
public interface Shape {
   double area (double length);
}

public static double triangleArea (double side) {
    return Math.sqrt(3) * Math.pow (side, 2) / 4;
}

public static void main (String[] args) {
   int x = 5;
   int height = 10;

   Shape triangle = ShapeExample::triangleArea;

   System.out.println (triangle.area (x));
}
```

Method reference

The above code is in the `ShapeExample` class which was omitted for space reasons.

- ***Streams*** are a sequence of elements
  - Characters in file
  - Bytes over the network
  - Elements in a List
  - etc.
- Java provides a lot of support for Streams
- In particular, today we are going to look as the `stream` method in the `Collection` interface
  - The `Collection` interface is the base interface for `List`, `Queue`, `Set`, and others (but not `Map`)
- The `Stream<E> stream()` method returns a `Stream` of elements in a `Collection`
- Alone this is not exceptionally useful, but Java has `several methods` that work on streams that will be useful

Streams



Streams are a continuous sequence of information that can be acted on and manipulated.

forEach & Consumer

```
static int sum = 0;
public static void main(String[] args) {
  List<Integer> grades = new ArrayList<>();
  grades.add (54);
  grades.add (85);
  grades.add (97);
  grades.add (72);

  grades.stream().forEach (e -> sum += e);

  grades.stream().forEach
    (System.out::println);

  System.out.println ("Sum = " + sum);
```

`sum` cannot be local due to lambda constraints

Method reference

Use `forEach` to perform some action on each element of `Stream`

- `Stream.forEach` is used to perform some operation on each element in the stream
- It's funny looking parameter (`Consumer <? super T> action`) accepts an instance of the `Consumer` interface
  - `Consumer` is a functional interface
  - I bet you can see where this is going …
- `Consumer`'s only method takes in an element of the stream and performs the action on it
  - We can use lambdas to quickly create simple actions like:
    - Printing the element
    - Performing math on the element
    - Writing it to a file
    - etc.

- Another common stream operation is to ***filter data*** on specific ***criteria*** before doing other work
- The is accomplished using the `Stream.filter` method
- `filter` accepts a single parameter which is an instance of the `Predicate` interface
  - You guessed it, another functional interface
  - It's lambda time!
- `filter` is different from `forEach` in that instead of performing an action, `filter` returns a ***modified*** `Stream`
  - The returned stream only contains the elements that satisfy the `Predicate`
- For this reason, `filter` is often used as part of other `Stream` operations which can be chained together using dot notation

`filter` & `Predicate`

```
public static void main(String[] args) {
  List<Integer> grades = new ArrayList<>();
  grades.add (54);
  grades.add (85);
  grades.add (97);
  grades.add (72);

  // Print only passing grades
  grades.stream().filter(e -> e > 69)
                 .forEach(System.out::println);
}
```

Filter on all the passing grades

Print the filtered stream

`filter` is used to ***reduce*** the amount of data in a stream based on some criteria.

Functional Programming

"Functional programing is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program."

Functional programming is often associated with recursion, though that is not the only way to use it.

- In functional programming, functions are first class citizens
  - They can be assigned to variables
  - Passed as arguments
  - Returned by functions
- In Java this is achieved by creating classes that are a functional interface
  - I.E. They have only a single method
- Pure functions:
  - Are not effected by state
  - Have no side effects (don't change state)
  - Always return the same result with the same arguments
- Since pure functions do not change any state they can be used without any fear of concurrency issues

Why Lambda's

- We often see functional programming associated with lambda's why?
- In Java, lambda's are a shorthand for functional interfaces
    - This is the tool by which Java manages functions as first class citizens
- Lambda's do not allow any local state from a calling function to be changed
- However, you can still modify global state
    - This is a side effect
    - If they modify state, they are not a pure function

Stream Library

Java 8+ has extensive support for streams which use the functional programming paradigm.

A select few of the available methods are detailed on the right.

Functional Programming Term – **Reduction**

The mechanism for executing **functional programs** is **reduction. Reduction** is the process of converting an expression to a simpler form.

- `Stream.filter(predicate)` – removes items from the stream that fail the predicate
  - `String.matches(regex)` – returns true if a string matches the specified regular expression
- `Stream.foreach(action)` – performs the specified action on each element in a stream
  - Terminates the stream
  - Action is often implemented as a lambda
- `Stream.map(mapper)` – applies the mapper 'function' to each element in a stream
  - Intermediate operation
  - Mapper is often implemented as a lambda
- `Stream.collect(collector)` – adds each element in the stream to a collection
  - Terminates the stream
  - May include other reductions on top of adding to the collection
  - Often used with `Collectors`

- `IntStream.range(start, end)` – creates a stream of integers from start to end (non-inclusive)
- `Arrays.stream(array)` – uses array as the source of a stream
- `Arrays.stream(array, start, end)` – uses a portion of an array (end is non-inclusive)
- `Files.lines(path)` – uses a file interpreted as lines as the source of a stream
  - Can use `File.toPath()` to easily get a file's path
- `Collection.stream()` – uses the collection as the source of a stream
- `Collectors.toList()` – accumulates a stream into a List

Stream Extensions

In addition to the stream library. There is support for creating and working with streams in many other libraries.

A few examples are listed to the left.

Java has way more support for Streams than what is mentioned here. If you don't see something try searching for it.