

# LAS3006 – Extreme Java Development

---

## LECTURE 1 – INTRODUCTION TO JAVA 8 (... 11)

# What's the fuss about Java 8?



# The Eco-System is Evolving Continuously

---

- Changing Computing Approaches
  - Web-based Applications using HTML5, CSS3 and JS (JQuery, Angular etc.)
  - Responsive web applications and Mobile / Tablet applications.
  - Cloud – Regional Distribution and on demand Horizontal Scalability
- Higher Expectations
  - More functionality
  - Performance and Response (esp. due to AJAX and native apps)
  - Faster start-up times (esp. due to Microservices)
  - Shorter release cycles
- Other Programming Paradigms
  - Functional Programming
  - Type Inference
  - Dynamic Languages and Dynamic Typing

# Java 8 – A Language and Library Refresh

---

- Reduce verbosity and boiler plate required to do simple things
  - Anonymous class implementations vs Lambda Expressions
  - Explicit iterations vs Stream Operations
- New Programming Paradigms
  - Functional Interfaces
  - Monadic operations
  - Deferred Execution and Lazy Processing
  - Traits (Interfaces with default methods)
- Enhancements to Core Libraries
  - Collections
  - Concurrency
  - Date and Time APIs

# Java 9 – Improvements

---

- Improved support for system modularity (Project Jigsaw)
  - Modules and their dependencies can be specified as part of your JAR.
- Improved security support (DTLS, RSA, PKCS12 keystore, SHA-3).
- Applets and the Java Browser plugin officially deprecated (finally!)
- Garbage collection improvements (G1 is better tuned and now default)
- Improvements to the core libraries:
  - Concurrency (JEP 266)
  - Collections (JEP 269)
  - Reactive Programming interface support (Reactive Streams / Flow API)
- Other goodies (HTTP/2 Client API, Process Management API ...)

***Java 9 was not Long Term Support.***

# Java 10 – Less Boilerplate

---

- Local-variable type inference (with the **var** keyword).
- Improved syntax to create unmodifiable collections and new methods to core libraries.
- Garbage collection improvements to G1 for concurrent collections.
- Security improvements
  - Root certificates in `cacerts` keystore (JEP 319).
  - TLS support enhancements.

***Java 10 was not Long Term Support***

# Java 11 – Cleaning up (and licensing)

---

- Removal and deprecation of various legacy or non SE features:
  - Removed JAX-WS, JAXB, JAF, JTA, CORBA and related tools.
  - Deprecated Nashorn Javascript engine.
- Very minor syntactical improvements
  - Local-variable type inference in lambda expressions.
- New (experimental) garbage collection options:
  - No-Op for high performance or benchmarking requirements.
  - ZGC for low latency GC (pause times  $\leq 10\text{ms}$  even with large heaps).
- Support for TLS 1.3 and new cryptographic algorithms.
- **The licensing model has now changed.**







***Java 11 is the Long Term Support (LTS) release after Java 8***

# Why Functional Programming?

Functional Programming actually came way before OOP, with languages such as Lisp and Scheme. Interest came mostly from academia.

Increase in processing power and horizontal scalability requirements:

- Event-driven / Reactive Programming paradigms

	Purely functional programming language (influenced by Miranda).	Facebook's spam and malware detection engine
	Created for Ericsson as a massively scalable telecoms platform.	RabbitMQ – a very popular message broker
	Developed by Microsoft, strongly influenced by Haskell, Erlang and Scala.	XBox live gamification engine.
	Provides a mix of functional and object oriented concepts and runs on the JVM.	Akka Framework, Apache Spark, Flink, Kafka, Samza.
	Released by Apple in 2014, influenced by Haskell and very similar to Scala.	Replacing Objective-C for development of iOS etc.
	Maintained by JetBrains, functional and OO, runs on the JVM and native (LLVM). <i>A more concise Java, simpler than Scala.</i>	Google has chosen Kotlin for Android.



# Functional Programming Core Concepts

---

- **Functions are first-class citizens.** They can be passed around, and composed together to create more sophisticated behaviour.
- **Higher Order Functions** - functions that can take other functions as parameters and/or return a function as a result.
- **Immutability** – variables can only be assigned to one value. (In pure functional programming there is no concept of state variables.)
- **Lazy evaluation** – postponing the evaluation of a variable to when it is needed, and reusing its computed value for all subsequent evaluations.
- **Monads** – wrapping data types with functionality which can be chained or interleaved in the execution pipeline.
- ***null*** references are not allowed (or highly discouraged).  
<http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>

*FP has its roots from a system of formal logic called **lambda calculus**.*

# FP avoids Side-Effects

---

A **pure function** is one that

1. Always evaluates to the same result when given the same arguments. (That is, it does not depend on any other state information of the system, including external I/O.)
2. Does not modify any other mutable variable in the system as part of the computation needed for its evaluation (including performing I/O). This is known as having **no side-effects**.

In contrast, **methods** in OOP interact with the internal state of an object instance.

**Limiting side-effects** enables better **testing** and **verification** of the **correctness** of a program, and makes it easier to **parallelize** and **distribute** business logic across multiple process instances (**horizontal scaling**).

Of course, a program needs to have side-effects (especially I/O). FP just encourages you to keep them to the minimum necessary.

Side-effect free code can also be subject to compiler optimisations.

# Java Lambda Expressions

---

- A Lambda Expression is a block of code, which can also take parameters.
- A Lambda Expression represents a set of instructions that you want to be executed at a **later point in time**.
- Lambda Expressions have access to the **enclosing scope** within which they are declared, and can access **free variables** which are neither parameters nor local variables within the expression.
- They provide a neater way to attach **deferred execution** such as event handling, without the bloat of inner or anonymous classes.

*A block of code accompanied by free variables bound to some value is also known in other languages as a **closure**. Essentially it represents a function together with the 'context' in which it is to be executed.*

# Syntax of a Lambda Expression

---

A Lambda Expression consists of 3 components:

- The parameters (empty parenthesis if none are needed). The type of each parameter is needed, but can be omitted if it can be inferred.
- The arrow operator `->`
- The expression (or multiple expressions enclosed in a `{ }`, with explicit **return** statements if applicable).

```
Thread thread1 = new Thread(new Runnable() {  
    public void run() {  
        doSomething();  
    }  
});
```

```
Comparator<String> lengthComparator =  
    new Comparator<String>() {  
        public int compare(String s1, String s2) {  
            return Integer.compare(s1.length(),  
                                   s2.length());  
        }  
    };
```

Using Anonymous Classes

```
Thread thread2 = new Thread(() -> doSomething());
```

```
Comparator<String> lengthComparator =  
    (String s1, String s2) ->  
        Integer.compare(s1.length(), s2.length());
```

**or**

```
Comparator<String> lengthComparator = (s1, s2) ->  
    Integer.compare(s1.length(), s2.length());
```

Using Lambda Expressions

# Exercise

---

Create a new class **StudentData** which has 3 member variables, **name**, **surname**, **id**. (Make these fields immutable (**final**) and assign them through the constructor.)

In your **main()** method populate an array with 10 students. Use the same surname for some students and a few with the same name and surname. Use a unique ID.

Sort the student array with **Arrays.sort(T[] a, Comparator<? super T> c)** using a **Lambda Expression** as the **Comparator** argument.

- Sort the students first by surname, if both are the same, then by name, and finally by ID, in alphabetical (ascending) order, and then print the contents of the sorted array.

# Functional Interfaces

---

- Lambda Expressions can only be used when the expected type is a **Functional Interface**.
- A **Functional Interface** is an **Interface** with only **one abstract method**:
  - An interface with more than one abstract method is **not** a functional interface.
  - An abstract class with only one abstract method **cannot** be used either.
  - An interface with **one abstract method** and one or more **default methods** is also a **functional interface**. (More on *default methods* later).
- A Lambda Expression essentially just implements that abstract method.
- You can add the annotation **@FunctionalInterface** to your interface so that the compiler checks that the interface complies with these rules.

```
@FunctionalInterface
public interface EncryptionProvider {
    public String encrypt(EncryptionKey key, String data);
}
```

# Checked Exceptions in Lambda Expressions

---

- The code inside a Lambda Expression may also throw exceptions.
- If the code throws a checked exception, there are two ways the exception can be handled:
  - The abstract method of the Functional Interface must declare the checked exception, such it can be thrown to be handled by the caller.
  - The lambda expression itself can catch it and handle it.
- Unchecked exceptions can be thrown from the Lambda Expression without having to be declared by the abstract method.
  - Of course, the exception still needs to be handled at some point or the thread will stop executing.

# Method References

---

- Often, the code of a Lambda Expression just calls another method.
- **Method references** provide an even more concise way to call single methods as Lambda Expressions, without explicit parameter declaration.
- The parameters of the Lambda Expression are passed onwards to the method reference implicitly (the compiler will infer them).
- Method references are written using the `::` operator to refer to a method of a class or object.

## Full Lambda Syntax

```
Arrays.sort(stringArray, (String s1, String s2) -> s1.compareToIgnoreCase(s2));
```

## Concise Lambda Expression using a Method Reference

```
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

***When using method references use appropriate method names for readability!***



# Kinds of Method References

Kind	Syntax	Example Lambda Expression equivalent
Reference to a static method	<code>Class::staticMethod</code>	<code>String::valueOf (s) -&gt; String.valueOf(s)</code>
Reference to an instance method of a specific object	<code>object::instanceMethod</code>	<code>x::toString () -&gt; x.toString()</code>
Reference to an instance method of an object supplied later (as a parameter)	<code>Class::instanceMethod</code>	<code>String::toString (s) -&gt; s.toString()</code>
Reference to a constructor	<code>Class::new</code>	<code>String::new () -&gt; new String()</code>

# Reference to a Static Method

---

- A **static method reference** refers to a static method of the specified class.
- The method needs to have **the same signature** of the abstract method of the expected functional interface.
- The arguments of the abstract method will be automatically copied to the static method being referenced when it is to be executed.
- Overloaded static methods can also be used, as long as the signature matches.

```
Integer[] numbers = {  
    5, 1, 63, 8, 19, 23, 11  
};  
  
Arrays.sort(numbers, (x, y) -> Integer.compare(x, y));
```

```
Arrays.sort(numbers, Integer::compare);
```

*Equivalent*

# Reference to bound non-static methods

---

- A **bound non-static method reference** is a non-static method that is bound to a specific object instance (*receiver*).
- The method also needs to have the **same signature** of the abstract method of the expected functional interface.
- The arguments of the abstract method will be automatically copied to the non-static method being referenced when it is to be executed.

```
MyIntegerManipulator myIntegerManipulator = new MyIntegerManipulator();  
Arrays.setAll(numbers, (x) -> myIntegerManipulator.square(x));
```

```
Arrays.setAll(numbers, myIntegerManipulator::square);
```

*Equivalent*

Note that the object instance in the outer scope is being accessed inside the lambda expression / method reference (even though the execution could be deferred to later).

The reference to the object instance is known as a **free variable**.

The lambda expression accessing a free variable is known as a **closure**.

# Referring to the Current Instance

---

- You can also refer to a method of the current instance by using the **this** keyword.

```
public void sayHello() {  
    System.out.println("Hello");  
}
```

```
Thread thread = new Thread(this::sayHello);  
thread.start();
```

- You can also refer to a method of the super class of the current instance by using the **super** keyword.
- Same signature matching rules of **bound non-static methods** apply.
- Within an inner class, you can also call:
  - A method of the enclosing class, **EnclosingClass.this::method**
  - A method of the super class of the enclosing class, **EnclosingClass.super::method**(where **EnclosingClass** is the class name of the outer enclosing class).

# Reference to Unbound Non-Static Methods

---

- An **unbound non-static method reference** is a non-static method that is not bound to a specific object instance (*receiver*).
- The type of the *receiver* must be compatible with that of the **first parameter** of the abstract method of the expected functional interface.
- The method needs to have the signature that matches that of the abstract method of the expected functional interface, **without the first parameter**.
- The **second till the last argument** of the abstract method will be automatically copied to the non-static method being referenced.

```
Integer[] numbers = {  
    5, 1, 63, 8, 19, 23  
};
```

```
Arrays.sort(numbers, (x, y) -> x.compareTo(y));
```

```
Arrays.sort(numbers, Integer::compareTo);
```

*Equivalent*

# Constructor References

---

- We can also use constructors just like method references. In this case the method name used will be the special keyword **new**.
- The parameters of the constructor must match those of the abstract method of the expected functional interface.
- The arguments of the abstract method will be automatically copied to the constructor's arguments and a new instance of the class is created.

```
MyClass[] myObjects = new MyClass[20];  
Arrays.setAll(myObjects, (x) -> new MyClass(x));
```

```
Arrays.setAll(myObjects, MyClass::new);
```

*Equivalent*

# Exercise

---

1. Move the previous lambda expression to a separate **static** method in the class **StudentData** named **compareTo()** that takes two parameters of type **StudentData**.
  - Change the lambda expression in the call to **Arrays.sort()** to call the **compareTo()** static method of **StudentData**. Confirm the sorting behaviour is still the same.
2. Change the **compareTo()** method to an instance method, and remove the first **StudentData** parameter (replacing it in the code with a reference to **this**). *(Keep a copy of the static method commented out)*
  - Does the code still compile? Is the sorting behaviour still the same?
3. Put back the static method (so now you have two methods with the same name, one static and one not).
  - Does the code still compile?

# Variable Scopes and Closures

---

- Variables from an enclosing method or class can be accessed within a lambda expression.
- However, a lambda expression is intended to be executed **later**. This raises the issue of what value will the variables have when the lambda expression is executing.
- Furthermore, if the lambda expression is executed after the enclosing method has finished, any local variables accessed from the lambda expression would have been destroyed.
- A lambda expression **captures** the values of any free variables.
- A lambda expression that captures free variables is called a **closure**.
- The body of a lambda expression has the same scope as a nested block. Same rules for naming conflicts and shadowing of variable names apply.
- You can also refer to the enclosing instance through the **this** keyword, and to its superclass through the **super** keyword.



# Closures and Free Variables

---

- In order to ensure such values are well defined, these variables have to be **effectively final** (their value does not change).
- This ensures thread safety.
- The compiler won't check all possible instances of concurrent access (for instance an object can be mutated even if the variable referring to it is final), so you still have to be careful and take care of thread safety.

```
int j = 5;  
Thread t = new Thread(() -> System.out.println(j));
```

```
for (int i = 0; i < 20; i++) {  
    Thread t = new Thread(() -> System.out.println(i));  
}
```

! Error:(86, 54) java: local variables referenced from a lambda expression must be final or effectively final

# Local Variable Type Inference

---

- While Java is still **statically typed**, advancements in type inference algorithms enable the compiler to *infer* the type of a variable from the context within which it is being used.
- Java 10 introduced type inference for local variables, by using the **var** keyword.
- This allows you to reduce boiler plate where the type is pretty obvious.
- You still need to specify the type explicitly in member variables and types where the inferred type is ambiguous.

```
var str = "Hello World";  
  
var list = List.of(5, 4, 3, 2, 1);  
  
var list2 = new ArrayList<String>();
```

# Exercise

---

1. Prompt the user to enter a directory path, and then prompt him to enter a file extension.
2. List the files that are in the specified directory and match the required extension using the **File.listFiles()** method with a **closure** (a lambda expression using the file extension **free variable**). You can just check that the absolute path ends with the extension, and print the absolute path.

Hint: You can use the **Scanner** class to read strings from keyboard input, via the **Scanner.next()** method.