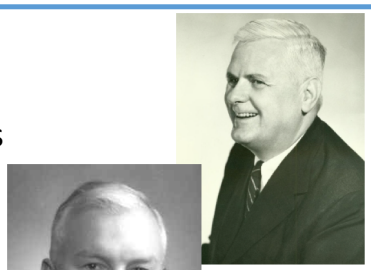
Lambdas and Functional Programming in Java 8

GURUTEAM
specialist ICT learning

Functional Programming

- Type inference
- First-class and higher-order functions
  - lambdas
  - closures
- Immutable state
- Use of recursion
- Declarative style
- Lazy evaluation



Functional Programming in Java

- Functional programming paradigm regaining popularity
  - to help with concurrency
- Many languages have support for FP
- Java support formalized in Java 8

Function Objects

- Encapsulation of operations as data
  - examples from the Java API

```
public interface Runnable {
  public void run()  // Do something
}
```

```
Runnable r = new Runnable() {
   @Override
   public void run() {
      System.out.println("Hello, world);
   }
};
```

Function Objects

- Encapsulation of operations as data
  - examples from the Java API

```
public interface Callable <T> {
  public T call()  // Calculate something
}
```

```
Callable<Integer> c = new Callable<>() {
   @Override
   public Integer call() {
      return java.util.Random.nextInt();
   }
};
```


Function Objects

- Encapsulation of operations as data
  - extending the idea

```
public interface Func1 <A, R> {
  public R apply ( A arg )
}
```

```
Func1<Integer, Integer> f = new Func1<>() {
   @Override
   public Integer apply( Integer i ) {
      return i * 2;
   }
};
```

Functional Interfaces

- Java 8 introduces Functional Interfaces
  - define methods for functional programming
  - annotation for compiler hints

```
package java.util.function

…

@FunctionalInterface
public interface Function<A,R> {
  R apply ( A arg )
  // Other default methods only

}

…
```


Functional Interfaces

- Java 8 introduces Functional Interfaces
  - can have any number of static or default methods
  - must have a *single* abstract method

```
package java.util.function

…

@FunctionalInterface
public interface Function<A,R> {
  R apply ( A arg )
  // Other default methods only

}

…
```

Functional Interfaces

- Multiple Functional Interfaces are defined

```
…
@FunctionalInterface
public interface Predicate<T> {
  boolean test( T arg )
  …
}
@FunctionalInterface
public interface Consumer<T> {
  void accept( T arg )
  …
}
@FunctionalInterface
public interface Supplier<T> {
  T get()
  …
}
…
```


Functional Interfaces

- Multiple Functional Interfaces are defined

```
…
@FunctionalInterface
public interface BiFunction<A,B,R> {
  R apply ( A arg1, B arg2 )
  …
}
@FunctionalInterface
public interface BiPredicate<A,B> {
  boolean test( A arg1, B arg2 )
  …
}
@FunctionalInterface
public interface BinaryOperator<T>
   extends BiFunction<T,T,T> {
  …
}
…
```

Working with Functions

```
public class DoubleIt implements Function<Integer, Integer> {
  @Override
  public Integer apply(Integer t) {
    return t * 2;
  }
}
```


Working with Functions

```
public class DoubleIt implements Function<Integer, Integer> {
  @Override
  public Integer apply(Integer t) {
    return t * 2;
  }
}
```

```
public class SquareIt implements Function<Integer, Integer> {
  @Override
  public Integer apply ( Integer i ) {
    return i * i;

}
```

Working with Functions

```
public class DoubleIt implements Function<Integer, Integer> {
  @Override
  public Integer apply(Integer t) {
    return t * 2;
  }
}
```

```
public class SquareIt implements Function<Integer, Integer> {
  @Override
  public Integer apply ( Integer i ) {
    return i * i;
  }
}
```

```
public class FuncDriver {
  public static int doIt( int n, Function<Integer, Integer> f) {
    return f.apply(n);
  }
  public static void main(String[] args) {
    System.out.println( new DoubleIt().apply(3) );
    System.out.println("---");
    System.out.println( doIt(3, new DoubleIt()) );
    System.out.println( doIt(3, new SquareIt()) );
  }
}
```

```
6
---
6
9
```


Introducing Lambdas

- Lambda expression is a "function literal"
  - more concise syntax for representing functions
- Instance of Functional Interface type
  - as specified by `@FunctionalInterface`

```
( Integer i ) -> i * 2
```

`Function<Integer, Integer>`

Argument

Result

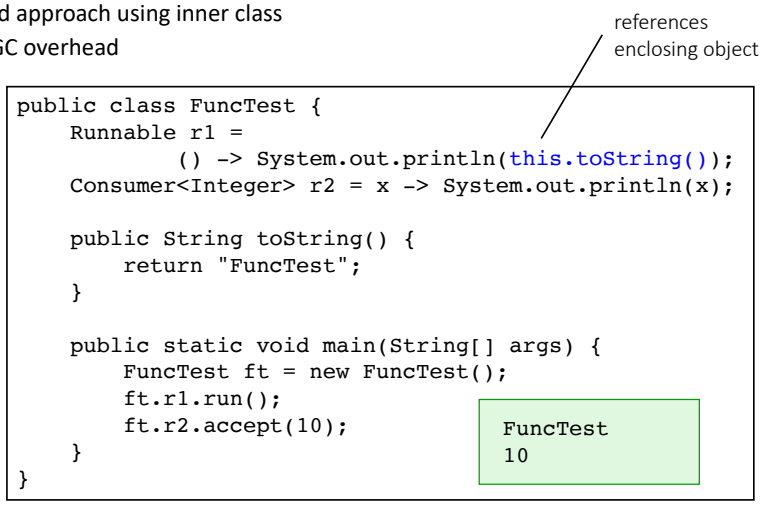
Explicit typing of argument not required here

```
…
System.out.println( doIt(3, i -> i * 2) );
…
```

```
…
Function<Integer, Integer> squareIt = i -> i * i;
System.out.println( doIt(3, squareIt) );
…
```

About Lambdas

- Lambda does not cause new object to be created
  - different from old approach using inner class
  - lower memory/GC overhead
- Lambda has no identity
  - uses context where lambda is defined



Capturing

- Lambda may access variables from its defining scope
  - "capturing" or "closing"
- Local variables must be "effectively final"
  - not changed after definition
  - final keyword not mandatory as for local classes

```
public class LambdaStuff {
  int val1 = 100;
  Runnable r3 = () -> {
     val1 += 1; System.out.println("Value: " + val1);
  } ;
  public static void main ( String [] args ) {
    int val2 = 10;
    Runnable r4 = () -> {
      val2 += 1; // invalid
      System.out.println("Value: " + val2);
    };
  }
}
```

Returning a Function

- Function may be returned by a function/method

```
public static Function<Integer, Integer> multBy ( int n ) {
  return (i -> i * n );
}
```

Must be effectively final

```
public static void main(String[] args) {

  Function<Integer, Integer> twice  = multBy(2);
  Function<Integer, Integer> thrice = multBy(3);

  System.out.println( "twice(10) = " + twice.apply(10) );
  System.out.println( "thrice(10) = " + thrice.apply(10) );

}
```

```
twice(10) = 20
thrice(10) = 30
```


Composing Functions

- Use result of one function as argument to another
  - key functional programming technique
  - given functions 'f' and 'g'
  - f.compose(g) is g followed by f
  - f.andThen(g) is f followed by g
- Support for composition in `Function<A,R>` Interface
  - default methods

```
@FunctionalInterface
public interface Function<A,R> {
  …
  default <V> Function <A, V> compose (
                Function<? super R, ? extends V> after )

  default <V> Function <A, V> andThen(
                Function<? super R, ? extends V> before )
}
…
```

Composing Functions

```
public static void main(String[] args) {

  Function<Integer, Integer> f1 = i -> i + 2;
  Function<Integer, Integer> f2 = i -> i * 3;

  System.out.println("f1(2) = " + f1.apply(2));
  System.out.println("f2(2) = " + f2.apply(2));

  System.out.println("(f1 compose f2)(2) = " +
                          (f1.compose(f2)).apply(2));
  System.out.println("(f1 andThen f2)(2) = " +
                          (f1.andThen(f2)).apply(2));

  Function<Integer, Integer> g = f1.compose(f2);
  System.out.println("g(2) = " + g.apply(2));

}
```

```
f1(2) = 4
f2(2) = 6
(f1 compose f2)(2) = 8
(f1 andThen f2)(2) = 12
g(2) = 8
```


Defining Lambdas

Lambdas can be defined in any of

- Variable Declaration
- Assignment
- Return statement
- Array Initializer
- Method or Constructor args
- Lambda body
- Ternary Condition Operator
- Cast Expression

Method References

- Allows constructors and methods to be referenced
  - without executing them
  - can then execute them at a future point in time if required
- Used to type existing method as Functional Interface
- Four Types of Method Reference
  - reference to a static method
  - reference to a constructor
  - reference to a method on a specific instance
  - reference to a method on an object of a particular type
- Arguments cannot be passed to Method References

Method References

- Given a type

```
class Person {
	private static int count = 0;

	private String name = "";

	public static int increment() {
		count = count + 1;
		return count;
	}

	public Person(String name) {
		this.name = name;
	}

	public String getName() {
	 return name;
    }
}
```

Method References

- Reference to a static method
  - Person::increment
- Reference to an instance method of a particular object
  - p::getName
- Reference to an instance method of an arbitrary object of a particular type
  - Person::getName
- Reference to a constructor
  - ClassName::new

Method References

```
import java.util.function.*;
public class MethodReferenceApp {
    public static void main(String[] args) {
        Person p = new Person("Bob");
        // Static Ref
        Supplier<Integer> staticRef = Person::increment;
        System.out.println(staticRef.get());
        // Constructor Ref
       Function<String,Person> consRef = Person::new;
       Person p2 = consRef.apply("Jane");
       System.out.println(p2);
       // Specific Instance
       Supplier<String> objRef = p::getName;
       System.out.println(objRef.get());
       // Any instance
       Function<Person,String> anyRef = Person::getName;
       System.out.println(anyRef.apply(p2));
       }
}
```