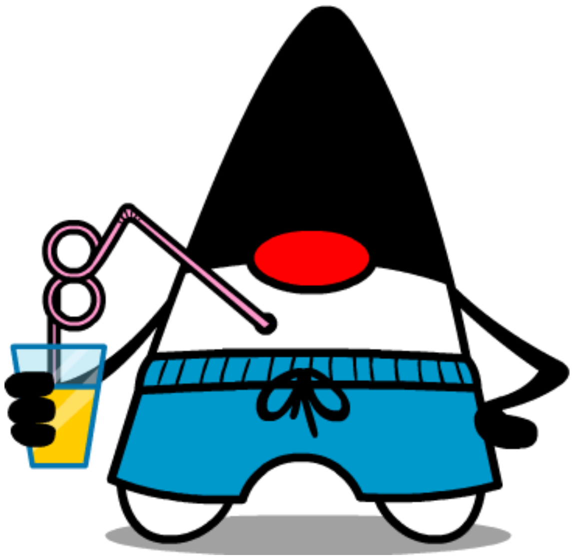
Functional Programming in Java

Lambdas

Functional Programming in Java

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

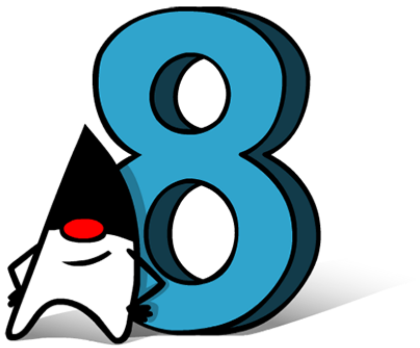
```
package java.util.function

…

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

}

…
```



Functional Interfaces

- Other 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

- Other 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;
  }
}
```

```
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

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

Explicit typing of argument not required here

```
…
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

No return value.
Use {…} to denote body of the lambda

```
public class LambdaStuff {

  Runnable r1 = () -> { System.out.println(this); };
  Runnable r2 = () -> { System.out.println(toString()); };

  @Override
  public String toString() { return "LambdaStuff"; }

  public static void main ( String [] args ) {
    new LambdaStuff().r1.run();
    new LambdaStuff().r2.run();
  }
}
```

```
LambdaStuff
LambdaStuff
```

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: " + value);
  } ;
  public static void main ( String [] args ) {
    int val2 = 10;
    Runnable r4 = () -> {
      val2 += 1; 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

```
(f o g) (x)  =>  f ( g(x) )
```

- 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 <V, R> compose (
               Function<? super V, ? extends T> 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("(f compose g)(2) = " +
                          (f1.compose(f2)).apply(2));
  System.out.println("(f andThen g)(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
(f compose g)(2) = 8
(f andThen g)(2) = 12
g(2) = 8
```

Optional<T>

- Representation of "no value"
  - Map lookup
  - empty collection

- Alternative to null
  - reduces likelihood of NullPointerException
  - reduces need for special handling of results

- Type advertises the possibility of no value

Example – Map lookup

```
…
 Map<String, String> capitals = new HashMap<>();
 capitals.put("UK", "London");
 capitals.put("USA", "Washington DC");
 capitals.put("India", "New Delhi");

 Map<String, Integer> populations = new HashMap<>();
 populations.put("London", 8_000_000);
 populations.put("Washington DC", 6_000_000);
 populations.put("Beijing", 20_000_000);
…
```

```
…
 System.out.println(capitals.get("UK").toUpperCase());
…
 System.out.println(capitals.get("China").toUpperCase());
…
```

```
LONDON
Exception in thread "main"
java.lang.NullPointerException
        at Optional1.main(Optional1.java:34)
```

Example – Map lookup

- Change retrieval of element from Map
  - do not return null, but give indication of "no value"
  - exception not appropriate

```
…
  public static Optional<String> getCap ( String key ) {
    return Optional.ofNullable(capitals.get(key));
  }
…
```

- `Optional<String>` type has two possibilities
  - a `String` value
  - `empty`

```
…
 System.out.println(getCap("UK"));
 System.out.println(getCap("China"));
…
```

```
Optional[London]
Optional.empty
```

Working with the `Optional<T>` Value

- Similar approach to processing Streams
  - no special handling for `empty` case

```
…
 System.out.println(getCap("UK").map(String::toUpperCase));
 System.out.println(getCap("China").map(String::toUpperCase));
…
```

```
Optional[LONDON]
Optional.empty
```

- Value can be extracted

```
…
System.out.println(getCap("UK").map(String::toUpperCase)
                               .orElse(""));
System.out.println(getCap("China").map(String::toUpperCase)
                               .orElse(""));
…
```

```
LONDON
```

Working with the Optional<T> Value

- Chain methods returning Optional<T> with flatMap()

```
…
 public static Optional<Integer> getPop ( String key ) {
  return Optional.ofNullable( populations.get(key) );
 }
…
```

```
…
 System.out.println( getCap("UK")
                        .flatMap(Optional1::getPop)
                        .orElse(-1));

 System.out.println( getCap("China")
                        .flatMap(Optional1::getPop)
                        .orElse(-1));
…
```

Optional<String>

Optional<Integer>

```
8000000
-1
```