Functional Programming

In Java 8

Why?

- Facilitates parallelism
- Being able to pass behaviors as well as data to functions

- Fundamentally different views on data and behavior
  - Physical (holistic)
  - Mathematical
- Mutatability of data

Example 1:
Print a list of integers with a lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> System.out.println(x));
```

- x -> System.out.println(x) is a lambda expression that defines an anonymous function with one parameter named x of type Integer

Example 2:
A multiline lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
   x += 2;
   System.out.println(x);
});
```

- Braces are needed to enclose a multiline body in a lambda expression.

Example 3:
A lambda with a defined local variable

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
   int y = x * 2;
   System.out.println(y);
});
```

- Just as with ordinary functions, you can define local variables inside the body of a lambda expression

Example 4:
A lambda with a declared parameter type

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach((Integer x -> {
   x += 2;
   System.out.println(x);
});
```

- You can, if you wish, specify the parameter type.

Delayed Execution

```
SystemInfo veryExpensiveOperation(){…}

WriteLog(SystemInfo sysInfo) {
        if (getLoggingLevel() == LoggingLevel.Info) {
                logger.Write(sysInfo);
        }
}

WriteLog(veryExpensiveOperation());
WriteLog(()->veryExpensiveOperation());
```

Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function
- It then calls the generated function
- For example, `x -> System.out.println(x)` could be converted into a generated static function

```
public static void lambda$1(Integer x) {
    System.out.println(x);
}
```

- But what type should be generated for this function? How should it be called? What class should it go in?

Functional Interfaces

- Design decision: Java 8 lambdas are assigned to functional interfaces.
- A functional interface is a Java interface with exactly one non-default method. E.g.,

```
public interface Consumer<T> {
   void accept(T t);
   default Consumer<T> andThen(Consumer<? super T> after);
}
```

- The package `java.util.function` defines many new useful functional interfaces.

Assigning a Lambda to a Local Variable

```
void forEach(Consumer<Integer> lambda {
  for (Integer i:items) {
    lambda.accept(i);
  }
}
List<Integer> ints = Arrrays.asList(1,2,3);

Consumer<Integer> fi = x -> System.out.println(x);
ints.forEach(fi);
```

Behind the Scene

```
class Demo {
        public void foo() {
                List<Integer> list = ...
                list.forEach( i -> { System.out.println(i); } );
        }
}
```

```
class Demo {
        public void foo() {
                List<Integer> list = ...
                list.forEach( lambda$1 as Consumer < Integer > );
        }
        private static void lambda$1(Integer i) {
                System.out.println(i);
        }
}
```

Further Behind the Scene

```
public void foo() {
       List<Integer> list = ...
       Consumer<Integer> lambda = invokedynamic#bootstrapLambda,#lambda$1;
       list.forEach( lambda );
}
```

```
public void foo() {
       List<Integer> list = ...
       Consumer<Integer> lambda;
       //call site is cached (not shown here)
       CallSite cs = bootstrapLambda(
              MethodHandles.lookup(),
              "lambda$1",
              MethodType.methodType(void.class, Integer.class));
       lambda = (Consumer < Integer > ) cs.getTarget().invokeExact();
       list.forEach( lambda );
}
```

Further Behind the Scene

```
//lookup = provided by VM, name = "lambda$1", provided by VM, type = Integer (void)
private static CallSite bootstrapLambda(Lookup lookup, String name, MethodType type) {
    MethodHandle functionPointer = lookup.findStatic(lookup.lookupClass(), name, type);
    return LambdaMetafactory.metafactory(lookup,
        "accept",
        //signature of lambda factory
        MethodType.methodType(Consumer.class),
        //signature of method Consumer.accept after type erasure
        MethodType.methodType(void.class, Object.class),
        //reference to method with lambda body
        functionPointer,
        type);
}
```

Variable Capture

- Lambdas can interact with variables defined outside the body of the lambda

- Using these variables is called variable capture

Variable Capture Example

```
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(1,2,3);
    int var = 10;
    ints.forEach(x -> System.out.println(x + var));
}
```

- Note: local variables used inside the body of a lambda must be final or effectively final

Behind the Scene

```
class Demo1 {
    public void foo() {
        List<Integer> list = ...
        final int low = ..., high = ...;
        list.removeIf( i -> (i >= low && i <= high) );
    }
}
```

```
class Demo1 {
    public void foo() {
        List<Integer> list = ...
        final int low = ..., high = ...;
        list.removeIf( lambda$1 as Predicate capturing (low, high) );
    }

    static boolean lambda$1(int low, int high, Integer i) {
        return (i >= low && i <= high;
    }
}
```

Conciseness with Method References

We can rewrite the statement

```
intSeq.forEach(x -> System.out.println(x));
```

more concisely using a method reference

```
intSeq.forEach(System.out::println);
```

Types of Method References

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

```
intList.stream()
.map(i -> Integer.toHexString(i))
.map(s -> s. toUpperCase())
.forEach(s->System.out.println(s));
```

Even works for multiple arguments as long as they are used in the same order

```
(a,b) -> Integer.sum(a,b)
(a,b) -> a.concat(b)
```

Thinking in Stream

- Given a list of numbers, double the even numbers, and sum
- List<Integer> numbers = Arrays.asList(1,…,10);
- Traditional Implementation

```
        For(int i = 0; i < numbers.size(); i++){
        }
```

- numbers.stream()
- numbers.parallelStream()

Thinking in Stream

- Data running down pipeline
  - The surprising order of processing

```
Stream.of(“a", “b", “c", “d", “e")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

- Operation applied at each step
- ‘swim lanes’

Interesting Properties

- Lazy evaluation
- Disposable: no reuse
- Difficulty error handling

Reactive Programming

- Streams on steroid
- Publisher
    - subscribe(Subscriber sub);
- Subscriber
    - onNext()
    - onError()
    - onComplete()