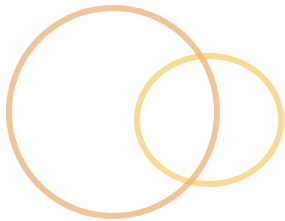
Java Lambda Expressions

Behavior variables for the 21st century

Objectives



- Outline the purpose of lambda expressions
- Read and write lambda expressions in Java

Why Lambdas?

- Programs benefit from flexible behavior; e.g:
  - Sort these using this ordering policy
  - Remove elements unless they match this criterion
- Historically, “variable behavior” has different approaches:
  - Pointer to function (e.g. C, C++)
  - Interpretable source code (e.g. SQL)
  - Object implementing an interface (e.g. Java)
  - Anonymous inner classes (e.g. Java)
  - “Code as data” or “First Class” functions (e.g. Lisp, functional languages, Java 8 Lambdas)

What Are Lambdas?

- In essence a Java lambda behaves like a pointer to a function
- Java is object oriented, and statically type-safe, so lambdas are essentially code that compiles to a pointers to a object that implements a single-method interface
  - With a lot less textual “clutter”

From Classes to Lambdas

```
public class Filter {
  public static <E> void filterList(List<E> data,
                                    Test<E> t) {
    Iterator<E> iterator = data.iterator();
    while (iterator.hasNext()) {
      if (!t.test(iterator.next()))
        iterator.remove();
      }
  }
```

From Classes to Lambdas

```
public static void main(String [] args) {
    List<String> ls = new LinkedList<>();
    ls.addAll(Arrays.asList(
      "Alice", "Bob", "Maverick", "Trent"));
    System.out.println("Before: " + ls);
    filterList(ls, new LongerThan5());
    System.out.println("After: " + ls);
  }
}
```

From Classes to Lambdas

```
public interface Test<E> {
    boolean test(E e);
}
```

Note: only one method is declared in this interface

```
public class LongerThan5 implements Test<String> {
    @Override
    public boolean test(String s) {
        return s.length() > 5;
    }
}
```

From Classes to Lambdas

- Previously:

```
public class LongerThan5 implements Test<String> {
    @Override
    public boolean test(String s) { // ->
        return s.length() > 5;
    }
}
```

- Becomes:

```
(s) -> s.length() < 5
```

Lambda General Format

- Lambdas provide behavior that implements a method in an interface
  - Commonly a generic interface
- Syntax defines:
  - Argument list
  - ->
  - Behavior
-

Type Inference

- Java compiler attempts to decide the type of the lambda based on the context
  - Lambda defined in method argument must satisfy the requirements of the method
    - Overloaded methods can cause ambiguity
  - Lambda defined in initialization of variable must satisfy the type of the variable
  - Lambdas frequently are used to implement generic types, in which case the generic type variables are inferred from the context too
- Inference isn't always possible

Lambda Argument Syntax

- Argument list is generally enclosed in parentheses
  - Types do not generally need to be specified

    ```
    (s,t,u) -> s + t / u
    ```
  - Types may be specified for the ***entire*** argument list to resolve type inference ambiguity

    ```
    (long s, long t, int u) -> s + t / u
    ```
  - Zero arguments use empty paretheses

    ```
    () -> (int)(Math.random() * 1000)
    ```
  - Single argument situations allow the parentheses to be dropped

    ```
    s -> System.out.println("Value is " + s)
    ```

Expression Lambda Syntax

- Simple lambdas may be expressed using a single expression to the right of `->`
  - These are called “Expression Lambdas”
  - Note that no semicolon is used to terminate the expression

```
(s)  -> s * 2
```

Complex Lambda Behavior Syntax

- For more complex lambda behavior, a block may be used

```
(s,t) -> {
  int rv = 0;
  for (int i = s; i < t; i++) {
    rv += i;
  }
  return rv;
}
```

- If a traditional-form method would require a return statement, then the block form lambda requires a return statement too

Lambdas And Closure

- Lambda expressions can refer to variables in enclosing scopes provided their lifetimes are suitable
- Rules are as variable access in inner classes
  - Fields of enclosing class or object are accessible
  - Method locals must be “effectively `final`”
- The design merits of this technique are highly debatable
  - Sometimes, very good, other times less desirable
  - As a general guide, try to avoid “side effects”

Functional Interfaces

- Lambdas can only be used to implement interfaces that have a single abstract method
- Such interfaces are called “Functional Interfaces”
- The annotation `@FunctionalInterface` tells the compiler to verify that this interface defines exactly one abstract method
  - `@FunctionalInterface` is not required to allow use in a lambda; the annotation only serves to warn if we accidentally created more than one abstract method

Java 8 API Functional Interfaces

- Package `java.util.function` defines many functional interfaces

Interface	Method
`Predicate<T>`	`boolean test(T t)`
`Supplier<T>`	`T get()`
`Consumer<T>`	`void accept(T t)`
`Function<T,R>`	`R apply(T t)`
`BiFunction<T,U,R>`	`R apply(T t, U u)`

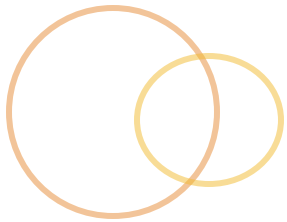
This is the “right” interface for the “Test” defined in the earlier example

- And many more…

Functional Interfaces And Primitives

- Most functional interfaces are generic, e.g. `BiFunction<T,U,R>` defines `R apply(T t, U u)`
- But, generics are incompatible with primitive data types
- So, several functional interfaces are defined for primitive types, `int`, `long`, and `double`
  - But `float` is generally ignored; use `double` instead, or define your own interface

Method References



- Method references allow pre-existing methods to be conveniently used where a lambda expression would be applicable

Method Reference Example

```
public static <E> E executeBinaryOp(
    E e1, E e2, BinaryOperator<E> op)
{
  return op.apply(e1, e2);
}
```

```
executeBinaryOp("Jim", " Jones", (s,t)->s.concat(t))
```

```
executeBinaryOp("Jim", " Jones", String::concat)
```

Method Reference Invocation

- Methods for references can be instance ***or*** static
- The interface `BinaryOperator<T>` requires two arguments (and returns a single value)
- Suppose the arguments are `s` and `t`
- The method reference `String::concat` will cause invocation as
  `s.concat(t)`
- However, a static method
  `String joinStrings(String a, String b)`
  will be invoked as `joinStrings(s, t)`

Method References Afterthought

- The use of method references effectively allows any arbitrary method to be used to implement a functional interface, without the original method or its defining class knowing anything about that interface
  - This is an aspect of “duck typing” in Java, although it happens at compile time and only relates to functional interface behavior, not to objects as a whole

Lab Exercise

- Create a `Customer` class representing a customer of our retail outlet. Give the customer a name, a credit limit, a credit balance, and optionally a `Set<String>` representing the items this customer has purchased from us
- Give the customer a `toString` method to allow easy textual representation
- In a `main` method create a `List<Customer>` with a few sample customers in it
- Print the list out

Lab Exercise

- Write a generic filter method, similar to the one in the example, that takes a list and “filter behavior” and creates a new list that contains only the items that pass the test of the filter
- Think about what interface the “filter behavior” should implement? Use the standard Java 8 interfaces

Lab Exercise

- Arrange for the main method to filter the list according to several criteria, printing the list each time
- Suggested criteria are:
  - Customers who have a credit balance greater than 1000
  - Customers who have a credit balance greater than their credit limit
  - Customers whose names begin with a particular letter
  - Customers who buy a particular item
- Optional: implement one of these criteria without using lambdas