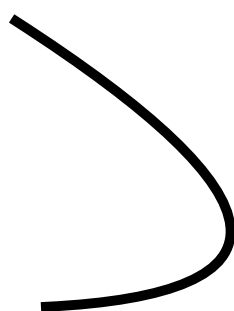




# Inner Class Access

- An inner class (ex: anon observer class) can access..  
all variables & methods visible in the enclosing scope
  - Including:
    - Local variables & parameters;  
must be final (or effectively final)
    - Fields & methods of containing object.
    - Fields & methods of inner class
  - How?
    - Inner class automatically creates a reference  
to containing object and needed local variables.
- 

# Inner Class and Final Local Variables

- Why can inner class access only final local variables?
  - *inner class runs much later than containing function*
  - So **parameters and local variables no longer exist.** **But, Java makes copy of needed local variables/parameters.**
    - Called *capturing variables*
  - If variable not final, Java does not know which value to capture.
- Effectively Final
  - Detects if a variable..  
*is not changed after initialized*
  - Effectively final OK for capturing variable.

```
void foo(int x) {  
    // Don't change x!  
    // x = 42;  
    model.addObserver(new Observer() {  
        @Override  
        public void event() {  
            System.out.println("VAL: " + x);  
        }  
    });  
}
```

*if initialized x then change x —> error later*  
*Observer is interface*

# Lambda Expression

- Awkward to create anon classes for small interfaces
  - Lambda expressions can be used instead when..  
interface has only one method

Use an anon-inner class:

```
void foo(int x) {  
    myModel.addObserver(new DaObserver() {  
        @Override  
        public void dataChanged(int newVal) {  
            System.out.println(newVal);  
        }  
    });  
}
```

Use a lambda expression:

Syntax:  
arg -> statement

```
void foo() {  
    myModel.addObserver(  
        (newVal) -> System.out.println(newVal)  
    );  
}
```

# Lambda Notes

- **Compactness**
  - **Functional interface**: an interface with one method
  - Prefer lambda expressions over anonymous classes for functional interfaces: they are way shorter!
- **Clarity**
  - Lambdas **don't state the type of their argument**:  
So, name parameters carefully
  - Don't express long operations as lambdas.

# Method References

- Situation

You are using a lambda expression to just call a function, passing along all the parameters

```
obj.register( x -> procssEvent(x) );
```

- Solution

Use a method reference:

[reference the method name without arguments or lambda](#)

```
obj.register( this::procssEvent );
```

# Method Reference

```
class Client {  
    public void regObserver() {  
        Model model = new Model();
```

```
        // Option 1: Anonymous Class  
        model.addObserver(new Observer() {  
            @Override  
            public void event(String description) {  
                handleEvent(description);  
            }  
        });
```

```
        // Option 2: Lambda  
        model.addObserver(msg -> handleEvent(msg));
```

```
        // Option 3: Method Reference  
        model.addObserver(this::handleEvent);
```

```
    }
```

```
    private void handleEvent(String description) {  
        System.out.println(description);  
    }
```

```
}
```

```
interface Observer {  
    void event(String description);  
}  
class Model {  
    public void addObserver(Observer obs) {  
        // ...  
    }  
}
```

# Streams



# Streams

- Stream
  - a sequence of objects supporting a pipeline of operations
  - *Think of it like a parade.*
- Stream Pipeline
  - Combine stream operations to process elements in a stream
  - Can be done in parallel
  - *Think of it like doing something to each vehicle in a parade.*
- Example

```
List<Double> heights_m =  
    Arrays.asList(10.0, 30.0, 20.0, 60.0, 50.0);  
  
heights_m.stream()  
    .map(x -> x * x)  
    .forEach(x -> System.out.println(x));
```

```
100.0  
900.0  
400.0  
3600.0  
2500.0
```

# Pipelines

- About Streams
  - Streams process elements of a collection (or generating func.)
  - Streams don't change the data structure, they operate on the elements
- Stream Pipeline built out of:
  - source stream: Provides the stream
  - intermediate operations: Operates on stream; returns stream
  - terminal operation: Collects result as desired for return

```
List<Double> heights_m =  
    Arrays.asList(10.0, 30.0, 20.0, 60.0, 50.0);  
  
heights_m.stream()  
    .map(x -> x * x)  
    .forEach(x -> System.out.println(x));
```

# Examples

<pre>class Student     implements Comparable&lt;Student&gt; {     private String name;     private double gpa;     private int creditHours;     // ... constructor, getters,     //     compareTo(), }  List&lt;Student&gt; students =     Arrays.asList(         new Student("Bill", 1.68, 52),         new Student("Alice", 3.5, 40),         new Student("Doris", 4.01, 102),         new Student("Charlie", 3.8, 12)     );</pre>	<pre>// <b>Terminal Operation: forEach</b> // ( Assume sout() is System.out.println(...) ) students.stream()     .forEach(std -&gt; sout(std.getName()));  // <b>Intermediate Operation: filter</b> students.stream()     .filter(std -&gt; std.getGpa() &gt;= 3.5)     .forEach(std -&gt; sout(std.getName()));  // <b>Terminal Operation: count</b> long numFailing = students.stream()     .filter(std -&gt; std.getGpa() &lt; 1.0)     .count();  // <b>Terminal Operation: collect</b> List&lt;Student&gt; honourRoll = students.stream()     .filter(std -&gt; std.getGpa() &gt;= 3.5)     .collect(Collectors.toList());  List&lt;Student&gt; studentsWithL = students.stream()     .filter(std -&gt; std.getName().contains("l"))     .collect(Collectors.toList());  // <b>Intermediate Operation: sorted()</b> List&lt;Student&gt; sorted = students.stream()     .sorted()     .collect(Collectors.toList());</pre>
---	--

# Pipeline Operations

- **Intermediate Operations**

- **filter**: Keep only wanted elements  
`int n = students.stream().filter(std -> std.getGpa() >= 3.5).count();`
- **sorted**: Reorder stream elements  
`List<Student> sorted =  
students.stream().sorted().collect(Collectors.toList());`
- **map**: Apply a transformation to each element (later)

- **Terminal Operations**

- **count()**: # elements  
`int num = students.stream().count();`
- **collect()**: To a type  
`List<Student> sts = stds.stream().filter(...).collect(Collectors.toList());`
- **forEach()**: Do on each  
`stds.stream().filter(...).forEach(s -> System.out.println(s.getName()));`

# Examples: Map & Reduce

*// Map (intermediate) - Transform value or type*

*// Map to change the value*

```
List<Double> heights_m = Arrays.asList(10.0, 30.0, 20.0, 60.0, 50.0);  
final double INCHES_PER_M = 39.3701;  
List<Double> heights_inch = heights_m.stream()  
    .map( m -> m * INCHES_PER_M)  
    .collect(Collectors.toList());
```

*// Map to change the type*

```
List<String> honourRoleNames = students.stream()  
    .filter(std -> std.getGpa() >= 3.5)  
    .map(std -> std.getName())  
    .collect(Collectors.toList());
```

*// Reduce (terminal) - Combine elements*

*// Takes stream of type Z and returns one element of type Z*

```
String message = "Student names: " + students.stream()  
    .map(std -> std.getName())  
    .reduce("", (ans, name) -> ans + "," + name);
```

*// This is the same as:*

```
String messageJoin = "Student names: " + students.stream()  
    .map(std -> std.getName())  
    .collect(Collectors.joining(", "));
```

# int, long, double Streams

- Streams operate on a sequence of objects
  - IntStream/LongStream/DoubleStream operate on *a sequence of primitives*

```
// mapToInt() and sum()
int allHours = students.stream()
    .mapToInt( std -> std.getCreditHours() )
    .sum();

// Max/Min/Avg may fail if stream is empty, so they return an Optional
// Must call "orElse()" on the optional to get a default value
double maxGpa = students.stream()
    .mapToDouble( std -> std.getGpa() )
    .max().orElse(0);

// IntStream.range() to generate a stream
int sumEvens = IntStream.range(0, 100)
    .filter(x -> x % 2 == 0)
    .sum();
```

# Stream Tips

- Stream API is fluent
  - Each intermediate stream operation returns a stream, so can chain operations together in one statement.
- Use streams when they simplify your code
  - Overuse makes code very hard to read
  - Use helper methods to simplify your code and add semantic value to otherwise complex statements
- Naming functions
  - Plural name for functions which return a stream: `students()`, `courses()`,...

# Summary

- Inner classes can access effectively final local variables.
- Lambda expressions replace most anonymous classes.
- Method references replace some lambdas
- Streams and stream pipelines replace some iteration:
  - intermediate operations transform/filter elements
  - terminal operations collect elements at end
- Fluent API: Functions return same type of object as their class (allows chaining)