Lambdas & Streams
Inner Class Access

• An inner class (ex: anon observer class) can access..

• Including:
  – Local variables & parameters;
  – Fields & methods of containing object.
  – Fields & methods of inner class

• How?
  – Inner class automatically..
    to containing object and needed local variables.
Why can inner class access only final local variables?
  - So parameters and local variables no longer exist. But, Java makes copy of needed local variables/parameters.
  - Called..
  - If variable not final, Java does not know which value to capture.

Effectively Final
  - Detects if a variable..
  - Effectively final OK for capturing variable.

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

- Awkward to create anon classes for small interfaces
  - Lambda expressions can be used instead when...

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

Use an anon-inner class:

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

Use a lambda expression:

```java
void foo() {
    myModel.addObserver(
    );
}
```

Syntax: `arg -> statement`
Lambda Notes

- **Compactness**
  - Functional interface:
  - Prefer lambda expressions over anonymous classes for functional interfaces: they are way shorter!

- **Clarity**
  - Lambdas don’t state the type of their argument: So,..
  - 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
  ```java
  obj.register(x -> procssEvent(x));
  ```

- Solution
  Use a method reference:
  ```java
  obj.register( this::procssEvent );
  ```
Method Reference

```java
class Client {
    public void registerObserver() {
        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
  - ..
  - *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

```java
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:
  – Provides the stream
  – Operates on stream; returns stream
  – Collects result as desired for return

```java
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

```java
class Student implements Comparable<Student> {
    private String name;
    private double gpa;
    private int creditHours;
    // ... constructor, getters, // compareTo(),
}

List<Student> 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)
);

// Terminal Operation: forEach
// (Assume sout() is System.out.println(...))
students.stream()
    .forEach(std -> sout(std.getName()));

// Intermediate Operation: filter
students.stream()
    .filter(std -> std.getGpa() >= 3.5)
    .forEach(std -> sout(std.getName()));

// Terminal Operation: count
long numFailing = students.stream()
    .filter(std -> std.getGpa() < 1.0)
    .count();

// Terminal Operation: collect
List<Student> honourRoll = students.stream()
    .filter(std -> std.getGpa() >= 3.5)
    .collect(Collectors.toList());
List<Student> studentsWithL = students.stream()
    .filter(std -> std.getName().contains("l"))
    .collect(Collectors.toList());

// Intermediate Operation: sorted()
List<Student> sorted = students.stream()
    .sorted()
    .collect(Collectors.toList());
```
Pipeline Operations

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

- Terminal Operations
  - count(): # elements
    ```java
    int num = students.stream().count();
    ```
  - collect(): To a type
    ```java
    List<Student> sts = stds.stream().filter(...).collect(Collectors.toList());
    ```
  - forEach(): Do on each
    ```java
    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

```java
// 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

- 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)