Principles of Software Construction: Objects, Design, and Concurrency

Lambdas and Streams in Java 8

Jonathan Aldrich    Charlie Garrod
Administrivia

- Homework 6 checkpoint due tonight
- Homework 6 due Thursday
- Review session Sunday noon-3pm in DH 1212
- Final exam Monday at 8:30am in Porter Hall 100 & 125C
Today’s Lecture: Learning Goals

• Understand the syntax, semantics, and typechecking of lambdas in Java
• Write code effectively with lambdas in Java
• Use the Java stream library both sequentially and in parallel
• Use default methods to put reusable code in Java interfaces
Recall Anonymous Inner Classes

```java
final String name = "Charlie";
Runnable greeter = new Runnable() {
    public void run() {
        System.out.println("Hi " + name);
    }
};

// add functionality to the step button.
step.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        worldPanel.step();
    }
});
```

• A lot of boilerplate for 1 line of code in each example!
final String name = "Charlie";
Runnable greeter = new Runnable() {
    public void run() {
        System.out.println("Hi " + name);
    }
};

// with Lambdas, can rewrite the code above like this
String name = "Charlie";
Runnable greeter = () -> System.out.println("Hi " + name);

The function can be assigned to a Runnable, because it has the same signature as run()

We use a lambda expression to define a function that takes no arguments

The name variable is used in the function; need not be final, but must be effectively final
Effectively Final Variables

```java
final String name = "Charlie";
Runnable greeter = new Runnable() {
    public void run() {
        System.out.println("Hi " + name);
    }
};

// with Lambdas, can rewrite the code above like this
String name = "Charlie";
Runnable greeter = () -> System.out.println("Hi " + name);
```

Lambdas can use local variables in outer scopes only if they are effectively final. A variable is **effectively final** if it can be made final without introducing a compilation error. This facilitates using lambdas for concurrency, and avoids problems with lambdas outliving their surrounding scope.
Replacing For Loops with Lambdas

// Java 7 code to print an array
List<Integer> intList = Arrays.asList(1,2,3);
for (Integer i in intList)
    System.out.println(i)

// Java 8 provides a forEach method to do the same thing...
intList.forEach(new Consumer<Integer>() {
    public void accept(Integer i) {
        System.out.println(i);
    }
});

// Java 8’s Lambda’s make forEach beautiful
intList.forEach((Integer i) -> System.out.println(i));
intList.forEach(i -> System.out.println(i));

This lambda expression takes one argument, i, of type Integer

Even cleaner...since intList.forEach() takes a Consumer<Integer>, Java infers that i’s type is Integer

Example adapted from Alfred V. Aho
Lambda Syntax Options

• Lambda Syntax

  (parameters) -> expression

  or  (parameters) -> { statements; }

• Details

  • Parameter types may be inferred (all or none)
  • Parentheses may be omitted for a single inferred-type parameter

• Examples

  (int x, int y) -> x + y       // takes two integers and returns their sum
  (x, y) -> x - y              // takes two numbers and returns their difference
  () -> 42                    // takes no values and returns 42
  (String s) -> System.out.println(s)  // takes a string, prints its value
  x -> 2 * x                  // takes a number and returns the result of doubling it
  c -> { int s = c.size(); c.clear(); return s; }   // takes a collection, clears it, and returns its previous size
Functional Interfaces

• There are no function types in Java
• Instead, Java has **Functional Interfaces**
  - interfaces with only one explicitly declared abstract method
    - methods inherited from `Object`, like `equals()`, don’t count
    - Optionally annotated with `@FunctionalInterface`
      - Helps catch errors if you intend to write a functional interface but don’t

• Some Functional Interfaces

```
java.lang.Runnable:   void run()
java.util.function.Consumer<T>:   void accept(T t)
java.util.concurrent.Callable<V>:   V call()
java.util.function.Function<T,R>:   R apply(T t)
java.util.Comparator<T>:   int compare(T o1, T o2)
java.awt.event.ActionListener:   void actionPerformed(ActionEvent e)
```

• There are many more, especially in package `java.util.function`
A lambda expression must match its *expected type*
- The type of the variable to which it is assigned or passed

```java
intList.forEach(i -> System.out.println(i));
```

**Example:** `forEach`
- `intList.forEach` accepts a parameter of type `Consumer<Integer>`, so this is the expected type for the lambda
- `Consumer<Integer>` has a function `void accept(Integer t)`, so the lambda’s argument is inferred to be of type `Integer`

```java
Runnable greeter = () -> System.out.println("Hi " + name);
```

**Example:** `Runnable`
- We are assigning a lambda to a variable of type `Runnable`, so that is the expected type for the lambda
- `Runnable` has a function `void run()`, so the lambda expression must not take any arguments
Comparison to Lambdas in a Functional Language

• Discuss: How do lambdas in Java compare to ML?
  ▪ (or your other favorite functional programming language)
Tradeoffs vs. Lambdas in ML

• **Succinctness**
  - ML’s functions shorter to invoke: `aRunnable()` vs. `aRunnable.run()`
  - ML’s non-local inference means fewer type annotations
  - Java’s expected types promote local reasoning, understandability

• **Type structure**
  - ML’s structural types need not be declared ahead of time
  - Java’s nominal types can have associated semantics described in Javadoc

```java
package java.util;
/** A comparison function, which imposes a total ordering on
 * some collection of objects. */
class Comparator<T> {
    /** The implementor must ensure that
     * `sgn(compare(x, y)) == -sgn(compare(y, x))` for all x and y
     * The implementor must also ensure that the relation is
     * transitive... */
    int compare(T o1, T o2);
}
```
Method References

// Recall Java 8 code to print integers in an array
List<Integer> intList = Arrays.asList(1,2,3);
intList.forEach(i -> System.out.println(i));

// We can make the last line even shorter!
intList.forEach(System.out::println);

• System.out::println is a method reference
  ▪ Captures the println method of System.out as a function
  ▪ The type is Consumer<Integer>, as required by intList.forEach
  ▪ The signature of println must match (and it does)
Method Reference Syntactic Forms

• Capturing an instance method of a particular object
  Syntax:    objectReference::methodName
  Example:   intList.forEach(System.out::println)

• Capturing a static method
  Syntax:    ClassName::methodName
  Example:   Arrays.sort(myIntegerArray, Integer::compare)

• Capturing an instance method, without capturing the object
  ▪ The resulting function has an extra argument for the receiver
  Syntax:    ClassName::methodName
  Example:   Function<Object,String> printer = Object::toString;

• Capturing a constructor
  Syntax:    ClassName::methodName
  Example:   Supplier<List<String>> listFactory = ArrayList::<String>new;
Collections Usage in Java

• Bulk operations: common usage pattern for Java collections
  ▪ Read from a source collection
  ▪ Select certain elements
  ▪ Compute collections holding intermediate data
  ▪ Summarize the results into a single answer

• Example: how much taxes do student employees pay?

```java
List<PayStub> studentStubs = new ArrayList<PayStub>();
for (Employee e in employees)
    if (e.getStatus() == Employee.STUDENT)
        studentStubs.addAll(e.payStubs());

double totalTax=0.0;
for (PayStub s in studentStubs)
    totalTax += s.getTax();
```

• Issues
  ▪ Inefficient to create temporary collections
  ▪ Verbose code
  ▪ Hard to do work in parallel
Streams: A Better Way

def double totalTax =
    employees.parallelStream()
    .filter(e -> e.getStatus() == Employee.STUDENT)
    .flatMap(e -> e.payStubs().stream())
    .sum()

• Benefits
  ▪ Shorter
  ▪ More abstract – describes what is desired
  ▪ More efficient – avoids intermediate data structure
  ▪ Runs in parallel
Streams

- Definition: a possibly-infinite sequence of elements supporting sequential or parallel aggregate operations
  - *possibly-infinite*: elements are processed lazily
  - *sequential or parallel*: two kinds of streams
  - *aggregate*: operations act on the entire stream
    - contrast: iterators

- Some stream sources
  - Invoking `.stream()` or `.parallelStream()` on any `Collection`
  - Invoking `.lines()` on a `BufferedReader`
  - Generating from a function: `Stream.generate(Supplier<T> s)`

- Intermediate operations
  - Produce one stream from another
  - Examples: `map`, `filter`, `sorted`, ...

- Terminal operations
  - Extract a value or a collection from a stream
  - Examples: `reduce`, `collect`, `count`, `findAny`
Demonstrations

- GetWords
- ComputeANumber
- ComputeABigNumber
Employees and Taxes

double totalTax =
    employees.parallelStream()
    .filter(e -> e.getStatus() == Employee.STUDENT)
    .flatMap(e -> e.payStubs().stream())
    .sum()

- Benefits
  - Shorter
  - More abstract – describes what is desired
  - More efficient – avoids intermediate data structure
  - Runs in parallel
Exercise: minimum age of seniors

- What is the minimum age of seniors in this course?
  - Assume the code opposite
  - You may use functions such as map, filter, reduce, etc.

```java
enum ClassStanding {
    FRESHMAN, SOPHOMORE,
    JUNIOR, SENIOR
}

class Student {
    String name;
    int age;
    ClassStanding year;
}

List<Student> roster = ...
```
Default Methods

- Java 8 just added several methods to Collection interfaces
  - Stream<E> stream()
  - Stream<E> parallelStream()
  - void forEach(Consumer<E> action)
  - Spliterator<E> spliterator()
  - boolean removeIf(Predicate<E> filter)

- If you defined a Collection subclass, did it just break?
- No! These were added as default methods
  - Declared in an interface with the default keyword
  - Given a body

```java
interface Collection<E> {
    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }
}
```
Default Methods: Semantics and Uses

• Semantics
  ▪ A method defined in a class always overrides a default method
  ▪ Default methods in sub-interfaces override those in super-interfaces
  ▪ Remaining conflicts must be resolved by overriding
  ▪ New syntax for invoking a default method from implementor

    A.super.m(...)

    • Important because m may be defined in two implemented interfaces, so can’t use simply super.m(...) 

• Benefits of default methods
  ▪ Extending an interface without breaking implementors
  ▪ Putting reusable code in an interface
    ▪ can reuse default methods from several interfaces
    ▪ known as traits in other languages (e.g. Scala)
Toad’s Take-Home Messages

Java 8 has new features useful in program expression

• Lambdas are a lightweight syntax for defining functions
  ▪ Support shorter and more abstract code

• Succinct manipulation of data through streams
  ▪ Support for pipelining and parallelism

• Default methods provide code reuse in interfaces
Sources and Resources

- Maurice Naftalin's Lambda FAQ
  - [http://www.lambdafaq.org/](http://www.lambdafaq.org/)

- The Java Tutorials:
  - Lambda Expressions
    - [https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)
  - Aggregate Operations
    - [https://docs.oracle.com/javase/tutorial/collections/streams/index.html](https://docs.oracle.com/javase/tutorial/collections/streams/index.html)

- Integer list example is adapted from Alfred Aho