Java Streams (1 Day)

Reference Notes

Simon Roberts, February 2017

Generics Review

- Java's generics mechanism is a compile-time consistency verification mechanism. At runtime (almost) all generic type information is lost.
- `List<String> ls;` declares a variable `ls`, and indicates that the variable refers to an object that implements the `List` interface, for which the generically typed variable will be a `String`. For all uses of the variable `ls`, the compiler verifies that uses are consistent with the declaration.
- The generalization / specialization behavior of generic type declarations do not parallel those of the type arguments. That is, a `List<Parent>` and `List<Child>` are entirely distinct types. Neither is a valid "Liskov substitution" for the other.
- To address the previous constraint, Java provides some special syntax:
  - `GT<? extends X> gt` indicates that whatever the generic type used in the object that will be assigned to `gt`, that generic type will be assignable to `X` (that is `?` is a placeholder for `X` or a subclass/implementation of `X`). This form is commonly used with function return values, as whatever they return must be assignable to the known type `X`.
  - `GT<? super X> gt` indicates that whatever the generic type used in the object that will be assigned to `gt`, that generic type will be assignable from `X` (that is `?` is `X` or a superclass/super-interface of `X`). This form is commonly used for arguments to a function, as whatever the function's argument type, it must accept the known type `X`.
  - `GT<?> gt` indicates that nothing is known about the generic type used in the object that will be assigned to `gt`. This form is typically used for operations that make no use of the generic type at all.

Functional Java Review

- Basic Functional programming concepts are central to the Streams API. In the API behavior is regularly passed into stream processing elements as an argument to a method. Behavior as a return value, along with behavior factories and behavior mutation are also commonly used idioms.

- Java is primarily an object-oriented language, and at a source code level it implements functional concepts using two tools. The first is the long-standing OO pattern that an object combines state and behavior in whatever proportions are necessary to the problem at hand. By creating objects that contain behavior but not state, these can be passed around and processed in the manner of functions.
- The second feature of Java's functional-style support capabilities are Lambda expressions. These are a syntactically simplified way to create the pure-behavior objects mentioned in the previous point. These strip down the declaration of a class and instantiation of an object from that class into a simple expression that describes the behavior. Thus the following code:

```
public class MyAdder implements
BinaryOperator<Integer> {
 public Integer apply(Integer a, Integer b) {
 return a + b;
 }
}
// …
BinaryOperator<Integer> add = new MyAdder();
```

  may be simply and cleanly implemented as:

```
BinaryOperator<Integer> add = (a,b)->a+b;
```

  Note the correspondance between the bold/underlined elements of the longhand version and the right hand side of the assignment in the lambda expression version.
- In support of the lambda expressions, Java defines a cluster of over forty interfaces that generalize the notion of a function in a way that supports use in a strongly, statically, typed language. These are defined in the package java.util.function. The key types are:
  - `Supplier<T>` an operation that takes zero arguments and returns a `T`
  - `Consumer<T>` an operation that takes a `T` as argument, and returns `void`
  - `Function<T,R>` an operation that takes a `T` as argument, and returns an `R`
  - `Predicate<T>` an operation that takes a `T` as argument, and returns `boolean`
  - `...Operator<T>` an operation for which `T` is both the argument and return type.

  Variations on these exist for two arguments, which use the `Bi…` prefix. (Operators are either `UnaryOperator` or

`BinaryOperator`, depending on whether they take a single or two arguments.) More variations exist for versions that operate on or return primitives.

Stream General Concepts

- The Stream concept supports processing many (possibly unbounded) data items through a sequence of operations, and then collecting them together incrementally to produce a final result.
- Stream sources can be infinite, but a result is not generally available until the stream processing is terminated.
- In support of potentially infinite streams, and efficiency in general, Streams are “lazy”. This means that processing of an element is initiated by the end of the stream “pulling” data out of the source, rather than the source “pushing” its items into the pipeline.
- The stream processing is intended to treat every data item as independent of all the others, and is able to run with multiple threads so that each thread handles a subset of the total data, creating an easy to use concurrent processing model.

The Stream Pipeline

- Stream processing involves three types of “step”
  - Origin, which supplies the data items one at a time
  - Non-terminal operations which transform the data items in the stream.
  - Terminal operations which absorb the stream data to produce a final, single, result. Although a single “thing”, that can be a structured type, with potentially many fields.
- Streams can be obtained from many sources, including any of Java’s collections, several file methods in the `Files` utility class, and from an interface called `Spliterator`.
- For each non-terminal operation, three relationships are possible with the data that move “downstream”
  - Each input item results in exactly one downstream item. This category is implemented using the `map()` operation. For a map, a single operation must be provided. That operation is used to mutate each data item and is typically represented using the `Function` interface.
  - Each input item results in either zero or one downstream item. Several operations provide this kind of behavior. A

common one is the `filter()` operation, which requires a behavior that implements the `Predicate` interface. Items that return true from the predicate survive downstream, those that produce false are deleted from the stream. Others limit based on other criteria, such as the total number of items passed [`skip()` and `limit()`] or whether an item has been seen already [`distinct()`].
    - Each input item might result in any number of downstream items. This operation is implemented using the `flatMap()` operation.

Terminal Operations

- Terminal operations come in three broad categories, built-in, reduction and collection.
    - Built in operations include the `forEach` operation which invokes a `Consumer` on each item it fetches from the stream.
- The stream provides some methods called `reduce`. These expect to create a single output value built by taking an “interim result” (which might be the first item in the stream) and another item from the stream and combining them to create an entirely new interim result. Importantly, this model never mutates any data, and is in consequence a) thread safe and b) prone to creating GC load (unless the intermediate result is a primitive type).
- The `collect()` operation varies the `reduce()` operation by employing mutable intermediate results. The concurrency problem is addressed by ensuring that every thread used by the stream infrastructure gets its own intermediate result.
- A class called `Collectors` provides a variety of pre-built collection tools
- The `Collector` interface can be used to package custom built collectors for repeated use (e.g. in a corporate library).

Miscellaneous Features

- Because of the cost of auto-boxing / unboxing (which instantiates potentially many wrapper objects) streams may be created in `int`, `long`, and `double` forms so that the stream data is primitive.
- Primitive streams offer some additional collection methods that might be useful for these data types.

- Stream and primitive streams provide specialized map operations that allow conversion between object and primitive stream types as needed.
- Streams provide a `sort()` method that allows putting downstream items into an order using a `Comparator`. This can be very expensive in terms of memory, since the items must all be stored in memory prior to releasing any downstream.
- The `peek()` method allows a `Consumer` operation to be executed on every item in a stream, without affecting what goes downstream. This can be useful for logging and debugging.
- Streams can be created from many sources:
  - Factories, and a builder mechanism, in the Stream interface: `empty`, `generate`, `iterate`, `of`
  - The `StreamSupport`, which can make a stream from a `Spliterator` . `Spliterator` can be implemented directly, or one can be extracted from any `Iterable`
  - Several classes in the core Java SE api provide methods that create streams of various kinds. These include: `Arrays`, `Random`, and `Files`
- Streams implement the AutoCloseable interface, so that streams that are attached to OS resources (files, databases, network connections) can be closed automatically using the try-with-resources construct.

Parallel Stream Operations

- The stream system is designed to allow “sharding” of the stream data across multiple threads for parallel execution.
- Parallel mode must usually be requested explicitly (it’s possible with a custom Spliterator implementation, along with a suitable collector to operate in parallel mode without deliberately requesting it)
- In parallel operations, it’s critical that none of the operations have side effects that are not thread safe (for example, using a map operation and incrementing a variable that is “external” to (i.e. not method-local) the behavior passed into the map operation.
- Spliterators and Collectors advertise their characteristics, including whether they’re sequential, parallel, ordered, or unordered. Attempting parallel operations on unsuitable participants (notably those that are ordered) can create significant performance impediments.
- Parallel collections might be performed in two distinct ways. One approach is to use a thread-safe collection destination. This can

cause loss of throughput if data arrive fast, and contention occurs. An alternative approach is to provide separate sub-destinations, one for each thread. This avoids contention, but requires a merge operation after the stream is exhausted. Which is faster depends on the complexity of the merge, and the contention rate.