Advanced Java

Notes

Simon Roberts, February 2016

Advanced OO Features

The `final` keyword

`final` variables must be provably assigned exactly once.
Reference-type final variables do not prevent modification of the object referred to.
`final` methods may not be overridden
`final` classes may not be subclassed

Equality

The primary purpose of the `equals` method is to find objects in collections. The meaning of equivalence, is more subtle, and almost always both domain-entity and usage dependent.

Features of an `enum`

Individual `enum` classes can have programmer defined constructors, which must be `private`.
They can also have fields and methods like any other class.
Example, including invocation of a non-default constructor:

```
public enum Suit {
  HEARTS, DIAMONDS, CLUBS, SPADES(true);
  private boolean isTrumps;

  private Suit() { this(false); }
  private Suit(boolean isTrumps) {
    this.isTrumps = isTrumps;
  }

  public String toPrettyName() {
    StringBuilder rv = new StringBuilder(
      name().toLowerCase());
    rv.setCharAt(0,
      Character.toUpperCase(rv.charAt(0)));
    if (isTrumps) {
      rv.append(" (Trumps)");
    }
    return rv.toString();
    }
```

Inner classes

Moving a class into another class creates an instance inner.
Instance inners must “belong” to an instance of the enclosing class.
Classes defined inside another class may be labeled `static`. This causes them to be associated with the class as a whole, not a single instance.
Anonymous inner classes must be instantiated in exactly one place in the source code. They do not have a name, and allow some syntax to be left out, making them shorter, and more readable.
If an anonymous inner is defined in a method, it may refer to variables in that method provided the variables are either `final` or in Java 8, “effectively `final`”

Advanced Exception Handling

Design with exceptions

The try/catch mechanism allows separation of happy-path concerns from error-recovery concerns.
The finally block should be used for cleanup / release of finite OS resources other than memory.
Careful though should be given to the exceptions that are thrown out of a method. Do not expose implementation details, report something meaningful to the caller. Use the exception's cause to report the underlying implementation problem for logging.
Good design seeks an API that requires minimum knowledge; seek to avoid APIs that can be misused if an alternate form would be robust.

try-with-resources

Resource must implement `AutoCloseable`
Resources must be declared and initialized in the try parameter block.
Separate/terminate them with semicolons:

```
try (
  FileReader in = new FileReader("input");
  FileWriter out = new FileWriter("output");
  ) {
```

Multi-catch

Multiple alternate formal parameter types can be used in Java 7+ catch blocks:

```
catch(IOException | SQLException ex)
```

These must not be on the same inheritance path
The parameter's actual type will be the nearest common ancestor
If rethrown, the methods throws clause need only refer to `IOException` and `SQLException`, not the common ancestor

Suppressed exceptions

Exceptions thrown during closing might become suppressed to allow an original cause of failure to be expressed to the caller. These are accessible to the caller via the `getSuppressed` method.

Generics

Defining a generic class / interface

```
class Thing<E> { // declare type variable E
  E item; // use it for variables etc.
```

Type erasure and non-reifiable types

Generic type information is unavailable, and cannot be used, at runtime:

```
E[] data = new E[]; // fails, cannot instantiate
array of E
```

Bounded generic types

```
class Thing<E extends Xxx>
class Thing<E extends AClass & AnIf & OtherIf>
```

Defining a generic method

First `<E>` is generic type variable declaration, all other E are using the type:

```
<E> E findOne(Collection<E> c, Predicate<E> p) {
  for (E item : c) {
```

Wildcard types

`<? extends Thing>` : generic type that must be assignment compatible with `Thing`
`<? super Thing>` : generic type that must accept assignment of a `Thing`

Dynamic Java

Annotations

Annotations are labels on parts of source code. Define as:
`public @interface MyAnnotation { }`

Then can survive only in source, into classfile, or into running `java.lang.Class` object. Controlled by annotating declaration:
`@Retention(@RetentionPolicy...`

Control legitimate source constructs to which the annotation may be applied using: `@Target({@ElementType...`

Runtime annotation processing with reflection

Load and instantiate a class with:

```
Class cl = Class.forName("com.useful.MyStuff");
Object obj = cl.newInstance(); // 0-arg constructor
```

Find methods, fields:
`cl.getDeclaredMethods()` returns methods declared in this class
`cl.getMethods()` gets accessible methods in this class
`cl.get[Declared]Fields` similar methods for accessing fields

Find annotations on method or field objects:
`method/field.getAnnotation(Class<T> annotationClass)`

Invoke methods:
`method.invoke(Object target, Object … args)`
For static methods target is `null`

Accessing fields

```
Field f;
f.set(Object target, Object value);
Object val = f.get(Object target);
```

Disable accessibility checks
`fieldOrMethod.setAccessible(true);`

Annotations with fields

Annotations can carry key / value pairs (literals in source, and read-only at runtime), which can have default values:

```
public @interface MyAnnotation {
  public String name() default “Fred”;
  public int value();
}
```

Valid types are: primitives, `String`, `Class`, an `enum`, an annotation, or an array of these

Applying annotations

If any annotation field does not have a default value, the use must specify that field's value. Do this with comma-separated x=y format:

```
@MyAnnot(name=”Fred”, value=10)
```

The special key named `value` can be assigned without the key being used provided it's the only key being defined in the annotation:

```
@MyAnnot(99) //value=99, name=“Fred” (the default)
```

Functional Java

Basic functional programming concepts

Behavior is passed as arguments and return values
Methods can take behavior as an argument, and return modified behavior as a return.
Provides reuse at a function level, not just at object level

Lambda expressions

When an interface with a single abstract method is to be implemented, it's sufficient to provide only the parameter list of the method, and the method body:

```
BiPredicate<String> bps =
  (s,t)->{return s.length() > t.length();};
```

(s,t) are the formal parameter list, inferred to be of type String.
Block is the method body.

If the body contains only a single statement returning an expression, then the example can be reduced to:

```
BiPredicate<String> bps =
  (s,t)->s.length() > t.length();
```

Standard functional interfaces

The package `java.util.function` defines 43 functional interfaces covering many single and two argument methods with different return types, and variations for primitive argument and return types.

Java 8 Interface Enhancements

New interface method types

Java 8 allows interfaces to define `static` methods, and `default` methods. Default methods are instance methods with implementations inherited from interface definition if, and only if, no class-based implementation is available. Intended for interface expansion while avoiding breaking all legacy code that does not implement the new method.

Method references

Method references create lambda expressions using a shorthand of the form: `X::y` These exist in four forms:
1) X is a classname, y is a `static` method
2) X is a classname, y is an instance method
3) X is an instance reference, y is an instance method
4) X is a classname, y is `new`

Generally, arguments to the lambda will be mapped to the arguments of the method, however, in case 2) the first lambda argument becomes the instance on which the method is invoked, and the remaining lambda arguments become the arguments to the method.

Collections API enhancements

Several new methods were added to the Collection and Map interfaces. Some of these take functional interface arguments, examples include:
```
Iterable.forEach(Consumer c)
Collection.removeIf(Predicate p)
Map.compute[IfPresent/Absent](
  K key, BiFunction remappingFunction)
```

Streams

Basic concepts

Streams are a lazy, potentially infinite, sequence of data items, with standard transformations that can be applied to create a final result.

Two broad categories of operations are known as terminal and non-terminal operations. Non-terminal operations return another stream object, possibly of a different data type. Primitive streams are also defined, so as to avoid unnecessary boxing and unboxing.
Stream operations should generally not have side effects, treating all data as immutable. This supports thread-safety and the parallel execution model that streams offer.

A `Stream` implements `AutoCloseable`, although this is only needed if the underlying data is coming from a non-memory source (such as the filesystem).

Key operations

`forEach`, `filter`, `map`, `flatMap`, `reduce` and `collect`
Note that `reduce` is intended as an entirely immutable operation, but might result in a large GC load due to high volume of intermediate objects. The `collect` operations support mutating collections, with each thread in a concurrent stream using a thread-specific collection “bucket” followed by a single threaded aggregation after the main streams have completed.

The `Collectors` class provides a number of useful pre-created collection capabilities, notably including the `groupingBy` behavior that builds a `Map` from the data using programmer supplied key extraction and an optional downstream collector to post process each stream item into the map's value field.

More ways of obtaining Streams

Streams factories:
`Stream.empty`, `Stream.generate`, `Stream.iterate`, `Stream.of`

`StreamSupport` can make a stream from a `Spliterator`, and `Iterable` has a `default` method for providing a `Spliterator`, facilitating creation of a `Stream` from an `Iterable`.

Streams are also returned from several methods in
`java.nio.file.Files`

Concurrency

Streams support parallel operations if created as such from a collection (`aCollection.parallelStream()`), but streams might also be handled as sequential, ordered or unordered
Concurrent modes often have hidden impediments/costs, particularly any necessary to maintain ordering, but also during collect operations, where data might need to be merged after processing, or might need to be collected into a thread-safe collection.

Threading

Thread and Runnable

`java.lang.Thread` defines a “virtual CPU”, this runs code starting in the run method of a `Runnable`.
Start the Thread execution using `aThread.start()`

Thread cooperation

If two threads are sharing data, and any the data might mutate, all kinds of problems can arise unless care is taken to ensure that an appropriate “happens-before” relationship is established between the mutating thread and the reading thread.

If a happens-before relationship exists between two statements a and b in code, then the effects of a that are observed by b will be visible to b. Normal sequential processing creates the happens-before relationships expected, but code optimization by reordering of unobserved effects is still possible.

2 important happens-before relationships:

The monitor unlock that occurs when a thread exits a synchronized block on an object happens-before a monitor lock operation by another thread synchronized on the same object.

The Thread.start() call happens-before any code executed by that thread.

Blocking threads

“Busy waiting” is generally wasteful of CPU. Instead, threads can be made to block (become non-runnable) awaiting conditions that will be caused by other threads using the `wait()` and `notify()` primitives of `Object`. These are hard to use well, however, and `ReentrantLock` is preferred. Java's libraries also include a `Semaphore` class, but this can be hard to use well too.

Simple approach to thread safety

The so-called “pipeline architecture” which has become popular again recently in a number of message passing toolkits is easy to use and simple enough to give high confidence in the implementation. Implement the pipleline using an implementation of java.util.concurrent.BlockingQueue (probably the array blocking queue) Ensure that data are only mutated by a thread that is the sole owner of a pointer to that object at the time of mutation. Pass the data to another thread using `myQueue.put`, and receive it using `myQueue.take`.

Thread pools

Threads are expensive to create, and they are finite kernel resources. Reusing them is preferable to one time use. Pools are provided by the core APIs through the `ExecutorService` interface. The `Executors` class has static factory methods for creating them.
Executor services can run `Runnable` jobs, or `Callable<X>` jobs. When a `Callable<X>` is `submit()`ted, a handle on that job is returned. The handle implements Future<X> and provides job control and data access methods: `isDone()`, `cancel()`, `isCanceled()`, `get()`.
Cancellation of a running job requires the cooperation of that job's code, responding to the interrupt mechanism and shutting down cleanly. Jobs that have not been started are simply removed from the queue when canceled.

Synchronizers:

Several synchronization (timing) mechanisms are provided by the core APIs, including the Thread's `join` method, the `CyclicBarrier`, and the `CountDownLatch`.

Concurrent collections

Where collections will be shared across multiple threads, be sure to use a thread-safe collection. The `java.util.concurrent` package has several to choose among.

Concurrency and scalability

Amdahl's law shows that blocking threads reduces scalability, and should be minimized. Java's concurrency packages include several utilities designed to help minimize this. These include the `LongAccumulator` and `DoubleAccumulator` classes in the `java.util.concurrent.atomic` package, `CopyOnWrite`... and `Skip`... data structures. Also, for simulations, it's much preferred to use the `ThreadLocalRandom` class rather than `Math.random`.

ThreadLocal variables

If data should be associated with a particular request (such as credentials of original requester, or transaction information) a `ThreadLocal` type variable might be useful. Typically, a singleton wrapper will be used to provide access to a single instance of the `ThreadLocal` type, and each call to `get()` made on that singleton object will provide a data item that is created on a per-thread basis. You must provide the `initialValue()` behavior used for initialization.

`ThreadLocal` variables can also provide a means of avoiding concurrency issues by giving each thread its own data, avoiding race conditions, and the locking that might otherwise be used to correct such problems. This is exemplified by the `ThreadLocalRandom` class, which should be used in highly concurrent random number generation, such as simulations.

CompletableFuture

The `CompletableFuture` API provides an easy means to chain a sequence of asynchronous tasks. Each can be executed in a separate thread drawn from a pool. The API handles passing data from the result of one task to the input of the next task.

The API also facilitates integrating other tools, such as `AsynchronousFileChannel`, because the `complete` and related methods allow control both of the life cycle of a `CompletableFuture` object, and the data (or exception) passed on to the next stage in the chain.