# Intermediate Java Notes

Prepared for Salesforce

Simon Roberts, Rev 4. July 2017

# OO Design Patterns and Features

## *Key Design Patterns*

### *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 cannot be overridden

`final` classes cannot be subclassed

Subclassing carries responsibilities that are often overlooked (not least, the symmetry of equality. It's probably a safer default behavior to make all new classes final until you've given real though to what inheritance implies for this class.

### *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.

Providing equality mandates providing a hashcode. Equal objects must return equal hashcodes.

Collections often use equality to find objects within themselves. Be very careful if equality testing is supported on objects that are mutable.

### *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; that is, have an implicit reference to an instance of the outer 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. These are called nested classes.

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. More than one instance can be created if that section of code is executed more than once.

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`”. This is Java’s version of a “closure”

## *Lab for Equality and Collections*

Define a simple class with a `String` and an int field, for example, Flower with attributes `int petalCount` and `String color`. Create a `HashSet` and populate it with three of these objects. Define simple `equals` and `hashcode` methods for this. Create a second object with the same fields as one of the elements in the set, and prove that you cannot add this, and that the set reports that it already contains the object.

If your class does not provide setter methods add them now. Modify the object in the set for which you created a duplicate and show that the set now no longer finds the object already present in the set. Next modify the duplicate so that it matches the now-modified set member, and see if you can find the existing item in the set. What behavior do you see and why?

*Builder Pattern*

The builder pattern simplifies creation of objects that have many fields, particularly if not all are always required. The builder supports the "fluent" style of use, like this:

```
Flower f = Flower.builder()
      .petals(7)
      .color("Yellow")
      .latinName("Narcissus")
      .build();
```

A well encapsulated class should never expose an incompletely or improperly initialized object, the builder pattern facilitates this for classes that require complex initialization.

### *Lab for Builder Pattern*

Implement the builder for a class `LatinNamedFlower` with the fields indicated in this sample code. Use a `static` nested class to help with the implementation.

# Exception Handling

*When things go wrong*

Problems in a program can fall into one of two basic categories; program bugs, and environmental issues such as badly formed input, or an incorrect file name. Situations that should not happen accordingg to the design of the program should cause the program to shutdown, or the current transaction to be abandoned (if the transactions are essentially independent of one another). However a properly completed program should include code to recover, or at least attempt to recover from, any situation that can occur because of external factors.

Historically, three courses of action have been used to indicate a problem that should be handled (note that the same mechanisms have generally been used to indicate problems that are fatal too). These are:

1. Sentinel values--these are values in the range of the return type of a function, but which do not make sense. For example, a null pointer.
2. Exceptions--these indicate the type of a failure, but also participate in program flow control, acting like a special-case "goto" operation.
3. Special, variant-record, or union, return types. These are return types that can embody either a successful return value of

another type, or an indication of of failure, perhaps with information about the reason for the failure.

Historically, the first option has tended to result in many bugs when programmers have failed to write the test code that checks for the special return. Null pointer exceptions are perhaps the most common representation of a program bug.

Java's exception mechanism extends the basic notion of a special-case flow control with the "checked exception" idea that makes it harder for programmers to ignore the abnormal situations that might otherwise lead to bugs. It also provides helpful resource management constructs. Unfortunately, for various reasons of varying validity, it has become common for programmers to hide checked exceptions in unchecked ones.

Exceptions are fundamentally incompatible with key concepts in functional programming, and with the re-assertion of this programming paradigm into mainstream coding, it's becoming more appropriate to use the third approach; a special "container" type that can represent a successful return or an error. Used with care, this can create code that is very robust, and is equally capable of applying pressure to programmers not to simpy ignore abnormal returns from functions. Java 8 introduced the Optional<T> type for this purpose, though more complex containers might be appropriate since Optional does not represent a reason for failure, only the fact of failure and the absence of a successful return.

### *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 thought should be given to the exceptions that are thrown out of a method. Do not expose implementation details, and be sure to report something meaningful to the caller at the level of abstraction of your object's API. The most common complaint about checked exceptions is that the caller doesn't know what to do about the problem, suggesting bad encapsulation or abstraction in the object throwing the exception.

Create a "semantic exception" that is specific to your API (unless a pre-existing one is exactly suitable) and throw that. 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.

### ***Lab for Multi-Catch***

Define three sibling exceptions `AException`, `BException`, and `CException`. These can be direct subclasses of the base `Exception` class. Ensure that you define all the constructors that delegate to the parent class.

Define a method `flaky` that throws the three exceptions you defined. Code it so that it completes normally or throws one of these exceptions randomly.

Define another method `caller` that calls the flaky `method`. If `flaky` throws either `AException` or `BException`, these should be caught by the same handler using the multi-catch syntax. The handler should print a message. `CException` should be caught separately.

Next arrange that the `caller` method rethrows `AException` and `BException`. What declaration is necessary on the `caller` method?

### *Lab for AutoCloseable*

Define a simple class `MyCloseable` that implements `AutoCloseable`. Allow the object to identify itself based on a `String` passed into the constructor. Print a message that includes this identity when the `close` method is called.

Write a method using the old style try block and allocate a `MyCloseable` object. Use an explicit `finally` block to close the resource. In the `try` block, use a random number generator to throw a checked exception half the time and print a message when this occurs. Declare this on the method and `try`/`catch` it in the caller printing a message noting that the exception was caught. Run the code, and determine that the checked exception reaches the caller.

Modify the `close` method in the `MyCloseable` object so that half the time it throws an `IOException`. Print a message to indicate that this behavior is being performed. Run the code a few more times, and notice what happens when the `close` exception arises, both when the checked exception does, and does not arise.

Modify the code to use a `try`-with-resources structure instead of the traditional `try`/`finally`. Run the code a few more times, and determine how the behavior has changed.

Modify the code so that it allocates two `MyCloseable` objects, with different identities, in the resources block. Run the code, and observe the order the messages are printed by the `close` operations.

# 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 { }
```

Annotations might survive only in source, or into classfile, or survive all the way into the running `java.lang.Class` object in memory. This is controlled by annotating the annotation's 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, wrappers, `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)
```

## ***Lab for Dependency Injection***

Create an annotation called `SetMe`. The annotation should be applicable only to fields, and be visible at runtime. It should accept one mandatory argument of type `String`. Create a class called `Framework` and give that class a `main` method.

The `main` method should read textual key value pairs from a properties file. The keys should include “Instantiate” and the value associated with

that key should be a fully qualified class name. Arrange that the `main` method loads and instantiates the named class, storing a reference to it in a variable called `target`.

The `main` method should proceed by scanning all the fields in the class. For each field that carries the `SetMe` annotation, extract the parameter value of the annotation and use that as a key to perform a lookup in the properties file. Assign the value that's found into the field in the target object.

Finally, arrange that the main method prints out the textual value of the target object (simply use the `toString` method).

Build a class to be used as the `target` value. Ensure that at least two `String` fields are annotated with `SetMe`. Give the annotations parameters, and ensure that the parameters are entered against those keys in the properties file. Give the class a `toString` method that reports the values of all the fields.

Add the fully qualified name of the class to the properties file and run the program.

# More Java 8 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)
```

### *Lab for Collections Enhancements*

Create two `Map<LocalDate, String>` objects, one called `workCalendar`, and the other called `personalCalendar`, and initialize them with some events with simple, short, textual descriptions (such as Birthday, Meeting, Vacation). Ensure that some dates are common to both maps.

Using the merge method of Map, arrange to create a new map that contains all the events from both calendars. Where more than one event occurs on the same day, present them as a comma separated string in the value of the map, for example:

> Birthday, Vacation

### *Lab for Method References*

Take the behavior that combines two events into a comma separated string used in merging the two maps and embed it in a static function that can be called using a method reference. Modify the merge invocation to use that method reference.

# 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();
```

If the method takes only a single argument, the parentheses around it can be omitted:

```
Predicate<String> ps = s -> s.length() > 3;
```

### *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.

## *Lab for Functional Operations*

If you do not already have it, create an interface called `Criterion<E>` such that it defines an abstract method and three default methods:

```
boolean test(E e);
default Criterion<E> negate();
default Criterion<E> and(Criterion<E> c);
default Criterion<E> or(Criterion<E> c);
```

The `test` method takes a generic object as an argument, and returns `true` or `false`, representing some kind of test on the argument object.

The `negate` operation should return a criterion that inverts the sense of the test performed by the original criterion. The `and` and `or` operations should take a second `Criterion` as an argument, and return a new `Criterion` object that computes the effect of combining the original and argument criteria according to the standard rules of boolean logic.

Next create a number of objects of varying kinds (for example a `LocalDate`, a `String`). Define some tests using lambda expressions for these objects and assign them to variables of `Criterion<E>` type (choose an appropriate value for the type `E`). For example, tests might represent “is the string longer than 3 characters”, “does the string start with a lowercase letter”.

Run the tests on the appropriate objects, printing out the results. Then create combined tests using the existing tests and the operations `and`, `or` and `negate`. So, for example, you should be able to assign a variable

```
Criterion<String> longAndLower =
      longString.and(lowerString);
```

Apply the combined tests to your example objects, and print the results.

# Streams

## *Basic concepts*

A Java 8 Stream represents 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`

### *Concurrent Streams*

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.

## ***Lab for Streams***

A concordance is a table of word frequencies in a document; a concordance of this sentence is:

```
a           : 4
concordance : 2
is          : 2
table       : 1
of          : 2
word        : 1
frequencies : 1
in          : 1
document    : 1
this        : 1
sentence    : 1
```

Use the streams API to create a concordance for the text of “Pride and Prejudice” by Jane Austen. The book can be obtained as a plain text (UTF 8) document from `gutenberg.org`.

First create the concordance and print it out.

Second, modify the output so that only the most frequent 200 words are

printed, and those are printed in descending order of frequency.

# Developer's View of Java Modules

### *Purpose of Modules*

Java 9 introduced the module system, which creates a larger level of encapsulation than classes. Prior to the advent of the module system, all elements of a library (classes, methods, fields, etc.) were simply public, protected, default-access, or private. There was no concept that an element might be freely accessible inside the library, but hidden from the uses of that library. The module system addresses this.

Since Java 9, elements in a module are hidden from users of that module--even if they're declared public in their sources--unless they are explicitly exported from the module.

****

# Overview of Java 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.

### *8 language-defined happens-before relationships:*

1. In code executed by a ***single thread***, program order defines happens-before relationships.
2. If a happens-before b, and b happens-before c, then a happens-before c.
3. The monitor ***unlock*** when a thread exits a `synchronized`

block happens-before a subsequent monitor ***lock*** operation by another thread synchronizing on the same object.

4. The ***write*** to a `volatile` variable happens-before a subsequent ***read*** of the same variable.

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

6. The termination of a thread happens-before any code that determines the thread has terminated (using `join` or `isAlive` methods)

7. A call to `Thread.interrupt()` happens-before the thread determines that is has been interrupted (using the methods `interrupted`, or `isInterrupted`, or by catching an `InterruptedException`)

8. The completion of a constructor happens-before the start of a finalize method of that object.

### *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` or `ReentrantReadWriteLock` are generally 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 `ArrayBlockingQueue`)

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.

For simulations, 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`; `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.