Java Best Practices

Key Issues of Software Design and Implementation

Professional software engineering differs from academic coding perhaps most notably in the need for, and variety of demands placed by, maintenance.

- The first problems that must be solved are identifying candidate areas for making a change, and understanding the code that is found.
- The second problem is that the change must be made within the constraints of the existing design, and/or the consequences of the changes to the design must be closely controlled to avoid chaos and new bugs being introduced.
- A third problem that looms large is a series of architectural concerns such as performance and scalability.

Object creation and initialization

Object creation has traditionally been performed using constructors. However, this approach has significant limitations, and other approaches are generally preferred for new development.

- Constructors must differ by argument lists (must be valid overloads).
- Constructors always return a new object or an exception.
- Constructors can only create a single data type
- However, an accessible constructor is necessary if subclassing is required.

Collection types

Collections are commonly stored using a List, and the Set is often overlooked.

- List preserves order and allows duplicates.
- Set typically imposes order, and rejects duplicates.
- Set has faster lookup (to support rejection), but imposes requirements on the contained data type to support this.
- Set interface hides the requirements, which might be equals/hashcode or an ordering
- Equality in structured types is non-trivial, equality in the presence of object oriented style inheritance is even more complex.
- Generally avoid inheritance
- Consider defaulting to final classes
- Ensure commutative equality and hashcode

Enums and Flyweights

Creating many object instances to represent identical immutable values is a waste of memory (and perhaps CPU power).

- An enum type defines type-safe, immutable, values to represent a fixed number of values.
- However, the available values of enums are fixed at compilation, for a more flexible approach, consider the flyweight property, or pools of immutable values.
- These cannot be created using constructors.

Generalization in API design, interfaces and Generics

Carefully considered generalization promotes reusability and reduces the consequences of many changes.

- Argument types impose a constraint on the code that calls the method, ensure this is as unrestrictive as it can be; make argument types as general as possible while supporting the needed features.
- Return types impose a constraint on the implementation, but offer additional features to the calling code. Make the return type as specific as it can usefully be, without creating a future maintenance headache for the method itself.
- Whenever possible, create APIs that avoid the caller having to have knowledge of the called concept's constraints (e.g. addDays, lastOfMonth, rather than setDayOfMonth in a Date object)

OO Inheritance vs composition

Traditional "implementation inheritance" provides both generalization and code reuse. However, it has a major weaknesses in class-based languages, and another major weakness in single inheritance languages. Since Java is both, it suffers from both problems.

- Single inheritance restricts variations to a single dimension; if independent dimensions of variation are needed then code must be duplicated. (This problem is avoided in multiple inheritance languages, but other problems are generally introduced).

- In class-based languages, variation is only possible up to the instant of instantiation. Once constructed, the behavior of an object is fixed by its class.
- Both of these problems are avoided using any of several composition/delegation based patterns, perhaps the best known being the “Strategy” pattern.

Immutability and final

Working with immutable data can provide two valuable benefits.

- Data passed to other subsystems (methods etc.) cannot cannot be unexpectedly changed by those subsystems. This eliminates a whole class of hard-to-debug problems.
- Concurrency issues are very greatly reduced.
- A common objection to using immutable data is that it consumes too much memory and is likely to overload the garbage collector.
- Well designed and implemented systems built on immutable data can typically offset much of these concerns by allowing data to be reused/shared. This is possible precisely because the data are immutable.
- The final keyword does not directly create immutable data nor immutable objects, but can be used to help.
- A final variable of primitive type will be immutable
- A final variable of reference type cannot refer to a different object, but if the object allows mutation, final does not affect this.
- A final field must be assigned exactly once before completion of construction. This is checked for all paths through overloaded constructors.

- A final field is not a requirement for creating immutable data, it's sufficient to provide no means of mutation.
- Data structures, such as a List might need special handling to permit efficient mutation internally within an object, but prevent mutation by external users of that object
- Java 9 enlarges the set of immutable data structures built into the core APIs

Encapsulation

Encapsulation provides means of limiting the consequences of change (by making the change invisible to clients of a class and its instances) and of knowing where to look when a change must be made.

- Features marked private are, superficially, inaccessible outside the enclosing top level curly braces that surround them.
- In principle therefore, if a field has an invalid value, an error exists inside those curly braces.
- It's important to make an effort to protect the "internal integrity" of a data type, otherwise the value of encapsulation might be lost.
- If a caller makes a meaningless request this should probably cause an exception.
- If an object's method returns a reference to a private mutable field, this breaks encapsulation.
- If an object uses a mutable object passed as an argument to a constructor or method, this too breaks encapsulation.
- If mutable fields are being used, consider copying them, or wrapping them in immutable proxies before passing them out to a caller.

- Consider copying mutable objects passed into the object as arguments to methods or constructors.
- Remember that the default clone method is a shallow copy.
- Interfaces should be designed to be “true to the concept” rather than being specific to either convenience of implementation, or convenience of use.

Object equality

Equality in simple, mathematical, types is a simple concept. However, where structured data exists, and particularly in systems with object oriented inheritance, the concept rapidly becomes more subtle, even tricky.

- Structured data cannot be simply compared with a bit for bit comparison, therefore in a language that supports structured types, a compiler cannot create an equality test, rather equality must be programmed as part of the definition of the data structure.
- If two classes exhibit a parent/child relationship, and both define an equality test, then a.equals(b) will invoke the behavior defined in the class of a, while b.equals(a) will invoke the behavior defined in the class of b. Since these are different methods, it’s likely that a non-commutative equality results.
- If inheritance is to be supported, consider defining a final equals method in the parent class.
- If inheritance is to be supported and the equality tests for parent and child are to be different, then both types should compare as equal only to exact instances of their own type, not to either parent or child.

- If the Object.equals method is overridden in a class, that class should also override the Object.hashcode method.
- Storing mutable objects in data structures that locate objects based on their internal values is error prone. If the object's values are altered, it will be in the wrong place in the structure, and will be impossible to find (or delete).

Effective checked exceptions

When under time pressure, programmers have exhibited a strong tendency to overlook potential failure modes, creating bugs that might be hard to find and that only show up under rare situations. It's quite common that these bugs don't show up at the moment of the unhandled failure, but perhaps much later when the consequences cause further consequences.

- Exceptions have an advantage over error codes, in that execution skips out of the "happy path" making it impossible to accidentally treat the error code as valid data. This avoids an entire category of bugs.
- Exceptions have been described as a "structured goto", and since goto is known to create code that's hard to understand, it's sometimes considered that exceptions do the same thing. This might be true if the handler is too far up the call stack.
- Checked exceptions document the "dual return path" nature of a method that throws an exception, making it easier to trace that abnormal flow.
- Checked exceptions also put pressure on programmers to perform a proper recovery from problem situations, since they cannot be entirely ignored.

- Problems (whether represented as exceptions or not) should almost always be handled where they arise if at all possible. However, delegating to the calling method stack is often necessary.
- When a problem should be impossible by the design intent, this represents a program bug. It's generally unsound to try to recover from this situation. Instead shut down the program, or at least the processing of this particular transaction/request.
- When delegating exceptions to callers, ensure that the exception is reported at the same "level of abstraction" as the rest of the API.
- Generally avoid simply throwing the original exception to the caller, this is likely to have two problems; first, the exception is an implementation detail that is meaningless at the caller's level of abstraction, second, that implementation detail is likely to change, and should be encapsulated like any other implementation detail.

Alternatives to exceptions

Although exceptions are generally a more robust programming mechanism than status codes, they are incompatible with the strict rules of functional programming. However, FP offers an alternative approach that has many benefits.

- A monadic approach, such as Java's Optional, or an Either class that may be found in many functional libraries, is generally preferred in fp.
- The monad avoids any possibility of proceeding down the "happy path" in an error situation.

- The monad draws attention to the possibility of failure that should be handled.
- The monad is not as “aggressive” as a checked exception, but given that many programmers simply wrap checked exceptions in unchecked ones, the overall effect is likely no less effective in practice.
- To handled code that throws checked exceptions in a functional way, it’s normal to have to define additional functional interfaces and provide a “wrapping” behavior that turns a lambda/function/method that throws an exception into one that returns an Either.
- The wrapper is usually required for unchecked exception handling too, to avoid the exception killing the processing in an uncontrolled way.
- A special case of status code is the return of a null pointer, indicating an absent result. This is better handled in an API using the Optional class.

Resource cleanup

If a program fails to clean up operating system resources in a timely fashion (other than memory) it is likely to be killed by the host operating system.

- A object wrapping that resource should implement the AutoCloseable interface and be used in a try-with-resources structure.
- Using the finally mechanism is cumbersome due to scope and other issues, and can also fail to propagate unhandled exceptions correctly to the caller.
- Java’s original finalization method should not be used since it interferes with the garbage collector.
- If a last-chance resource cleanup mechanism is required, it’s better to use the References API.

- The References API provides weak, soft, and phantom references.
- References can be used to store crucial OS-level information duplicating what's in the primary resource-wrapping object.
- The references can be collected on queues after their primary objects have been garbage collected.
- Process the references on the queue to clean up the underlying OS resource.

Performance and benchmarking

The Java Virtual Machine performs dynamic optimizations of running code, resulting in good performance.

- When a class is first loaded no optimizations have been provided, and performance will be relatively poor.
- Optimizations include restructuring of code (such as inlining a method call) and compilation to native host binary.
- During optimizations, CPU power is typically borrowed from other operations, possibly including Java user threads.
- Run benchmark experiments for long enough to ensure all optimizations are complete, this might be in the region of 30 seconds after the last class has loaded.
- Java class loading is lazy, so "the last class" might not happen until a particular execution path is followed that requires it.
- The JVM can generate log entries indicating when compilation is occurring, the flag -XX:+PrintCompilation is the most basic form of this.

- Flags with the -XX prefix are non-standard and subject to variation between JVMs (different distributions and different versions).
- System.nanoTime() returns an arbitrary offset time, suitable for measuring elapsed time, in units of nanoseconds, but likely with lower resolution.

Effective documentation

The problem of understanding the code of a project might be simplified by effective documentation. However, documentation that is incorrect (for example, if it's out of date and describes what used to happen) can be worse than no documentation at all.

- The smallest methods are easiest to understand. Strive for five lines or so, creating additional (probably private) methods to support this.
- Names should be accurate, kept up to date, and have a verbosity that avoids ambiguity.
- Generally name length should be correlated with the visibility of the method.
- Documentation that cannot be wrong is best. This can happen if documentation is created in the form of tests.
- The assert keyword allows statements of "this fact is true" to be embedded in the source code.
- Such assertions can describe key facts about how a piece of code works.
- The bytecode associated with an assert will be removed from the running system (during classloading) unless the -ea or -enableassertions runtime flags are used. This allows zero performance impact in a production system (although the checks are of course bypassed)

- Assertions can be enabled on a package by package basis if a problem must be tracked down and the performance impact of running all the assertions is too great.
- Well written unit tests perform a documentary function.
- Like the BDD given/when/then model a good unit test effectively says “if you set this up, then when you do this, you’ll get this”.