Principles of Software Construction: Objects, Design, and Concurrency

API Design 2: principles

Josh Bloch Charlie Garrod

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4c due Thursday, 10/29, 11:59pm EST
- Homework 5 coming soon
- Team sign-up deadline is Wednesday, 11/4, 5 pm EDT
- Next Tuesday, 11/3 is election day – no lecture
  - **VOTE, if you have not already done so!!!**
- Online midterm exam Thursday, 11/4–11/5
  - Same format as first midterm – no lecture on 11/5
  - Review session Monday, November 2nd, 6:30 - 8:30 pm

Key concepts from last lecture

- APIs took off in the past 30 years, & gave us super-powers
- Good APIs are a blessing; bad ones, a curse
- Using a design process greatly improves API quality
- Naming is critical to API usability

Unfinished Business

Naming
(See lecture 15 for these slides)

Outline

I. General principles (6)
II. Class design (5)
III. Method design (9)
IV. Exception design (4)
V. Documentation (2)

Characteristics of a Good API

Review

- Easy to learn
- Easy to use, even if you take away the documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

1. API Should Do One Thing and Do it Well

- Functionality should be easy to explain in a sentence
  - **If it's hard to name, that's generally a bad sign**
  - Be amenable to splitting and merging modules
- Several *composable* APIs are better than one big one
  - Learn and use the APIs as needed
  - Only pay for the functionality you need

What not to do

```
public abstract class Calendar implements
Serializable, Cloneable, Comparable<Calendar>
```

The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week. An instant in time can be represented by a millisecond value that is an offset from the Epoch, January 1, 1970 00:00:00.000 GMT (Gregorian).

What not to do, continued

Like other locale-sensitive classes, Calendar provides a class method, getInstance, for getting a generally useful object of this type. Calendar's getInstance method returns a Calendar object whose calendar fields have been initialized with the current date and time:

```
Calendar rightNow = Calendar.getInstance();
```

A Calendar object can produce all the calendar field values needed to implement the date-time formatting for a particular language and calendar style (for example, Japanese-Gregorian, Japanese-Traditional). Calendar defines the range of values returned by certain calendar fields, as well as their meaning. For example, the first month of the calendar system has value MONTH == JANUARY for all calendars. Other values are defined by the concrete subclass, such as ERA. See individual field documentation and subclass documentation for details.

etc., etc., etc., etc., etc., etc., etc., etc.

What is a `Calendar` instance? What does it do?

- **I have no clue!!!**
  - Combines every calendrical concept without addressing any
- Confusion, bugs, & pain caused by this class are immense
- Thankfully it's obsolete as of Java 8; use `java.time`
- Inexplicably, it's not deprecated, even as of Java 14!
- If you're working on an API and you see a class description that looks like this, run screaming!

2. API should be as small as possible but no smaller

"Everything should be made as simple as possible, but not simpler." – Einstein

- API must satisfy its requirements
  - Beyond that, more is not necessarily better
  - But smaller APIs sometimes solve more problems
  - **Generalizing an API can make it smaller(!)**
- **When in doubt, leave it out**
  - Functionality, classes, methods, parameters, etc.
  - **You can always add, but you can never remove**
    - More precisely, you can always provide stronger guarantees but you can never retract a promise.
    - e.g., you can expose additional methods, types, or enum constants; broaden parameter types; narrow return type
    - Stronger guarantees in extendable types are problematic

Conceptual weight (a.k.a. conceptual surface area)

- **Conceptual weight** more important than "physical size"
- *def.* The number & difficulty of new concepts in API
  - i.e., the amount of space the API takes up in your brain
- Examples where growth adds little conceptual weight:
  - Adding overload that behaves consistently with existing methods
  - Adding `arccos` when you already have `sin`, `cos`, and `arcsin`
  - Adding new implementation of an existing interface
- Look for a high *power-to-weight ratio*
  - In other words, look for API that lets you do a lot with a little

Example: generalizing an API can make it smaller

Subrange operations on `Vector` *– legacy List implementation*

```
public class Vector {
    public int indexOf(Object elem, int index);
    public int lastIndexOf(Object elem, int index);
    ...
}
```

- Not very powerful
  - Supports only search operation, and only over certain ranges
- Hard to use without documentation
  - What are the semantics of `index`?
  - I don't remember, and it isn't obvious.

Example: generalizing an API can make it smaller

Subrange operations on `List`

```
public interface List<T> {
    List<T> subList(int fromIndex, int toIndex);
    ...
}
```

- Extremely powerful!
  - **Supports *all* List operations on *all* subranges**
- Easy to use even without documentation

“Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.”

— Antoine de Saint-Exupéry, *Airman’s Odyssey*, 1942

3. Don’t make users do anything library could do for them

APIs should exist to serve their users and not vice-versa

- Reduce need for **boilerplate code**
  - Generally done via cut-and-paste
  - Ugly, annoying, and error-prone

```
// DOM code to write an XML document to a specified output stream.
static final void writeDoc(Document doc, OutputStream out) throws IOException {
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
                        doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out));
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can’t happen!
  }
}
```

4. Make it easy to do what's common, possible to do what's less so

- If it's hard to do common tasks, users get upset
- For common use cases
  - Don't make users think about obscure issues - provide reasonable defaults
  - Don't make users do multiple calls - provide a few well-chosen convenience methods
  - Don't make user consult documentation
- For uncommon cases, it's OK to make users work more
- Don't worry too much about truly rare cases
  - It's OK if your API doesn't handle them, at least initially

5. Implementation should not impact API

- Natural human tendency to design what you know how to implement – fight it!
  - Design for the user; then figure out how to implement
- APIs written once, used *many* times
  - So put in the time upfront to transcend implementation
- Implementation constraints may change; API won't
  - When this happens, API becomes unexplainable

6. Be consistent

Within your API and across the platform

- Users will assume consistency
  - Inconsistency causes frustration and errors
  - Worst case: silent errors based on false assumptions
- Many kinds of consistency are important
  - e.g., vocabulary, semantics, parameter ordering, type usage…
- But beware:

  "A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines."

  – Ralph Waldo Emerson, "Self Reliance", 1841

Outline

I. General principles (6)
II. Class design (5)
III. Method design (9)
IV. Exception design (4)
V. Documentation (2)

1. Minimize Mutability

- Parameter types should be immutable
  - Eliminates need for defensive copying
- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
  - Disadvantage: separate object for each value
- If mutable, keep state-space small, well-defined
  - Make clear when it's legal to call which method

Bad: `Date`

Good: `java.time.Instant`

2. Minimize accessibility of everything

- Make classes, members as private as possible
  - If it's at least package-private, it's not a part of the API
- Public classes should have no public fields (with the exception of constants)
- Minimizes *coupling*
  - Allows components to be, understood, used, built, tested, debugged, and optimized independently

3. Subclass only when an is-a relationship exists

- Subclassing implies substitutability (Liskov)
  - Makes it possible to pass an instance of subclass wherever superclass is called for
  - And signals user that it's OK to do this
- If not is-a but you subclass anyway, all hell breaks loose
  - Bad: `Properties extends Hashtable`
    `Stack extends Vector, Thread extends Runnable`
- Never subclass just to reuse implementation
- Ask yourself "Is every `Foo` really a `Bar`?"
  - If you can't answer yes with a straight face, don't subclass!

4. Design & document for inheritance or else prohibit it

- Inheritance violates encapsulation (Snyder, ’86)
  - Subclasses are sensitive to implementation details of superclass
- **If you allow subclassing, document *self-use***
  - How do methods use one another?
- Conservative policy: all concrete classes uninheritable
- See *Effective Java* Item 19 for details

Bad: Many concrete classes in Java libraries

Good: **`AbstractSet`**, **`AbstractMap`**

5. Don’t expose a new type that lacks meaningful contractual refinements on an existing supertype

- Just use the existing type
- Reduces conceptual surface area
- Increases flexibility
- Resist the urge to expose type just because it’s there

Outline

I. General principles (6)
II. Class design (5)
III. Method design (9)
IV. Exception design (4)
V. Documentation (2)

1. “Fail Fast” – prevent failure, or fail quickly, predictably, and informatively

- Ideally, API should make misuse impossible
  - Fail at compile time or sooner
- Misuse that’s statically detectable is second best
  - Fail at build time, with proper tooling
- Misuse leading to prompt runtime failure is third best
  - Fail when first erroneous call is made
  - Method should succeed or have no effect *(failure-atomicity)*
- Misuse that can lie undetected is what nightmares are made of
  - Fail at an undetermined place and time in the future

Misuse that's statically detectable (and fails promptly at runtime if it eludes static analysis)

```
// The WRONG way to require one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Need at least 1 arg");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

API that makes misuse impossible

```
// The right way to require one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

Won't compile if you try to invoke with no arguments

No validity check necessary

API that fails at an unknown time and place

Sweet dreams…

```
/** A Properties instance maps strings to strings */
public class Properties extends Hashtable {
   public Object put(Object key, Object value);

   // Throws ClassCastException if this properties
   // contains any keys or values that are not strings
   public void save(OutputStream out, String comments);
}
```

2. Use appropriate parameter and return types

- Input types as general as possible (but no more general)
- Return type as specific as possible (but no more specific)
- Avoid boolean input parameters
- **Don't use String if a better type exists**
  - Strings are cumbersome, error-prone, and slow
- Don't use floating point for monetary values
  - Binary floating point causes inexact results!
- Use double (64 bits) rather than float (32 bits)
  - Unless you know (via benchmarking) that you need the performance, and you can tolerate the low precision

3. Use consistent parameter ordering across methods

- Especially important if parameter types identical

```
#include <string.h>
char *strncpy(char *dst, char *src, size_t n);
void  bcopy  (void *src, void *dst, size_t n);
```

- Also important if parameter types "overlap," e.g., (`int`, `long`) can hurt you if you pass two `int` values

`java.util.collections` – first parameter always collection to be modified or queried

`java.util.concurrent` – time always specified as `long delay, TimeUnit unit`

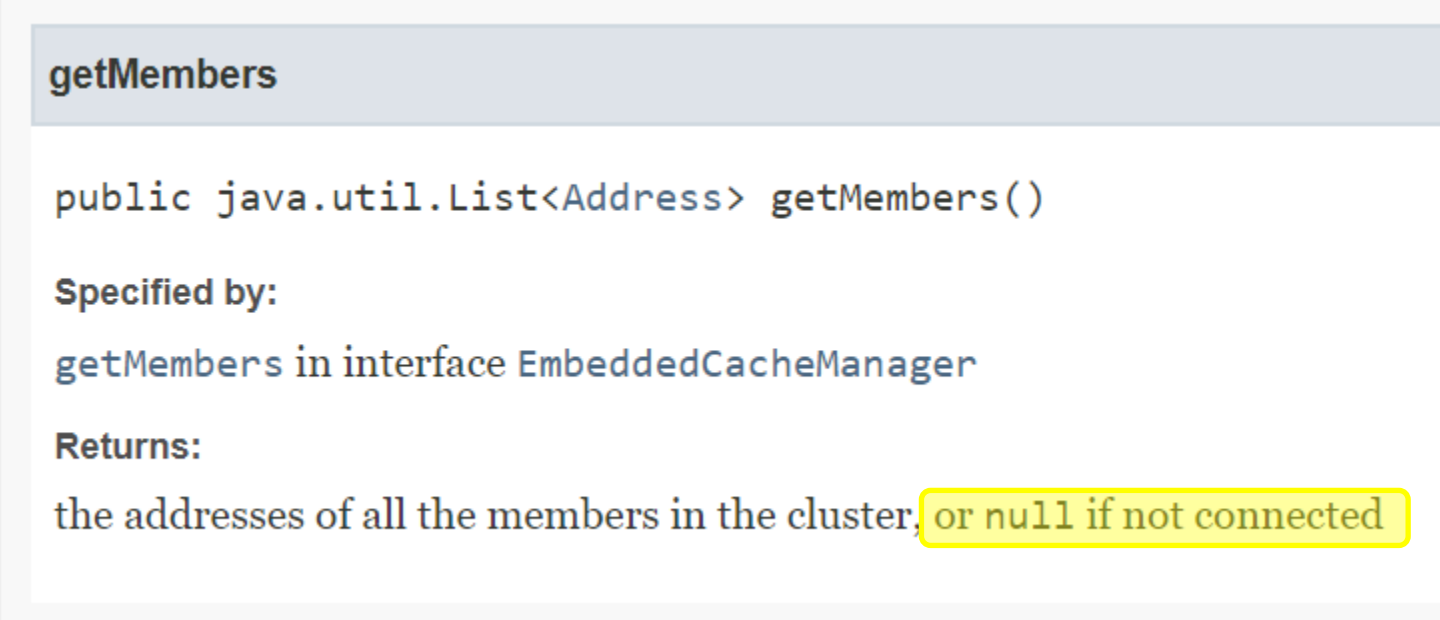
4. Avoid long parameter lists

- **Three or fewer parameters is ideal**
  - More and users will have to refer to docs
- Long lists of identically typed params are very harmful
  - Programmers transpose parameters by mistake
  - Programs still compile and run, but misbehave!
- Techniques for shortening parameter lists
  - Break up method
  - Create helper class to hold several parameters
    - Often they're otherwise useful, e.g., `Duration`
  - Use builder pattern

```
// Eleven (!) parameters including five consecutive ints
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
      DWORD dwStyle, int x, int y, int nWidth, int nHeight,
      HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
      LPVOID lpParam);
```

5. Avoid return values that demand exceptional processing

- Client should not have to write extra code
  - All cases should just work (including boundary cases).
- e.g., return empty collection or 0-length array , not **`null`**
- This example is from the JBOSS Application Server



getMembers

```
public java.util.List<Address> getMembers()
```

Specified by:

`getMembers` in interface `EmbeddedCacheManager`

Returns:

the addresses of all the members in the cluster, or `null` if not connected

6. Handle boundary conditions (edge cases, corner cases) gracefully

- Previous example is one example of what not to do
  - Returning null instead of empty collection or 0-length array
- And there are many others
  - e.g., C's `bsearch` function returns no useful information if target is not found

7. Do not overspecify the behavior of methods

- Don't specify internal details
  - It's not always obvious what's an internal detail
- All tuning parameters are suspect
  - **Let client specify intended use, not internal detail**
  - Bad: number of buckets in table; Good: intended size
  - Bad: number of shards; Good: intended concurrency level
- Do not specify value returned by hash functions!
  - You lose the flexibility to improve them

8. Provide programmatic access to all data available in string form

- Otherwise, clients will be forced to parse strings
  - Painful
  - Error prone
  - **Worst of all, it turns string format into de facto API**
- Java got this wrong for exception stack traces…
  - But `StackTraceElement[] getStackTrace()` added in Java 4
  - At the same time as we enhanced stack trace string format
  - A few users complained at the time, but quickly got over it

9. Overload with care

- Avoid *ambiguous overloadings*
  - Multiple overloadings applicable to same actuals
- Just because you can doesn't mean you should
  - **Often better to use a different name**
  - But overloadings that really do the same thing for different types are a good thing; they reduce conceptual weight
    - Especially true for primitive types and arrays in Java
- If you must provide ambiguous overloadings, ensure same behavior for same arguments

```
// Bad – ambiguous overloading with different behaviors
public TreeSet(Collection<E> c); // Ignores order
public TreeSet(SortedSet<E> s);  // Respects order
```

Outline

I. General principles(6)
II. Class design (5)
III. Method design (9)
IV. Exception design (4)
V. Documentation (2)

1. Throw exceptions to indicate exceptional conditions

Don't force client to use them for control flow

```
private byte[] a = new byte[CHUNK_SIZE];

void processBuffer (ByteBuffer buf) {
    try {
        while (true) {
            buf.get(a);
            processBytes(a, CHUNK_SIZE);
        }
    } catch (BufferUnderflowException e) {
        int remaining = buf.remaining();
        buf.get(a, 0, remaining);
        processBytes(a, remaining);
    }
}
```

2. Favor unchecked exceptions

- Use checked when client *must* take recovery action
- Unchecked is generally for programming error
- Overuse of checked exceptions causes boilerplate

```
try {
    Foo f = (Foo) super.clone();
    ....
} catch (CloneNotSupportedException e) {
    // This can't happen, since we're Cloneable
    throw new AssertionError();
}
```

3. Favor the reuse of existing exception types

Special case of final class design principle

- Especially `IllegalArgumentException` and `IllegalStateException`
- Makes APIs easier to learn and use
- Subclass existing types if you need extra methods

4. Include failure-capture information in exceptions

- e.g., `IndexOutOfBoundsException` should include index and ideally, bounds of access
  - In early releases, it didn't; now it includes index, but not bounds
  - Index was added to detail message in JDK 1.1
  - `IndexOutOfBoundsException(int index)` added in Java 9!
- Eases diagnosis and repair or recovery
- For unchecked exceptions, message suffices
- For checked exceptions, provide accessors too

Outline

I. General principles (8)
II. Class design (5)
III. Method design (9)
IV. Exception design (4)
V. Documentation (2)

API documentation is critical

- Documentation *is* specification
- Poor documentation risks loss of control over spec
- Stack overflow *becomes* the spec…
- And you're forced to support incorrect uses forever
- Accelerates *Hyrum's Law*:

  With a sufficient number of users of an API,
  it does not matter what you promise in the contract:
  all observable behaviors of your system
  will be depended on by somebody.

Document religiously

- Document **every** class, interface, method, constructor, parameter, and exception
  - Class: what an instance represents
  - Method: contract between method and its client
    - Preconditions, postconditions, side-effects
  - Parameter: indicate units, form, ownership
- Document thread safety
- If class is mutable, document state space
- If API spans packages, JavaDoc is *not* sufficient
  - Remember the collections framework?

API Design Summary

- A good API is a blessing; a bad one a curse
- API Design is hard
  - Accept the fact that we all make mistakes
  - But do your best to avoid them
- This talk and the last covered some heuristics of the craft
  - Don't adhere to them slavishly, but...
  - Don't violate them without good reason
- Your APIs won’t be perfect, but with a lot of hard work and bit of luck, they’ll be good enough