How to Design a Good API and Why it Matters

Joshua Bloch

Chief Java Architect

Google™

Why is API Design Important?

- APIs can be among a company's greatest assets
  - —Customers invest heavily: buying, writing, **learning**
  - —Cost to stop using an API can be prohibitive
  - —Successful public APIs capture customers
- Can also be among company's greatest liabilities
  - —Bad API can cause unending stream of support calls
  - —Can inhibit ability to move forward
- Public APIs are forever - one chance to get it right

Why is API Design Important *to You*?

- If you program, you are an API designer
  - —Good code is modular–each module has an API
- Useful modules tend to get reused
  - —Once module has users, can't change API at will
  - —Good reusable modules are corporate assets
- Thinking in terms of APIs improves code quality

Characteristics of a Good API

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

Outline

I. The Process of API Design

II. General Principles

III. Class Design

IV. Method Design

V. Exception Design

VI. Refactoring API Designs

I. The Process of API Design

Gather Requirements–with a Healthy Degree of Skepticism

- Often you'll get proposed solutions instead
  - —Better solutions may exist
- Your job is to extract true requirements
  - —Should take the form of **use-cases**
- Can be easier and more rewarding to build something more general

What they say: *"We need new data structures and RPCs with the Version 2 attributes"*

What they mean: *"We need a new data format that accommodates evolution of attributes"*

Start with Short Spec–1 Page is Ideal

- At this stage, agility trumps completeness
- Bounce spec off as many people as possible
  - —Listen to their input and take it seriously
- If you keep the spec short, it's easy to modify
- Flesh it out as you gain confidence
  - —This necessarily involves coding

Sample Early API Draft (1)

```
// A strategy for retrying computation in the face of failure.
public interface RetryPolicy {
    // Called after computation throws exception
    boolean isFailureRecoverable(Exception e);

    // Called after isFailureRecoverable returns true.
    // Returns next delay in ns, or negative if no more retries.
    long nextDelay(long startTime, int numPreviousRetries);
}
```

Sample Early API Draft (2)

```
public class RetryPolicies {
    // Retrying decorators (wrappers)
    public static ExecutorService retryingExecutorService(
      ExecutorService es, RetryPolicy policy);
    public static Executor retryingExecutor(
      Executor e, RetryPolicy policy);
    public static <T> Callable<T> retryingCallable(
      Callable<T> computation, RetryPolicy policy);
    public static Runnable retryingRunnable(
      Runnable computation, RetryPolicy policy);

    //  Delay before nth retry is random number between 0 and 2^n
    public static RetryPolicy exponentialBackoff(
       long initialDelay, TimeUnit initialDelayUnit,
       long timeout, TimeUnit timeoutUnit,
       Class<? extends Exception>... recoverableExceptions);

    public static RetryPolicy fixedDelay(long delay,
       TimeUnit delayUnit, long timeout, TimeUnit timeoutUnit,
       Class<? extends Exception>... recoverableExceptions);
}
```

Write to Your API Early and Often

- Start *before* you've implemented the API
  - —Saves you doing implementation you'll throw away
- Start *before* you've even specified it properly
  - —Saves you from writing specs you'll throw away
- Continue writing to API as you flesh it out
  - —Prevents nasty surprises right before you ship
- Code lives on as examples, unit tests
  - —Among the most important code you'll ever write
  - —Forms the basis of *Design Fragments* [Fairbanks, Garlan, & Scherlis, OOPSLA '06, P. 75]

Writing to SPI is Even More Important

- Service Provider Interface (SPI)
  - Plug-in interface enabling multiple implementations
  - Example: Java Cryptography Extension (JCE)
- Write multiple plug-ins before release
  - If one, it probably won't support another
  - If two, it will support more with difficulty
  - If three, it will work fine
- Will Tracz calls this “The Rule of Threes”
  (*Confessions of a Used Program Salesman*, Addison-Wesley, 1995)


Good


Bad

Maintain Realistic Expectations

- Most API designs are over-constrained
  - —You won't be able to please everyone
  - —Aim to displease everyone equally
- Expect to make mistakes
  - —A few years of real-world use will flush them out
  - —Expect to evolve API

II. General Principles

API Should Do One Thing and Do it Well

- Functionality should be easy to explain
  - —**If it's hard to name, that's generally a bad sign**
  - —Good names drive development
  - —Be amenable to splitting and merging modules

Good: `Font, Set, PrivateKey, Lock, ThreadFactory, TimeUnit, Future`

Bad: `DynAnyFactoryOperations, _BindingIteratorImplBase, ENCODING_CDR_ENCAPS, OMGVMCID`

API Should Be As Small As Possible But No Smaller

- API should satisfy its requirements
- **When in doubt leave it out**
  - —Functionality, classes, methods, parameters, etc.
  - —**You can always add, but you can never remove**
- *Conceptual weight* more important than bulk
- Look for a good *power-to-weight ratio*

Implementation Should Not Impact API

- Implementation details
  - —Confuse users
  - —Inhibit freedom to change implementation
- Be aware of what is an implementation detail
  - —Do not overspecify the behavior of methods
  - —For example: **do not specify hash functions**
  - —All tuning parameters are suspect
- Don't let implementation details “leak” into API
  - —On-disk and on-the-wire formats, exceptions

Minimize Accessibility of Everything

- Make classes, members as private as possible
- Public classes should have no public fields (with the exception of constants)
- Maximizes *information hiding* [Parnas]
- Minimizes *coupling*
  - Allows modules to be, understood, used, built, tested, debugged, and opimized independently

Names Matter–API is a Little Language

- Names Should Be Largely Self-Explanatory
  - —Avoid cryptic abbreviations
- Be consistent
  - —Same word means same thing throughout API
  - —(and ideally, across APIs on the platform)
- Be regular–strive for symmetry
- If you get it right, code reads like prose

```
if (car.speed() > 2 * SPEED_LIMIT)
   speaker.generateAlert("Watch out for cops!");
```

Documentation Matters

Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.

- D. L. Parnas, Software Aging. *Proceedings of the 16th International Conference on Software Engineering, 1994*

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 state space very carefully

Consider Performance Consequences of API Design Decisions

- Bad decisions can limit performance
  - Making type mutable
  - Providing constructor instead of static factory
  - Using implementation type instead of interface
- Do not warp API to gain performance
  - Underlying performance issue will get fixed, but headaches will be with you forever
  - Good design usually coincides with good performance

Effects of API Design Decisions on Performance are Real and Permanent

- `Component.getSize()` returns `Dimension`
- `Dimension` is mutable
- Each `getSize` call must allocate `Dimension`
- Causes *millions* of needless object allocations
- Alternative added in 1.2; old client code still slow

API Must Coexist Peacefully with Platform

- Do what is customary
  - —Obey standard naming conventions
  - —Avoid obsolete parameter and return types
  - —Mimic patterns in core APIs and language
- Take advantage of API-friendly features
  - —Generics, varargs, enums, default arguments
- Know and avoid API traps and pitfalls
  - —Finalizers, public static final arrays
- **Don’t Transliterate APIs**

III. Class Design

Minimize Mutability

- 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`, `Calendar`

Good: `TimerTask`

Subclass Only Where It Makes Sense

- Subclassing implies substitutability (Liskov)
  - —Subclass only when is-a relationship exists
  - —Otherwise, use composition
- Public classes should not subclass other public classes for ease of implementation

Bad:
```
Properties extends Hashtable
Stack extends Vector
```

Good:
```
Set extends Collection
```

Design and Document for Inheritance or Else Prohibit it

- Inheritance violates encapsulation (Snyder, ‘86)
  - —Subclass sensitive to implementation details of superclass
- If you allow subclassing, document *self-use*
  - —How do methods use one another?
- Conservative policy: all concrete classes final

Bad: Many concrete classes in J2SE libraries

Good: `AbstractSet`, `AbstractMap`

IV. Method Design

Don't Make the Client Do Anything the Module Could Do

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

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// 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!
  }
}
```

Don't Violate the Principle of Least Astonishment

- User of API should not be surprised by behavior
  - It's worth extra implementation effort
  - It's even worth reduced performance

```
public class Thread implements Runnable {
  // Tests whether current thread has been interrupted.
  // Clears the interrupted status of current thread.
  public static boolean interrupted();
}
```

Fail Fast–Report Errors as Soon as Possible After They Occur

- Compile time is best - static typing, generics
- At runtime, first bad method invocation is best
  - Method should be *failure-atomic*

```
// 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);
}
```

Provide Programmatic Access to All Data Available in String Form

- Otherwise, clients will parse strings
  - Painful for clients
  - Worse, turns string format into de facto API

```
public class Throwable {
  public void printStackTrace(PrintStream s);
  public StackTraceElement[] getStackTrace(); // Since 1.4
}

public final class StackTraceElement {
  public String getFileName();
  public int getLineNumber();
  public String getClassName();
  public String getMethodName();
  public boolean isNativeMethod();
}
```

Overload With Care

- Avoid *ambiguous overloadings*
  - —Multiple overloadings applicable to same actuals
  - —Conservative: no two with same number of args
- Just because you can doesn't mean you should
  - —Often better to use a different name
- If you must provide ambiguous overloadings, ensure same behavior for same arguments

```
public TreeSet(Collection c); // Ignores order
public TreeSet(SortedSet s);  // Respects order
```

Use Appropriate Parameter and Return Types

- Favor interface types over classes for input
  - Provides flexibility, performance
- Use most specific possible input parameter type
  - Moves error from runtime to compile time
- 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)
  - Precision loss is real, performance loss negligible

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);
```

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

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

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 harmful
  - —Programmers transpose parameters by mistake
  - —Programs still compile, run, but misbehave!
- Techniques for shortening parameter lists
  - —Break up method
  - —Create helper class to hold parameters

```
// Eleven parameters including four 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);
```

Avoid Return Values that Demand Exceptional Processing

- Return zero-length array or empty collection, not **`null`**

```
package java.awt.image;
public interface BufferedImageOp {
  // Returns the rendering hints for this operation,
  // or null if no hints have been set.
  public RenderingHints getRenderingHints();
}
```

V. Exception Design

Throw Exceptions to Indicate Exceptional Conditions

- Don’t force client to use exceptions for control flow

```
private byte[] a = new byte[BUF_SIZE];
void processBuffer (ByteBuffer buf) {
  try {
    while (true) {
      buf.get(a);
      processBytes(tmp, BUF_SIZE);
    }
  } catch (BufferUnderflowException e) {
    int remaining = buf.remaining();
    buf.get(a, 0, remaining);
    processBytes(bufArray, remaining);
  }
}
```

- Conversely, don’t fail silently

```
ThreadGroup.enumerate(Thread[] list)
```

Favor Unchecked Exceptions

- Checked – client must take recovery action
- Unchecked – 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();
}
```

Include Failure-Capture Information in Exceptions

- Allows diagnosis and repair or recovery
- For unchecked exceptions, message suffices
- For checked exceptions, provide accessors

VI. Refactoring API Designs

1. Sublist Operations in Vector

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

- Not very powerful - supports only search
- Hard too use without documentation

Sublist Operations Refactored

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

- Extremely powerful - supports *all* operations
- Use of interface reduces conceptual weight
  - —High power-to-weight ratio
- Easy to use without documentation

2. Thread-Local Variables

```
// Broken - inappropriate use of String as capability.
// Keys constitute a shared global namespace.
public class ThreadLocal {
    private ThreadLocal() { } // Non-instantiable

    // Sets current thread's value for named variable.
    public static void set(String key, Object value);

    // Returns current thread's value for named variable.
    public static Object get(String key);
}
```

Thread-Local Variables Refactored (1)

```
public class ThreadLocal {
     private ThreadLocal() { } // Noninstantiable

     public static class Key { Key() { } }

     // Generates a unique, unforgeable key
     public static Key getKey() { return new Key(); }

     public static void set(Key key, Object value);
     public static Object get(Key key);
}
```

- Works, but requires boilerplate code to use

```
static ThreadLocal.Key serialNumberKey = ThreadLocal.getKey();
ThreadLocal.set(serialNumberKey, nextSerialNumber());
System.out.println(ThreadLocal.get(serialNumberKey));
```

Thread-Local Variables Refactored (2)

```
public class ThreadLocal<T> {
    public ThreadLocal() { }
    public void set(T value);
    public T get();
}
```

- Removes clutter from API and client code

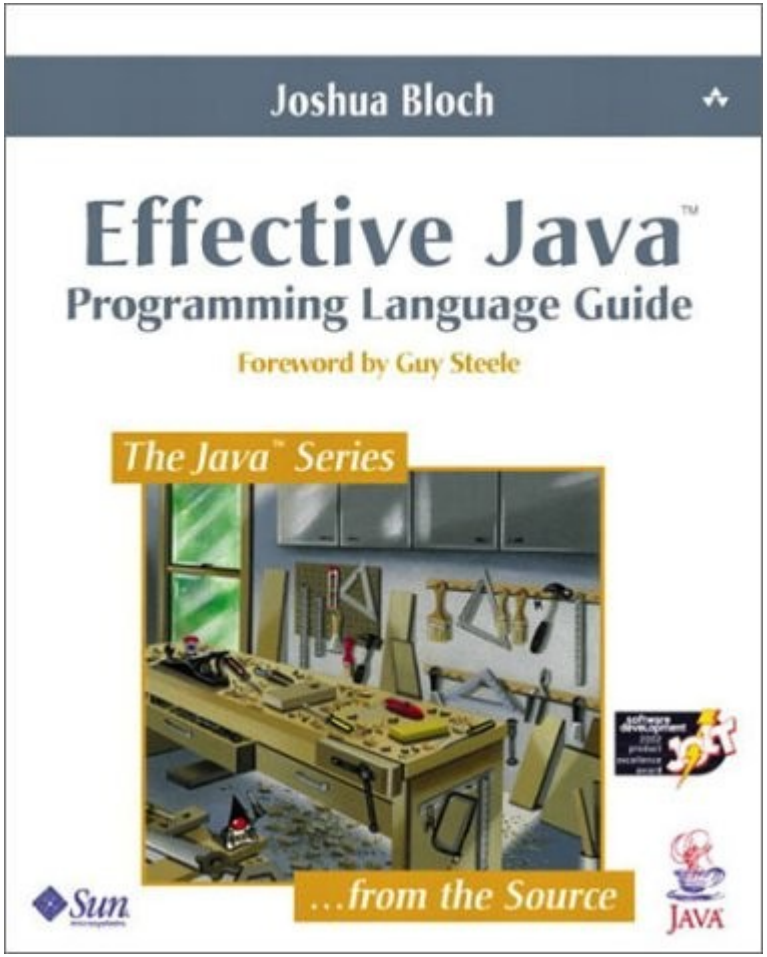
```
static ThreadLocal<Integer> serialNumber =
    new ThreadLocal<Integer>();
serialNumber.set(nextSerialNumber());
System.out.println(serialNumber.get());
```

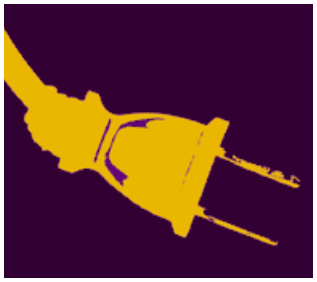
Conclusion

- API design is a noble and rewarding craft
  - —Improves the lot of programmers, end-users, companies
- This talk covered some heuristics of the craft
  - —Don't adhere to them slavishly, but...
  - —Don't violate them without good reason
- API design is tough
  - —Not a solitary activity
  - —Perfection is unachievable, but try anyway

Shameless Self-Promotion

“Bumper-Sticker API Design” - P. 506 in OOPSLA ‘06 Program





How to Design a Good API and Why it Matters

Joshua Bloch

Chief Java Architect

Google™