Principles of Software Construction:

How to Design a Good API and Why it Matters

Josh Bloch Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 4b due **next Thursday**
- HW 4a feedback available after class

Key concepts from Tuesday…

- Multiple threads run in same program concurrently
  - Can improve style and performance
  - But you must avoid shared mutable state…
  - Or synchronize properly
- GUI programming a simple case of multithreading
  - Event dispatch thread handles all GUI events
  - Swing calls must be on EDT
  - Time-consuming (non-Swing) must be off EDT

isr institute for SOFTWARE RESEARCH

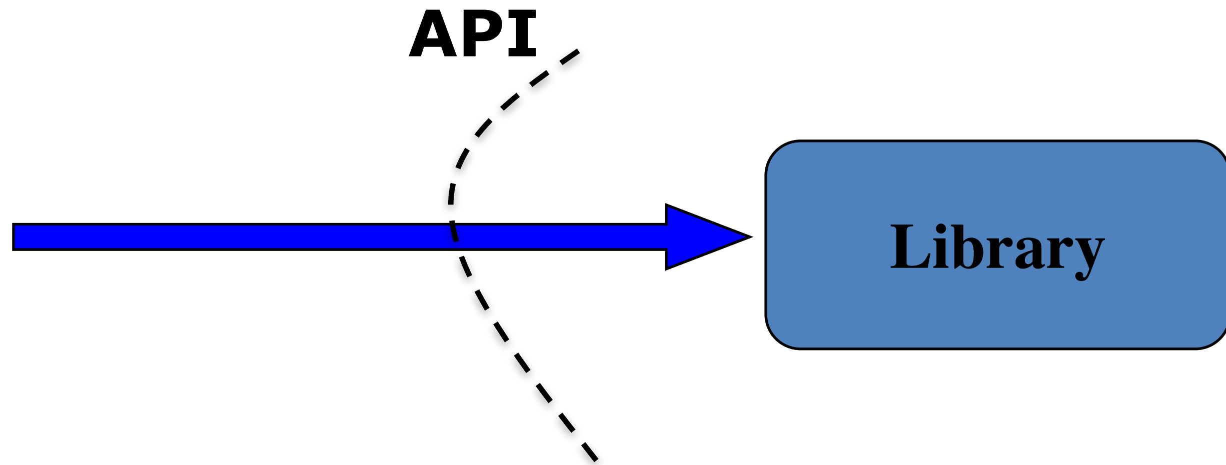
Today's topic API Design

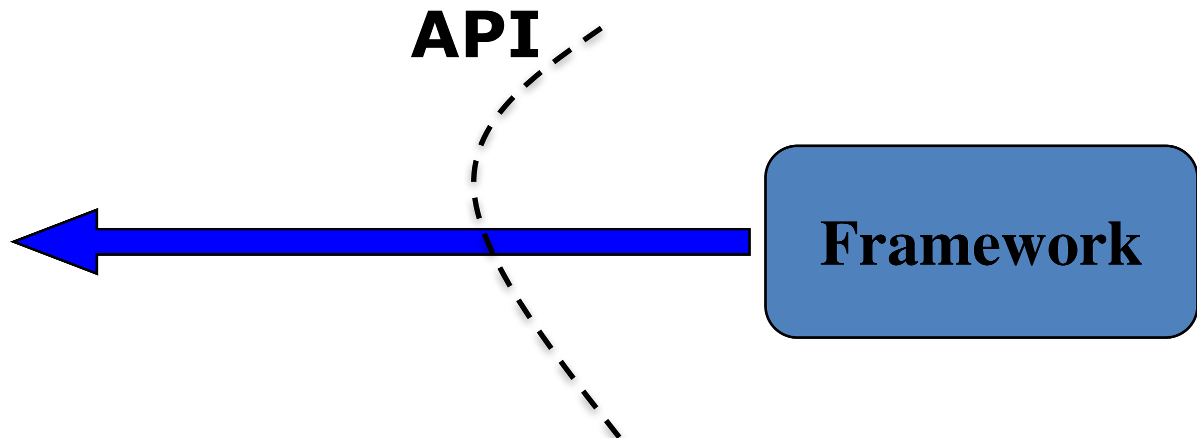
What is an API?

- Short for Application Programming Interface
- A component specification in terms of its operations, their inputs, and their outputs
  - Defines a set of functionalities independent of implementation
  - Allows implementation to vary without compromising clients
- Defines boundary between components in a programmatic system
  - Intermodular boundary
- A *public* API is one designed for use by others

isr institute for SOFTWARE RESEARCH

Libraries, frameworks both define APIs





isr institute for SOFTWARE RESEARCH

Why is API design important?

- APIs can be among your greatest assets
  - Users invest heavily: acquiring, writing, learning
  - Cost to **stop** using an API can be prohibitive
  - Successful public APIs capture users
- Can also be among your 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
  - Good reusable modules are an asset
  - Once module has users, can't change API at will
- 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

- The Process of API Design
- General Principles
- Class Design
- Method Design
- Exception Design

Gather requirements–skeptically

- 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 & more rewarding to build more general API

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

Sample early API draft

```
// A collection of elements (root of the collection hierarchy)
public interface Collection<E> {

    // Ensures that collection contains o
    boolean add(E o);

    // Removes an instance of o from collection, if present
    boolean remove(Object o);

    // Returns true iff collection contains o
    boolean contains(Object o) ;

    // Returns number of elements in collection
    int size() ;

    // Returns true if collection is empty
    boolean isEmpty();

    ...  // Remainder omitted
}
```

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]

isr institute for SOFTWARE RESEARCH

Write 3 implementations of each abstract class or interface before release

- If you write 1, it probably won't support another
- If you write 2, it will support more with difficulty
- If you write 3, it will work fine
- Will Tracz calls this "The Rule of Threes
  - *Confessions of a Used Program Salesman*, Addison-Wesley, 1995)

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

Outline

- The Process of API Design
- General Principles
- Class Design
- Method Design
- Exception Design

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<T>`
- Bad: `DynAnyFactoryOperations`, `_BindingIteratorImplBase`, `ENCODING_CDR_ENCAPS`, `OMGVMCID`

isr institute for SOFTWARE RESEARCH

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 in APIs are harmful
  - 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
  - Serialized forms, exceptions thrown

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 optimized 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

```
for (List<Integer> proposedSolution : Permutations.of(digits))
    if (isSolution(proposedSolution))
        solutions.add(proposedSolution);
```


isr institute for SOFTWARE RESEARCH

Grammar is a part of naming

- Nouns for classes
  - `BigInteger`, `PriorityQueue`
- Nouns or adjectives for interfaces
  - `Collection`, `Comparable`
- Nouns, linking verbs or prepositions for non-mutative methods
  - `size`, `isEmpty`, `plus`
- Action verbs for mutative methods
  - `put`, `add`, `clear`

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 thread safety
- If class is mutable, document state space

institute for SOFTWARE RESEARCH

Consider performance consequences

- 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

isr institute for SOFTWARE RESEARCH

Performance effects of a bad API decisions can be 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
  - `getX()`, `getY()`

isr institute for SOFTWARE RESEARCH

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, functional interfaces
- Know and avoid API traps and pitfalls
  - Finalizers, public static final arrays, etc.
- **Don’t transliterate APIs**

Outline

- The Process of API Design
- General Principles
- Class Design
- Method Design
- Exception 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`

isr institute for SOFTWARE RESEARCH

Subclass only where it makes sense

- Subclassing implies substitutability (Liskov)
  - Don't subclass unless an is-a relationship exists
  - Otherwise, use composition
- Don't subclass just to reuse implementation

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

Good: `Set` extends `Collection`

isr institute for SOFTWARE RESEARCH

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 final

Bad: Many concrete classes in J2SE libraries

Good: `AbstractSet`, `AbstractMap`

isr institute for SOFTWARE RESEARCH

Outline

- The Process of API Design
- General Principles
- Class Design
- Method Design
- Exception 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)); // Does actual writing
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```


isr institute for SOFTWARE RESEARCH

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();
}
```


institute for SOFTWARE RESEARCH

Here's what happens if you violate both of the last two rules at the same time

```
// Spec says: "Skips over n bytes of data from this input."
// But this is a a lie. Ignore return value at your peril!
public long skip(long n) throws IOException;
```

The *only* correct way to use `InputStream.skip`:

```
static void skipFully(InputStream in, long nBytes)
        throws IOException {
    long remaining = nBytes;
    while (remaining != 0) {
        long skipped = in.skip(remaining);
        if (skipped == 0)  // EOF
            throw new EOFException();
        remaining -= skipped;
    }
}
```


isr institute for SOFTWARE RESEARCH

APIs should fail fast: report errors as soon as possible

- Compile time is best – static typing, generics

```
public int max(int... args);
public int max(int first, int... rest);
```

- 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);
}
```


isr institute for SOFTWARE RESEARCH

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
- **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<E> c); // Uses natural ordering
public TreeSet(SortedSet<E> s);  // Uses ordering from s
```

Use appropriate parameter & 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 and run, but misbehave!
- Techniques for shortening parameter lists
  - Break up method
  - Create helper class to hold parameters
  - Builder Pattern

```
// 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();
}
```

Outline

- The Process of API Design
- General Principles
- Class Design
- Method Design
- Exception Design

Throw exceptions to indicate exceptional conditions

- Don’t force client to use exceptions 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);
  }
}
```

- Conversely, don’t fail silently

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

Favor unchecked exceptions

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


institute for SOFTWARE RESEARCH

Include failure-capture information in exceptions

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

API Design Summary

- A good API is a blessing; a bad one a curse
- API Design is hard, but you can't escape it
  - Accept the fact that we all make mistakes
  - Use your APIs as you design them
  - Get feedback from others
- This talk covered some heuristics of the craft
  - Don't adhere to them slavishly, but...
  - Don't violate them without good reason

Outline

- The Process of API Design
- General Principles
- Class Design
- Method Design
- Exception Design
- Two API refactorings (but we're out of time)
  - We didn't go over this section in class

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 to use without documentation

institute for SOFTWARE RESEARCH

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));
```


isr institute for SOFTWARE RESEARCH

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());
```