Principles of Software Construction: Objects, Design, and Concurrency

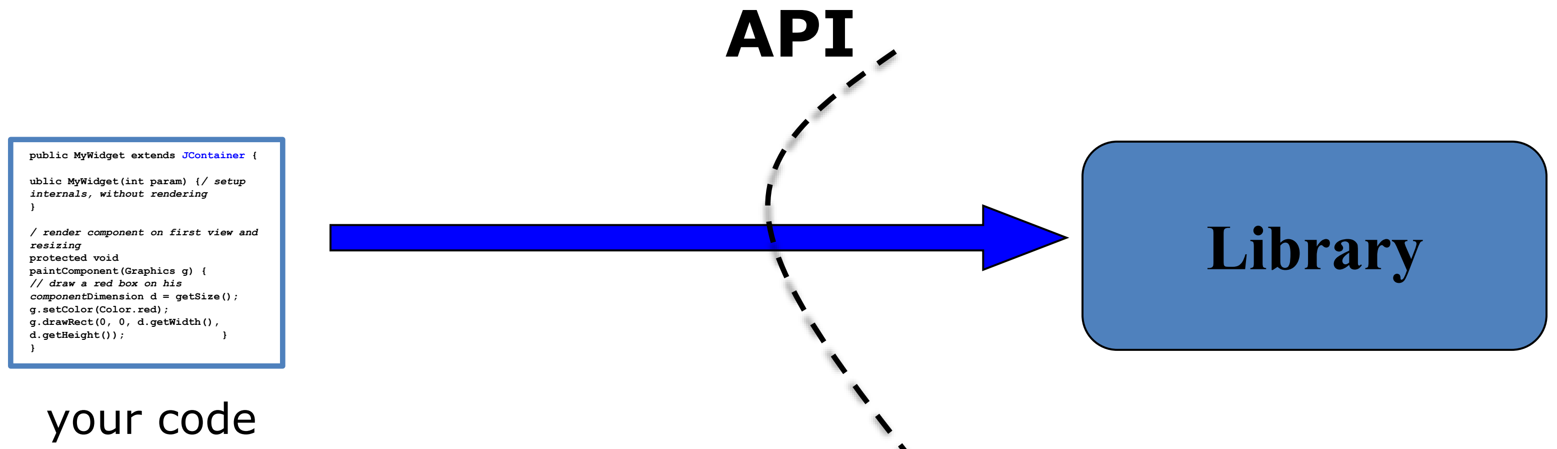
API Design 1: process and naming

Josh Bloch Charlie Garrod

Administrivia

- Homework 4b due Today (11:59 PM)

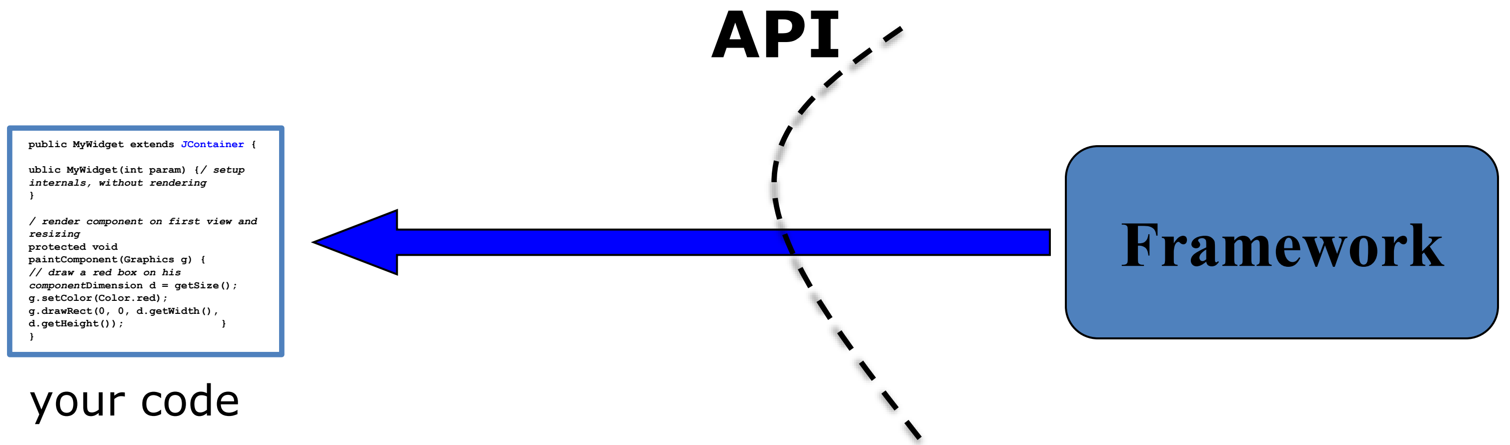
Review: libraries, frameworks both define APIs



```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                   }
}
```



```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                   }
}
```

Today’s topic: API Design

Review: what is an API?

- Short for Application Programming Interface
- Component specification in terms of operations, inputs, & outputs
  - Defines a set of functionalities independent of implementation
- Allows implementation to vary without compromising clients
- Defines component boundaries in a programmatic system
- A *public* API is one designed for use by others
  - Related to Java’s `public` modifier, but not identical
  - `protected` members are part of the public api

Exponential growth in the power of APIs

This list is approximate and incomplete, but it tells a story

’50s-’60s – Arithmetic. Entire library was 10-20 calls!

’70s – `malloc`, `bsearch`, `qsort`, `rnd`, I/O, system calls, formatting, early databases

’80s – GUIs, desktop publishing, relational databases

’90s – Networking, multithreading

’00s – **Data structures(!)**, higher-level abstractions, Web APIs: social media, cloud infrastructure

’10s – Machine learning, IOT, pretty much everything

What the dramatic growth in APIs has done for us

- Enabled code reuse on a grand scale
- Increased the level of abstraction dramatically
- A single programmer can quickly do things that would have taken months for a team
- What was previously impossible is now routine
- APIs have given us super-powers

Why is API design important?

- A good API is a joy to use; a bad API is a nightmare
- APIs can be among your greatest assets
  - Users invest heavily: learning, using
  - Cost to **stop** using an API can be prohibitive
  - Successful public APIs capture users
- APIs can also be among your greatest liabilities
  - Bad API can cause unending stream of support requests
  - 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 a module has users, you can’t change its 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
- Naming

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**
- You may get requirements that don't make sense
  - Ask questions until you see eye-to-eye
- You may get requirements that are wrong
  - Push back
- You may get requirements that are contradictory
  - Broker a compromise
- Requirements *will* change as you proceed

Requirements gathering (2)

- Key question: what problems should this API solve?
  - Defines scope of effort
- Also important: what problems shouldn't API solve?
  - Bounds effort
- Requirements can include performance, scalability
  - These factors can (but don't usually) constrain API
- Maintain a **requirements doc**
  - Helps focus effort, fight scope creep
  - Saves rationale for posterity

An often overlooked part of requirements gathering

- Ask yourself if the API **should** be designed (*inception*)
- Here are several good reasons **not** to design it
  - It's superfluous
  - It's impossible
  - It's unethical
  - The requirements are too vague
- If any of these things are true, **now** is the time to raise red flag
- If the problem can't be fixed, fail fast!
  - The longer you wait, the more costly the failure

Choosing an abstraction (model)

- The key: embed use cases in an underlying structure
  - Note their similarities and differences
  - Note similarities to physical objects (“reasoning by analogy”)
  - Note similarities to other abstractions in the same platform
- This step does not have to be explicit
  - You can start designing the spec without a clear model
  - Generally a model will emerge
- For easy APIs, this step is almost nonexistent
  - It can be as simple as deciding on static method vs. instantiable class
- For difficult APIs, can be the hardest part of the process

Model examples

- **Collections Framework** - Core collection interfaces describe data aggregates. Many implementations are provided including decorators. Polymorphic algorithms are provided to operate on collections independent of their representation.
- **Swing and other GUI APIs** -Trees of components representing GUI elements are arranged by layouts. Events are delivered to listeners, which modify components.
- **`java.io`** - Input and output stream interfaces, and many implementations, including decorators (which implement a stream backed by another).

Start with short spec – 1 page is ideal

- **At this stage, comprehensibility and agility are more important than completeness**
- Bounce spec off as many people as possible
  - Start with a small, select group and enlarge over time
  - Listen to their input and take it seriously
  - **API Design is not a solitary activity!**
- If you keep the spec short, it's easy to read, modify, or scrap it and start from scratch
- **Don't fall in love with your spec too soon!**
- Flesh it out (only) as you gain confidence in it

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
  - If you haven't written code to it, it probably doesn't work
- 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]

When you think you're on the right track, *then* write a prototype implementation

- Some of your client code will run; some won't
- You will find "embarrassing" errors in your API
  - They are obvious only in retrospect
  - Fix them and move on
- You may also find subtle performance problems
  - I found an inherently quadratic API at this state
  - Which is why we have `AbstractList.removeRange`

Then flesh out documentation so it’s usable by people who didn’t help you write the API

- Now you have an artifact you can share more widely
- Distribute, but ensure people know it’s subject to change
- If you’re lucky, you’ll get bug reports & feature requests
- Use the API feedback while you can

Try API on at least 3 use cases before release

- If you write one, it probably won’t support another
- If you write two, it will support more with difficulty
- If you write three, it will probably work fine
- Ideally, get different people to write the use cases
  - This will test documentation & give you different perspectives
- This is even more important for plug-in APIs
- Ted Biggerstaff calls this “The Rule of Threes”
  (Will, Tracz, Confessions of a Used Program Salesman, Addison-Wesley, 1995)

Maintain realistic expectations

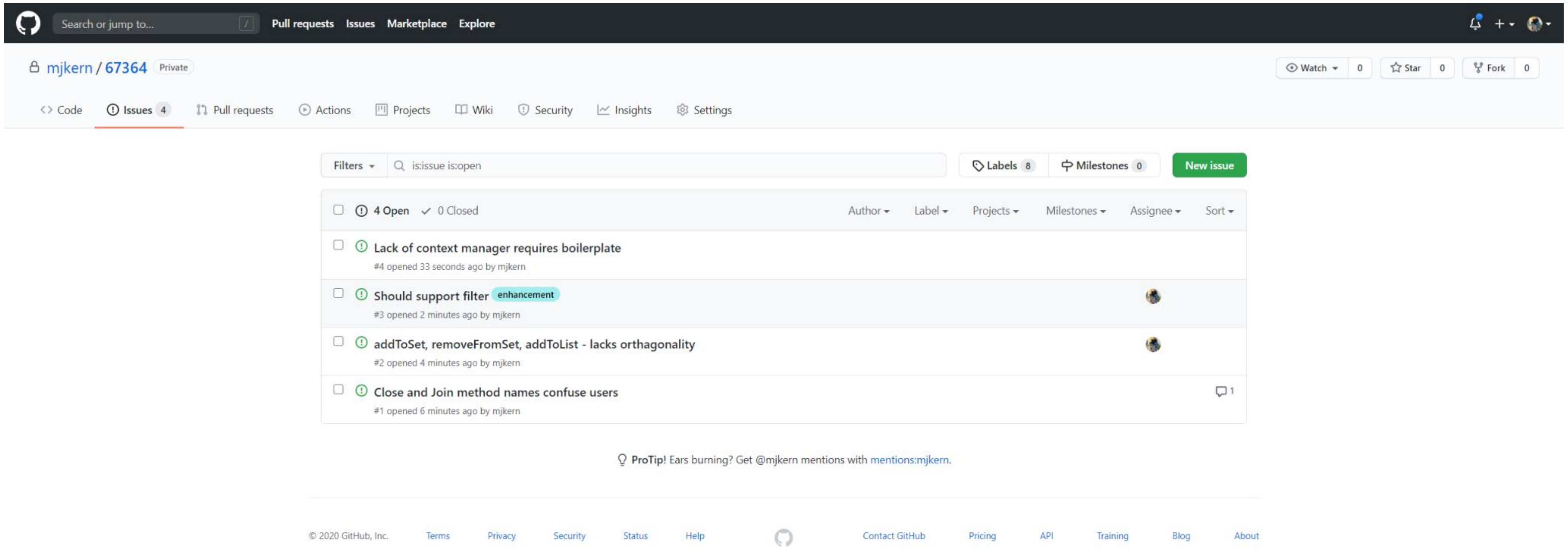
- Most API designs are over-constrained
  - You won't be able to please everyone – don’t try!
    - If you do, you’ll end up with a messy, bloated API
  - Come up with a unified, coherent design that represents a compromise
  - It can be hard to decide which “requirements” are important
- Expect to make mistakes
  - Real-world use will flush them out
  - Expect to evolve API

Issue tracking

- Throughout process, maintain a list of design issues
  - Individual decisions such as what input format to accept
    - **Write down all the options**
    - **Say which were ruled out and why**
    - **When you decide, say which was chosen and why**
- Prevents wasting time on solved issues
- Provides rationale for the resulting API
  - Reminds its creators
  - Enlightens its users
- I used to use text files and mailing lists for this, but now there are tools (github, Jira, Bugzilla, etc.)

Sample tracker-based issues list example

Github's issue tracker



Lack of context manager requires boilerplate
#4 opened 33 seconds ago by mjkern

Should support filter `enhancement`
#3 opened 2 minutes ago by mjkern

addToSet, removeFromSet, addToList - lacks orthagonality
#2 opened 4 minutes ago by mjkern

Close and Join method names confuse users 1
#1 opened 6 minutes ago by mjkern

ProTip! Ears burning? Get @mjkern mentions with mentions:mjkern.

Key design artifacts

1. **Requirements document**
2. **Issues list**
3. **Use-case code**

Maintain throughout design and retain when done

- They guide the design process
- When API is done, they're the basis of the **design rationale**
  - Public explanation for design
  - e.g., https://docs.oracle.com/javase/8/docs/technotes/guides/collections/designfaq.html

The process of API design – Summary

Not sequential; if you discover shortcomings, iterate!

1. **Gather requirements** skeptically, including *use cases*
2. **Choose an abstraction** (model) that appears to address use cases
3. **Compose a short API sketch** for abstraction
4. **Apply API sketch to use cases** to see if it works
   - If not, fix API sketch, or go back to step 3, 2, or even 1.
5. **Show API** to anyone who will look at it
6. **Write prototype** implementation of API
7. **Flesh out** the documentation & harden implementation
8. **Keep refining it** as long as you can

Disclaimer – one size does not fit all

- This process has worked for me
- Others developed similar processes independently
- But I'm sure there are other ways to do it
- The smaller the API, the less process you need

Puzzler: “Big Trouble”

```
public static void main(String [] args) {
   BigInteger fiveThousand = new BigInteger("5000");
   BigInteger fiftyThousand = new BigInteger("50000");
   BigInteger fiveHundredThousand = new BigInteger("500000");

   BigInteger total = BigInteger.ZERO;
   total.add(fiveThousand);
   total.add(fiftyThousand);
   total.add(fiveHundredThousand);

   System.out.println(total);
}
```

What Does It Print?

```
public static void main(String [] args) {
   BigInteger fiveThousand = new BigInteger("5000");
   BigInteger fiftyThousand = new BigInteger("50000");
   BigInteger fiveHundredThousand = new BigInteger("500000");

   BigInteger total = BigInteger.ZERO;
   total.add(fiveThousand);
   total.add(fiftyThousand);
   total.add(fiveHundredThousand);

   System.out.println(total);
}
```

What Does It Print?

(a) `0`

(b) `500000`

(c) `555000`

(d) It varies

`BigInteger` is immutable!

Another Look

```
public static void main(String [] args) {
   BigInteger fiveThousand = new BigInteger("5000");
   BigInteger fiftyThousand = new BigInteger("50000");
   BigInteger fiveHundredThousand = new BigInteger("500000");

   BigInteger total = BigInteger.ZERO;
   total.add(fiveThousand);          // Ignores result
   total.add(fiftyThousand);         // Ignores result
   total.add(fiveHundredThousand);   // Ignores result

   System.out.println(total);
}
```

How do you fix it?

```
public static void main(String [] args) {
   BigInteger fiveThousand = new BigInteger("5000");
   BigInteger fiftyThousand = new BigInteger("50000");
   BigInteger fiveHundredThousand = new BigInteger("500000");

   BigInteger total = BigInteger.ZERO;
   total = total.add(fiveThousand);
   total = total.add(fiftyThousand);
   total = total.add(fiveHundredThousand);

   System.out.println(total);
}
```

Prints 555000

The moral

- Blame the API designer
  - (In fairness, this was my first OO API, 1996)
- Names like `add`, `subtract`, `negate` suggest mutation
- Better names: `plus`, `minus`, `negation`
- Generally (and loosely) speaking:
  - Action verbs for mutation
  - Prepositions, linking verbs, nouns, or adjectives for pure functions
- **Names are important!**

Outline

- The Process of API Design
- Naming

Names Matter – API is a little language

Naming is perhaps the single most important factor in API usability

- Primary goals
  - **Client code should read like prose** (“easy to read”)
  - **Client code should mean what it says** (“hard to misread”)
  - **Client code should flow naturally** (“easy to write”)
- To that end, names should:
  - be largely self-explanatory
  - leverage existing knowledge
  - interact harmoniously with language and each other

The easy part: typographical naming conventions

*The **language specification** demands that you follow these*

- Package or module – `org.junit.jupiter.api`, `com.google.common.collect`
- Class or Interface – `Stream`, `FutureTask`, `LinkedHashMap`, `HttpClient`
- Method or Field – `remove`, `groupingBy`, `getCrc`
- Parameter – `numerator`, `modulus`
- Constant Field – `MIN_VALUE`, `NEGATIVE_INFINITY`
- Type Parameter – `T`, `E`, `K`, `V`, `X`, `R`, `U`, `V`, `T1`, `T2`

How to choose names that are easy to read & write

- Choose key nouns carefully!
  - Related to finding good abstractions, which can be hard
  - If you *can't* find a good name, it's generally a bad sign
- If you get the key nouns right, other nouns, verbs, and prepositions tend to choose themselves
- Names can be literal or metaphorical
  - Literal names have literal associations
    - e.g., **matrix** suggests inverse, determinant, eigenvalue, etc.
  - Metaphorical names enable reasoning by analogy
    - Helps you and your users
    - e.g., **mail** suggests send, cc, bcc, inbox, outbox, folder, etc.

Names drive development, for better or worse

- Good names drive good development
- Bad names inhibit good development
- Bad names result in bad APIs unless you take action
- **The API talks back to you. Listen!**

Vocabulary consistency

- Use words consistently throughout your API
  - Never use the same word for multiple meanings
  - Never use multiple words for the same meaning
  - i.e., words should be isomorphic to meanings

Vocabulary consistency as it relates to scope

APIs are actually little language ***extensions***

- The tighter the scope, the more important is consistency
  - **Within APIs, consistency is critical**
  - In related APIs on a platform, it’s highly desirable
  - Across the platform, it’s desirable
  - Between platforms, it’s nice-to-have
- **If forced to choose between local & platform consistency, choose local**
- But look to platform libraries for vocabulary
  - Ignoring obsolete and unpopular libraries
- Finally, look to similar APIs on other platforms for naming ideas

Avoid abbreviations except where customary

- Back in the day, storage was scarce & people abbreviated everything
  - Some continue to do this by force of habit or tradition
- Ideally, use complete words
- But sometimes, names just get too long
  - If you must abbreviate, do it tastefully
  - **No excuse for cryptic abbreviations**
- Of course you should use `gcd`, `Url`, `cos`, `mba`, etc.

Grammar is a part of naming too

- 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`
- If you follow these, they quickly become second nature

Names should be regular – strive for symmetry

- If API has 2 verbs and 2 nouns, support all 4 combinations
  - Unless you have a very good reason not to
- Programmers will try to use all 4 combinations
  - They will get upset if the one they want is missing
- In other words, good APIs are generally *orthogonal*

```
addRow        removeRow
addColumn     removeColumn
```

Don’t mislead your user

- Names have implications
  - Learn them and uphold them in your APIs
- **Don’t violate *the principle of least astonishment***
- Ignore this advice at your own peril
  - Can cause unending stream of subtle bugs

```
public static boolean interrupted()
```

Tests whether the current thread has been interrupted.
The interrupted status of the thread is cleared by this method....

Don’t lie to your user

- Name method for what it does, not what you wish it did
- If you can’t bring yourself to do this, fix the method!
- Again, ignore this at your own peril

```
public long skip(long n) throws IOException
```

Skips over and discards n bytes of data from this input stream. The skip method may, for a variety of reasons, end up skipping over some smaller number of bytes, possibly 0. This may result from any of a number of conditions; reaching end of file before n bytes have been skipped is only one possibility. The actual number of bytes skipped is returned…

Good naming takes time, but it's worth it

- Don't be afraid to spend hours on it; I do.
  - And I still get the names wrong sometimes
- **Don't just list names and choose**
  - Think about goals and anti-goals for names
  - Think of names consistent with these goals
  - Write out realistic client code and compare
- Discuss names with colleagues; it really helps.

Lecture summary

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

To be continued…