Principles of Software Construction

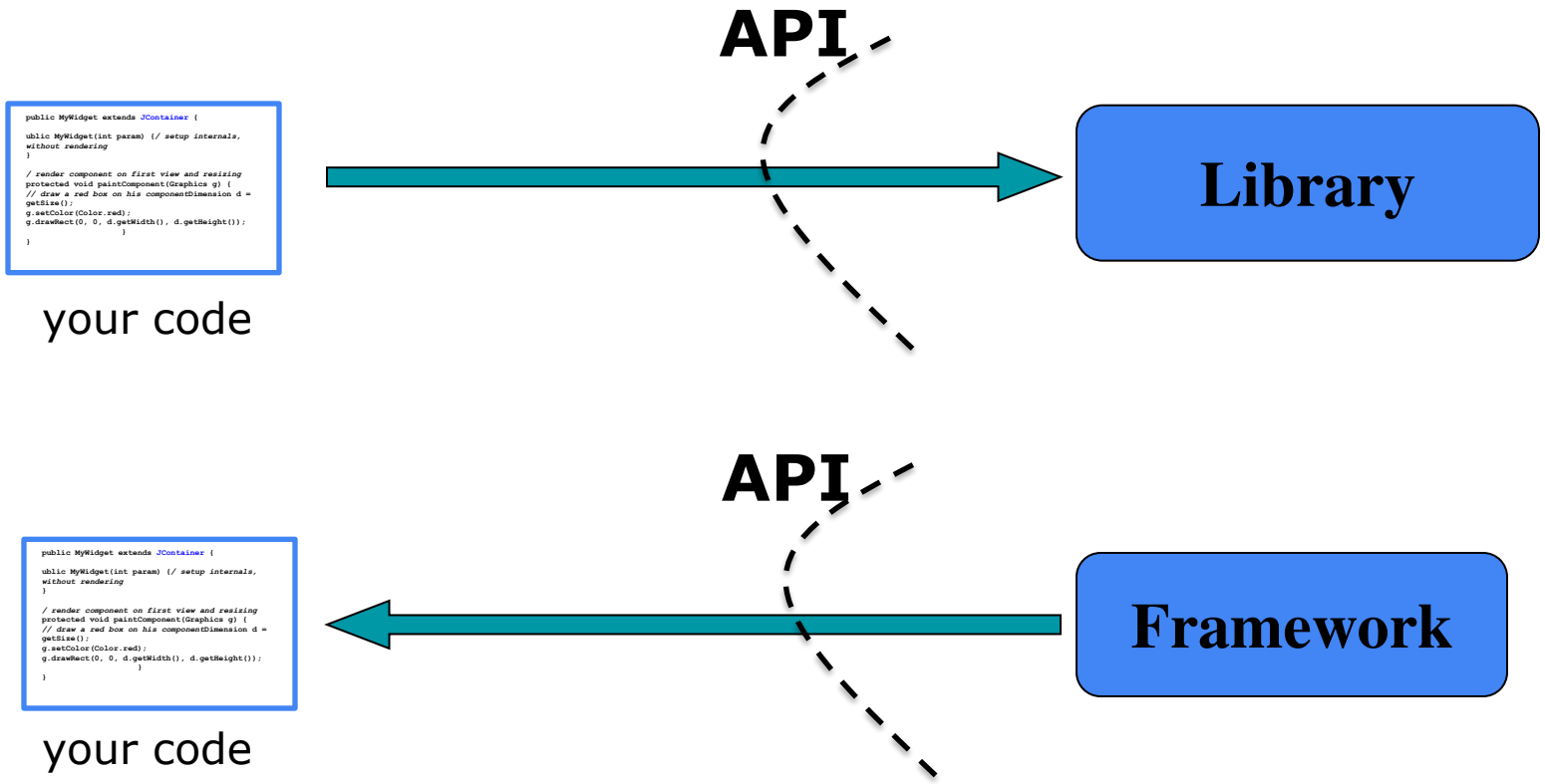
API Design

Christian Kästner Vincent Hellendoorn
(Many slides originally from Josh Bloch)

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Review: libraries, frameworks both define APIs



Upcoming

Midterm 2 on Thursday

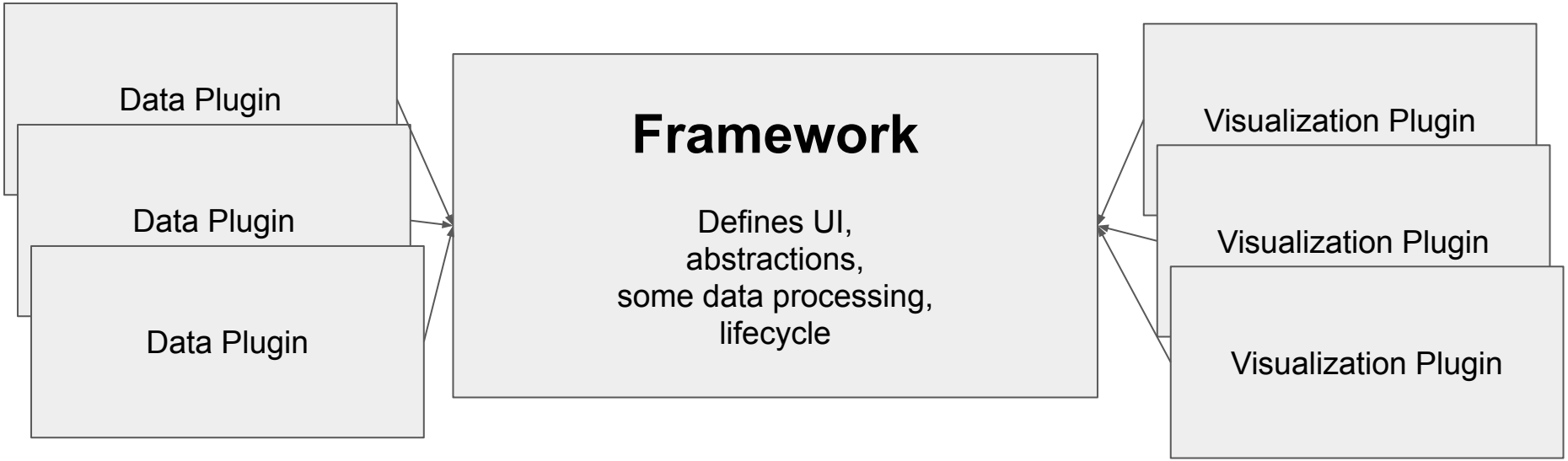
- 4 sheets of notes, handwritten or printed, both sides
- all topics in scope, focus on topics since midterm 1

Final homework released after midterm

- Milestones: (1) Design framework,
  (2) implement framework, (3) implement plugins
- Work with a partner (or two)

Homework 6

Data Analytics Framework



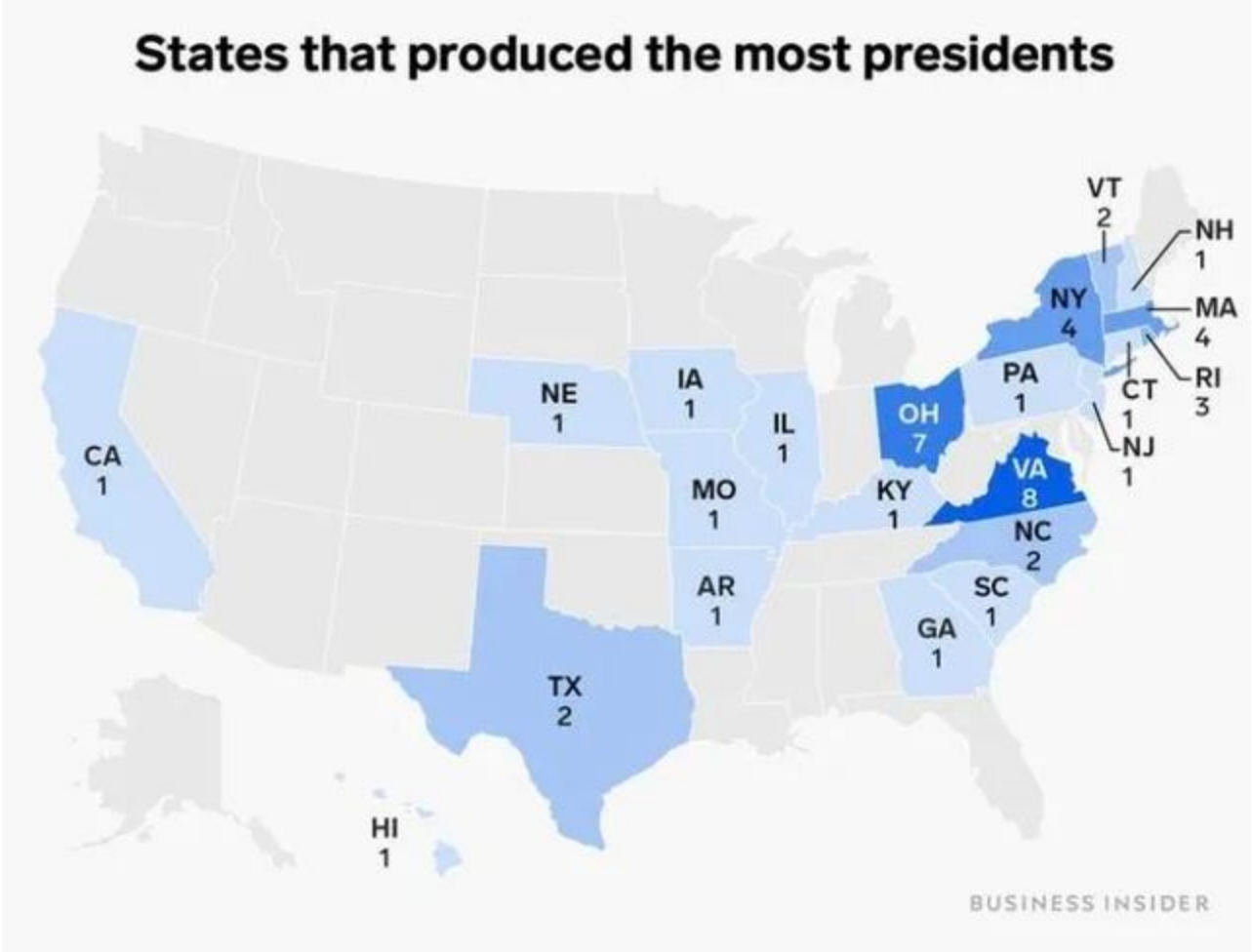
HW6: Map-Based Data Visualizations?

State, county, or country data

Data from many sources

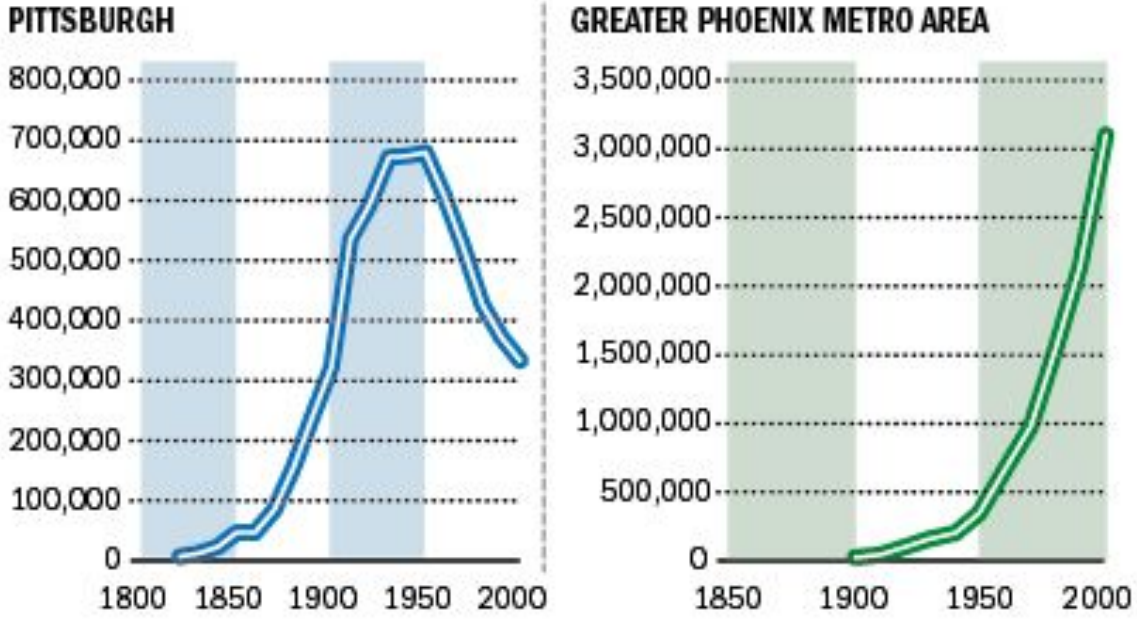
Visualization as map image, table, google maps

Animations for time series data

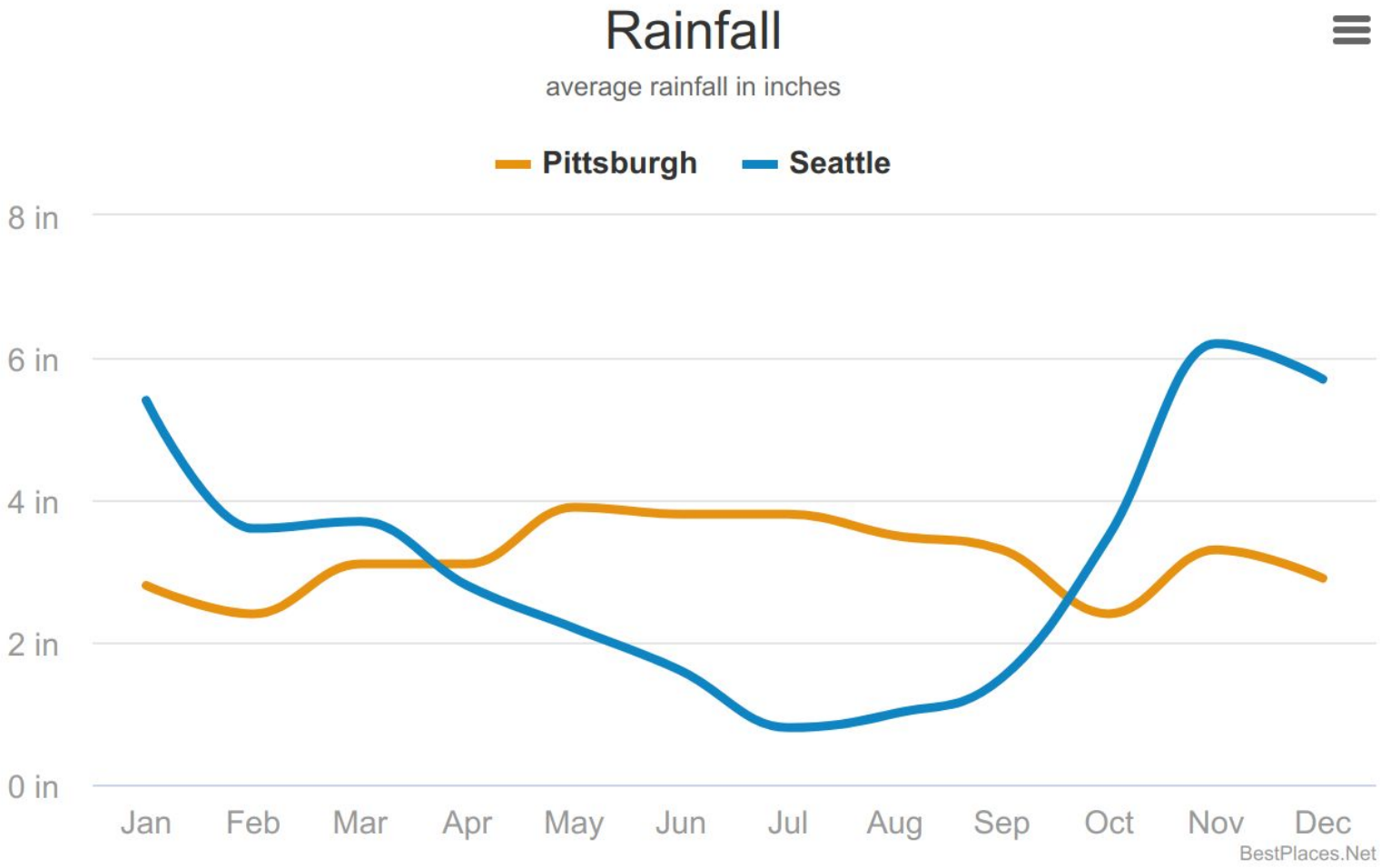


Population trends: Pittsburgh and Phoenix

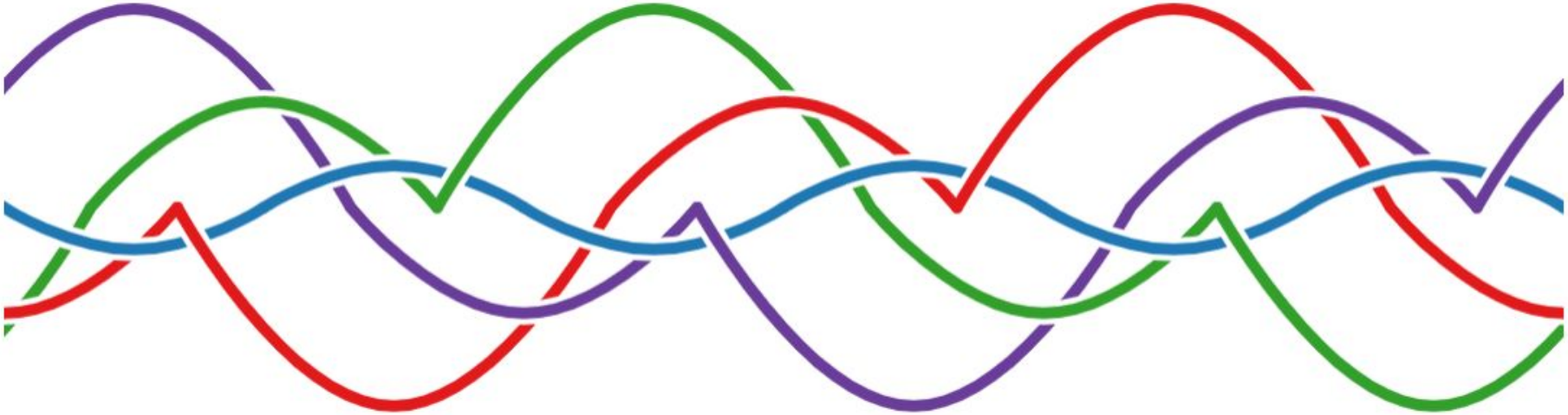
Population trends in Pittsburgh and the greater Phoenix metropolitan area (roughly Maricopa County) over the past 150-200 years.



James Hilston/Post-Gazette


Rainfall
average rainfall in inches
Pittsburgh
Seattle
8 in
6 in
4 in
2 in
0 in
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
BestPlaces.Net

345



plotly | Graphing Libraries

Search...

- Quick start
- Examples
  - Fundamentals
  - Basic Charts
  - Statistical Charts
  - Scientific Charts
  - Financial Charts
  - Maps
  - 3D Charts
  - Subplots
  - Chart Events
  - Animations


Waterfall Charts

Indicators

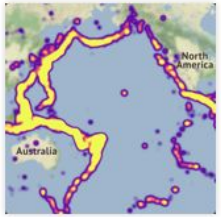

Candlestick Charts


Funnel and Ch

HW6: Consider plotting libraries (for web frontends) to brainstorm ideas

Maps


Mapbox Map Layers



Mapbox Density Heatmap

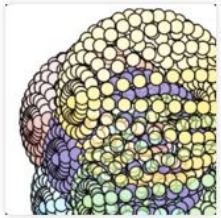

Choropleth Mapbox

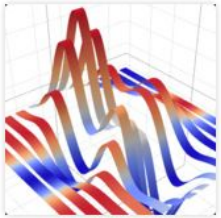

Lines on Maps

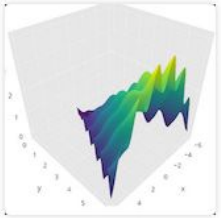

Bubble Maps

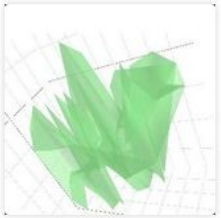
3D Charts

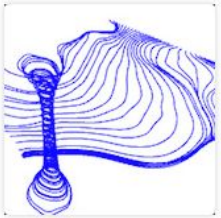
More 3D Charts »


3D Scatter Plots


Ribbon Plots

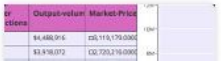

3D Surface Plots


3D Mesh Plots


3D Line Plots

Subplots

More Subplots »



Leftover topics

ReactJS (see last week's slides)

Where we are

Design for	*Small scale:* One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
understanding	Subtype Polymorphism	Domain Analysis	GUI vs Core
change/ext.	Information Hiding, Contracts	Inheritance & Deleg.	Frameworks and Libraries, **APIs**
reuse	Immutability	Responsibility Assignment, Design Patterns, Antipattern	Module systems, microservices
robustness	Types	Promises/Reactive P.	Testing for Robustness
...	Unit Testing	Integration Testing	CI, DevOps, Teams

Where we are

Design for	*Small scale:* One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
understanding	Subtype Polymorphism ✓	Domain Analysis ✓	GUI vs Core ✓
change/ext.	Information Hiding, Contracts ✓	Inheritance & Del. ✓	Frameworks and Libraries ✓, **APIs**
reuse	Immutability ✓	Responsibility Assignment, Design Patterns, Antipattern ✓	Module systems, microservices
robustness	Types	Promises/ Reactive P. ✓	Testing for Robustness
...	Unit Testing ✓	Integration Testing ✓	CI ✓, DevOps, Teams

Outline

- Introduction to API Design
- The Process of API Design
- Naming

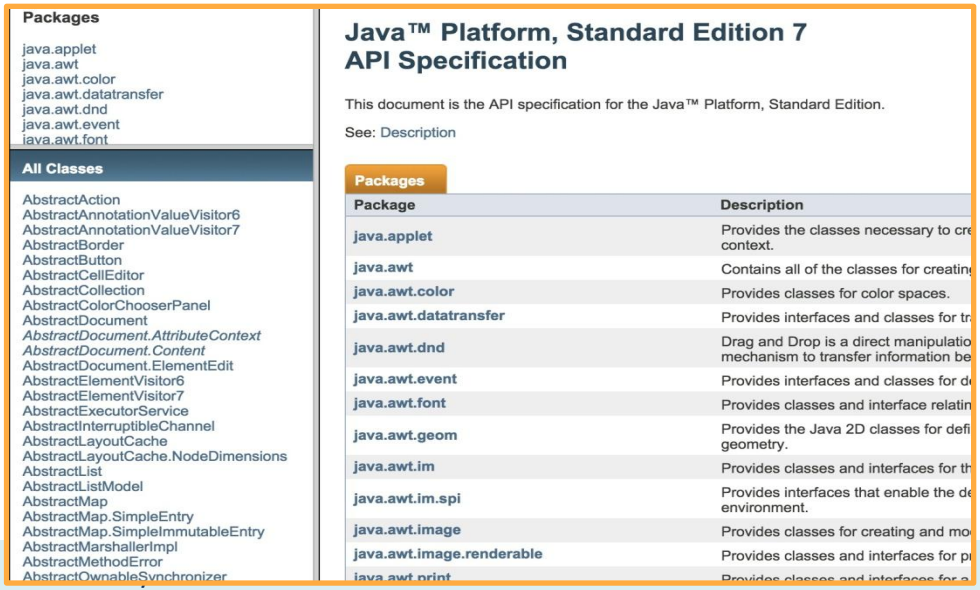
Introduction to API Design

What's an API?

- Short for Application Programming Interface
  - = Contract for a Subsystem/Library
- 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

API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system



Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font

All Classes

AbstractAction
AbstractAnnotationValueVisitor6
AbstractAnnotationValueVisitor7
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer

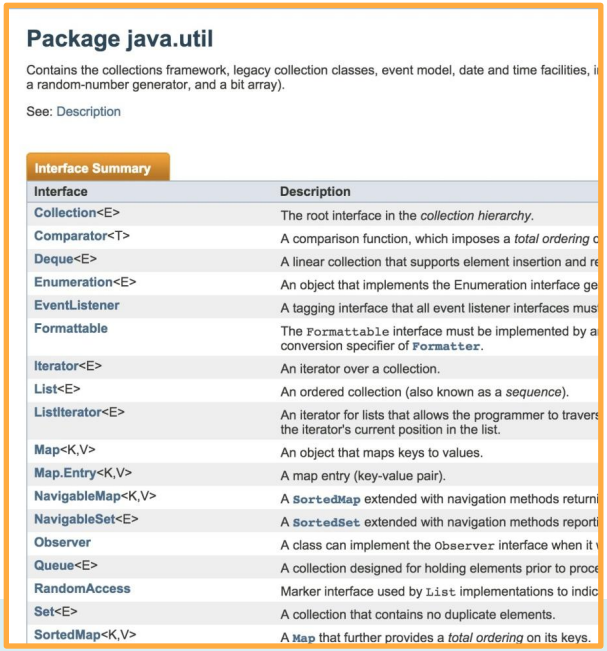
Java™ Platform, Standard Edition 7 API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

Packages

Package	Description
java.applet	Provides the classes necessary to cre context.
java.awt	Contains all of the classes for creatin
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for tr
java.awt.dnd	Drag and Drop is a direct manipulatio mechanism to transfer information be
java.awt.event	Provides interfaces and classes for d
java.awt.font	Provides classes and interface relatin
java.awt.geom	Provides the Java 2D classes for defi geometry.
java.awt.im	Provides classes and interfaces for th
java.awt.im.spi	Provides interfaces that enable the de environment.
java.awt.image	Provides classes for creating and mo
java.awt.image.renderable	Provides classes and interfaces for pi
java.awt.print	Provides classes and interfaces for a



Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, i a random-number generator, and a bit array).

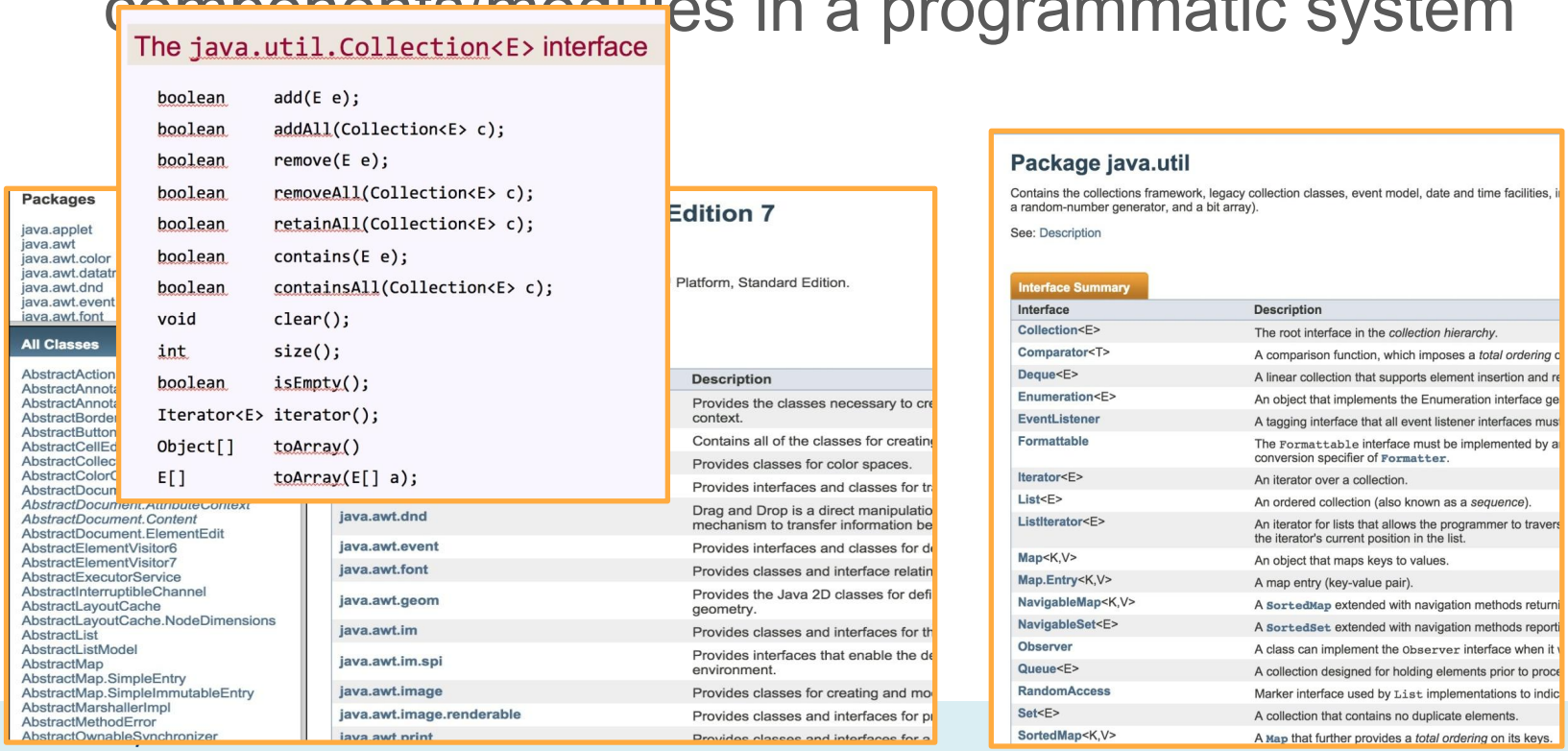
See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* c
Deque<E>	A linear collection that supports element insertion and re
Enumeration<E>	An object that implements the Enumeration interface ge
EventListener	A tagging interface that all event listener interfaces mus
Formattable	The `Formattable` interface must be implemented by a conversion specifier of `Formatter`.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to travers the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
NavigableMap<K,V>	A `SortedMap` extended with navigation methods returni
NavigableSet<E>	A `SortedSet` extended with navigation methods reporti
Observer	A class can implement the `Observer` interface when it v
Queue<E>	A collection designed for holding elements prior to proce
RandomAccess	Marker interface used by `List` implementations to indic
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A `Map` that further provides a *total ordering* on its keys.

API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system



The java.util.Collection<E> interface

```
boolean     add(E e);
boolean     addAll(Collection<E> c);
boolean     remove(E e);
boolean     removeAll(Collection<E> c);
boolean     retainAll(Collection<E> c);
boolean     contains(E e);
boolean     containsAll(Collection<E> c);
void        clear();
int         size();
boolean     isEmpty();
Iterator<E> iterator();
Object[]    toArray()
E[]         toArray(E[] a);
```

Packages

java.applet
java.awt
java.awt.color
java.awt.datatr
java.awt.dnd
java.awt.event
java.awt.font

All Classes

AbstractAction
AbstractAnnota
AbstractAnnota
AbstractBorde
AbstractButton
AbstractCellEd
AbstractCollec
AbstractColorC
AbstractDocun
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer

Edition 7

Platform, Standard Edition.

	Description
	Provides the classes necessary to cre context.
	Contains all of the classes for creatin
	Provides classes for color spaces.
	Provides interfaces and classes for tr
java.awt.dnd	Drag and Drop is a direct manipulatio mechanism to transfer information be
java.awt.event	Provides interfaces and classes for d
java.awt.font	Provides classes and interface relatin
java.awt.geom	Provides the Java 2D classes for defi geometry.
java.awt.im	Provides classes and interfaces for th
java.awt.im.spi	Provides interfaces that enable the de environment.
java.awt.image	Provides classes for creating and mo
java.awt.image.renderable	Provides classes and interfaces for p
java.awt.print	Provides classes and interfaces for a

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, i a random-number generator, and a bit array).

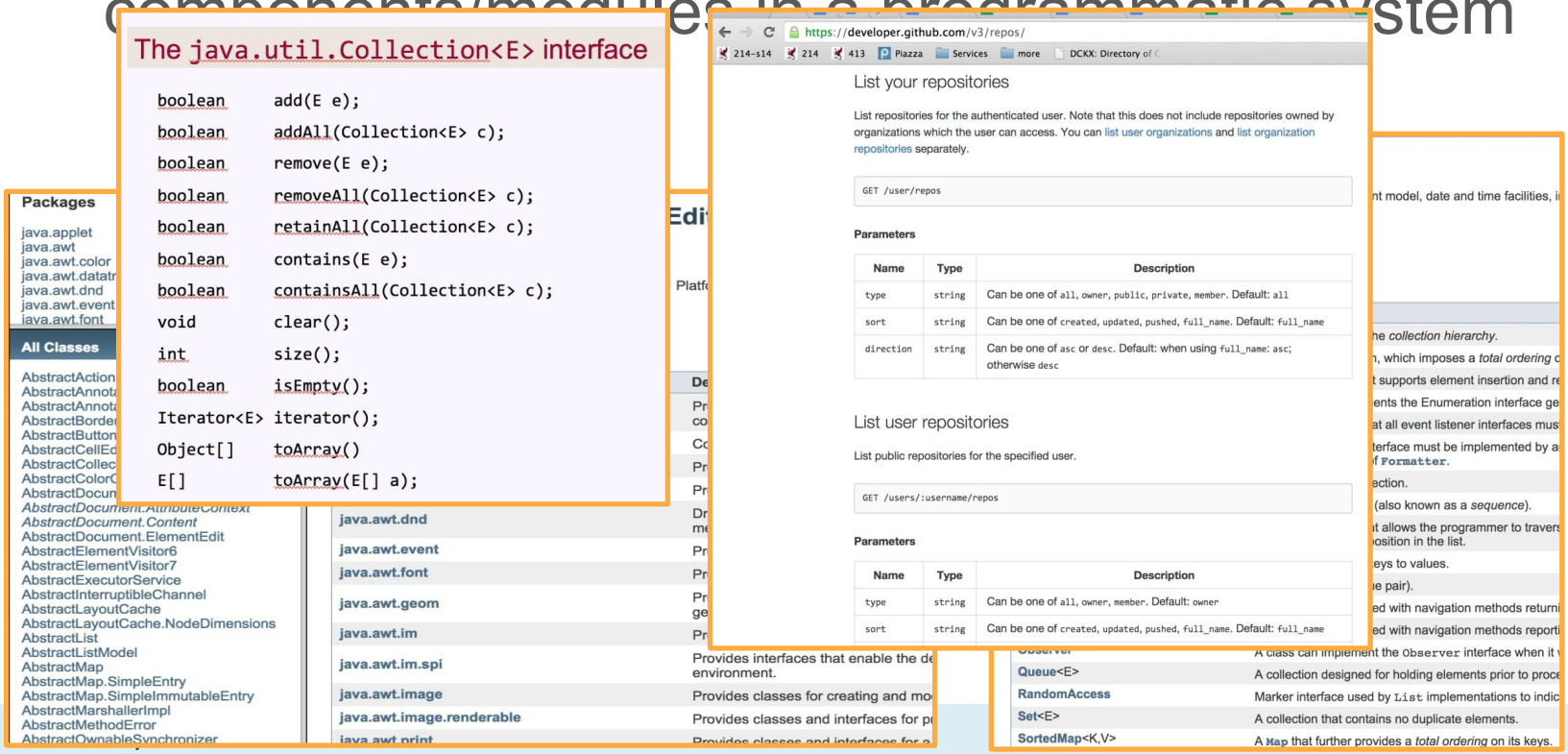
See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* c
Deque<E>	A linear collection that supports element insertion and re
Enumeration<E>	An object that implements the Enumeration interface ge
EventListener	A tagging interface that all event listener interfaces mus
Formattable	The `Formattable` interface must be implemented by a conversion specifier of `Formatter`.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to travers the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
NavigableMap<K,V>	A `SortedMap` extended with navigation methods returni
NavigableSet<E>	A `SortedSet` extended with navigation methods reporti
Observer	A class can implement the `Observer` interface when it v
Queue<E>	A collection designed for holding elements prior to proce
RandomAccess	Marker interface used by `List` implementations to indic
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A `Map` that further provides a *total ordering* on its keys.

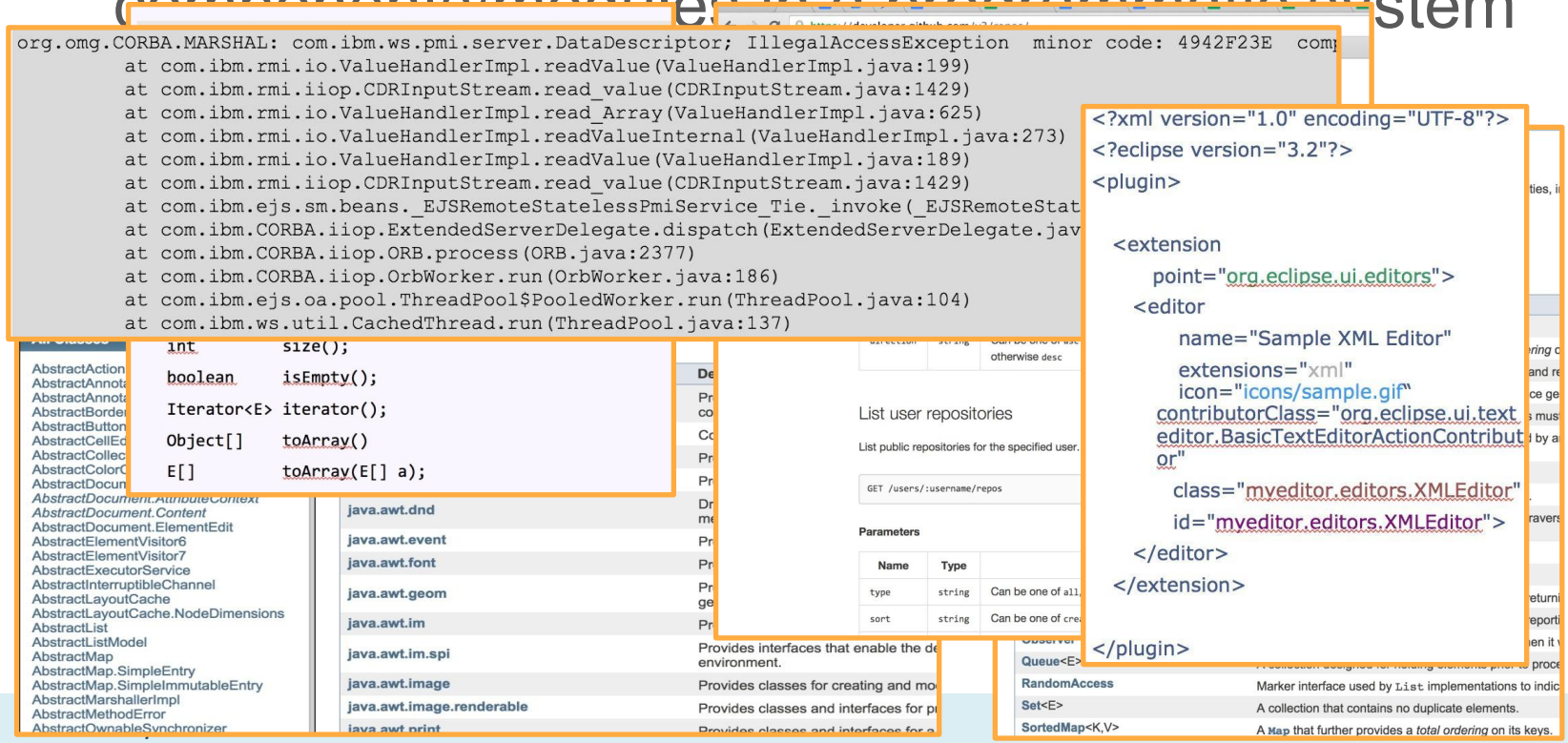
API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system

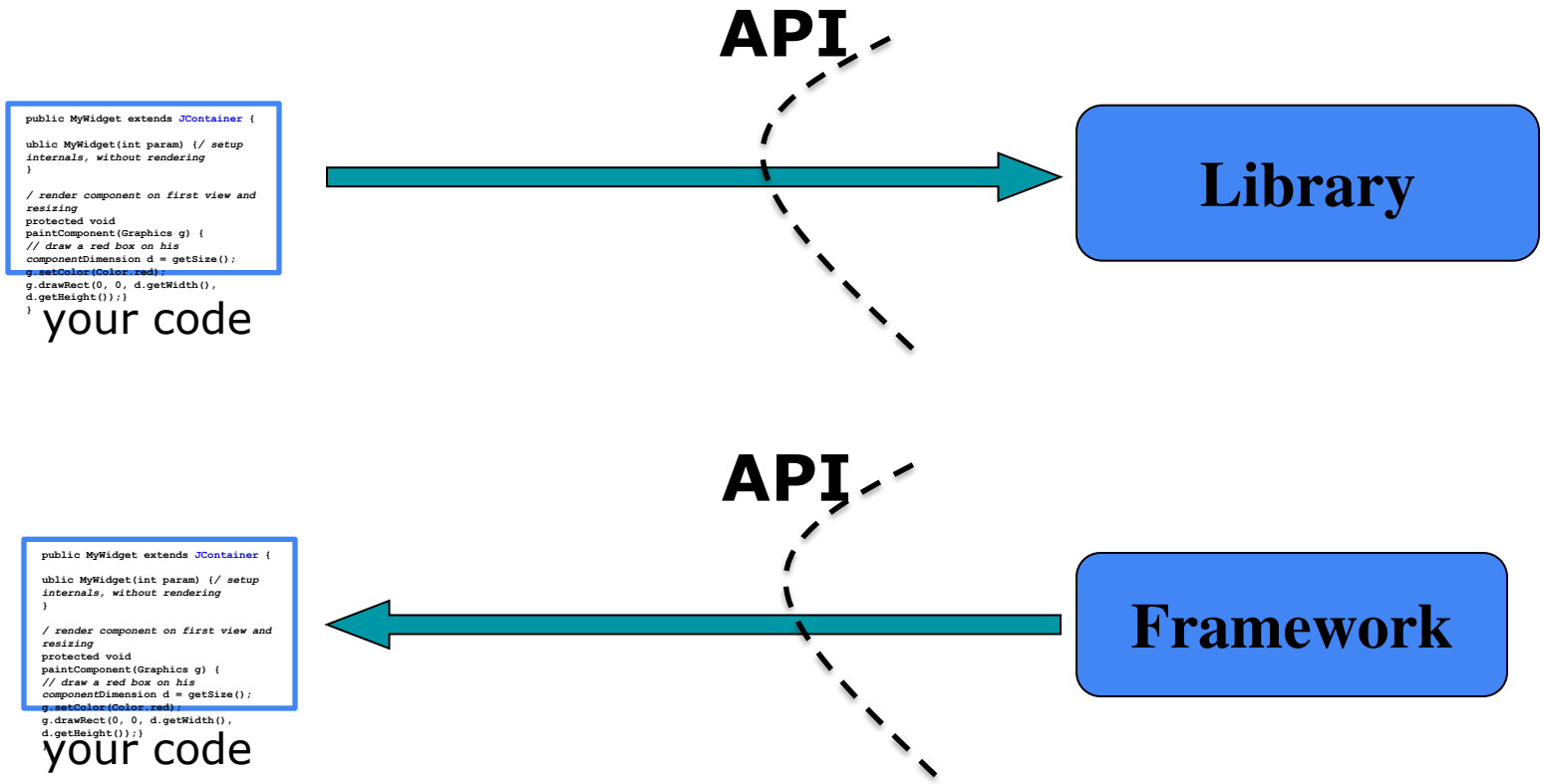


API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system



Libraries and frameworks both define APIs



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 functions!

’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

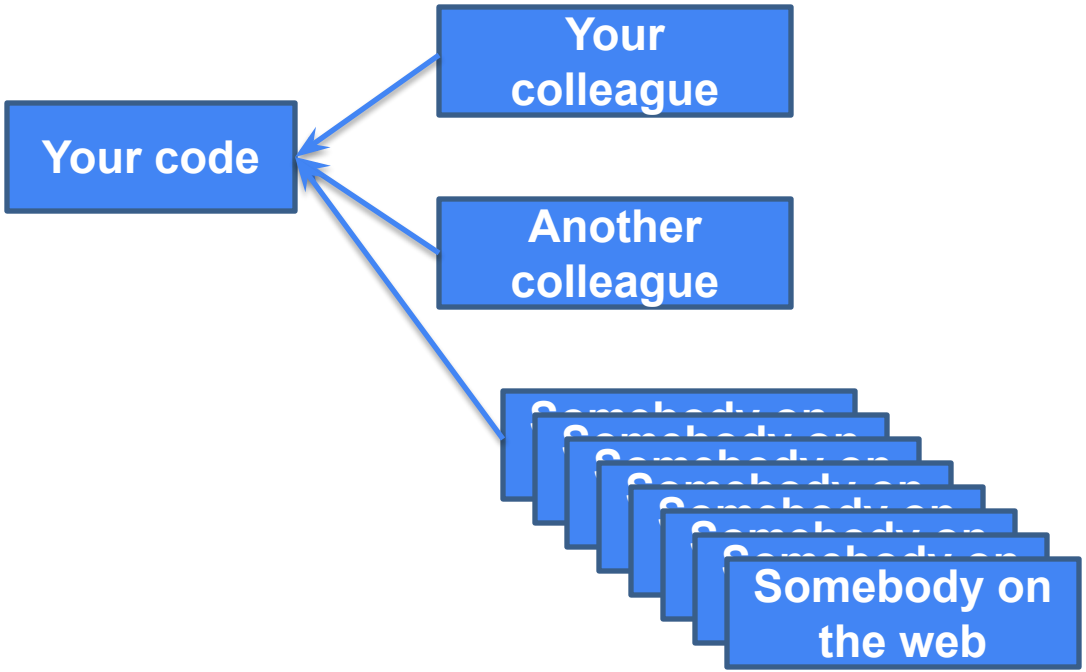
What the dramatic growth in APIs has done for us

- 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

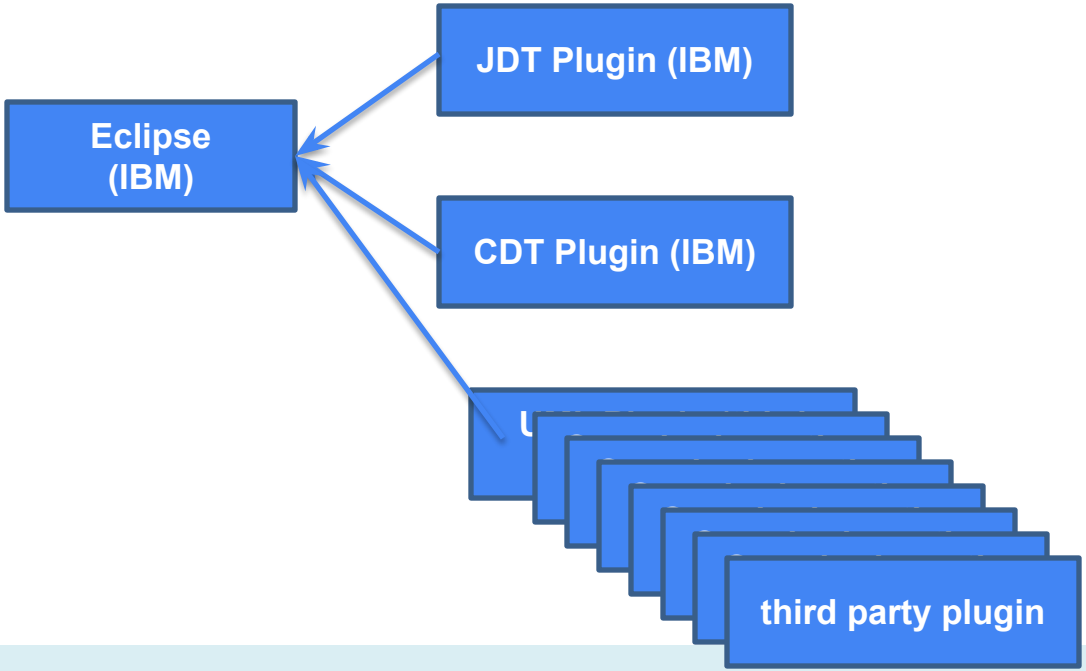
Why is API design important?

- 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**

Public APIs are forever



Public APIs are forever



Evolutionary problems: Public (used) APIs are forever

- "One chance to get it right"
- Can only add features to library
- Cannot:
  - remove method from library
  - change contract in library
  - change plugin interface of framework
- Deprecation of APIs as weak workaround

enable

```
@Deprecated
public void enable()
```

Deprecated. *As of JDK version 1.1, replaced by* `setEnabled(boolean)`.

enable

```
@Deprecated
public void enable(boolean b)
```

Deprecated. *As of JDK version 1.1, replaced by* `setEnabled(boolean)`.

disable

```
@Deprecated
public void disable()
```

Deprecated. *As of JDK version 1.1, replaced by* `setEnabled(boolean)`.

awt.Component, deprecated since Java 1.1 still included in 7.0

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."

https://www.hyrumslaw.com/

https://xkcd.com/1172/

Why is API design important to you?

- If you program, you are an API designer
  - Good code is modular – each object/class/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

Design for ...

- 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

The Process of API Design

An API design process

- Define the scope of the API
  - Collect use-case stories, define requirements
  - Be skeptical: Distinguish true requirements from so-called solutions, "When in doubt, leave it out."
- Draft a specification, gather feedback, revise, and repeat
  - Keep it simple, short
- Code early, code often
  - Write *client code* before you implement the API

Plan with Use Cases

- Think about how the API might be used?
  - e.g., get the current time, compute the difference between two times, get the current time in Tokyo, get next week's date using a Maya calendar, …
- What tasks should it accomplish?
- Should all the tasks be supported?
  - If in doubt, leave it out!
- How would you solve the tasks with the API?

Respect the rule of three

- Via Will Tracz, *Confessions of a Used Program Salesman*:
  Write 3 implementations of each abstract class or interface 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 work fine."

The process of API design – 1-slide version

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, 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

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
  - You need **use-cases**; if you don't get them, keep trying
- 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

- Key question: what problems should this API solve?
  - ***Goals*** - Define scope of effort
- Also important: what problems shouldn't API solve?
  - ***Explicit non-goals*** - Bound 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
  - Provides defense against cranks
  - Saves rationale for posterity

Choosing an abstraction (model)

- **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. class
- For difficult APIs, can be the hardest part of the process

Start with short spec – one 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();
```

Write to the API, early and often

- Start before you've implemented the API
  - Saves you from 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**

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
  - Remember, they are obvious *only* in retrospect
  - Fix them and move on

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

- You'll likely find more problems as you flesh out the docs
  - Fix them
- Then you'll have an artifact you can share more widely
- Do so, but be sure 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!
  - Read it all…
  - But be selective: act only on the good feedback

Maintain realistic expectations

- **Most API designs are over-constrained**
  - You won't be able to please everyone…
  - So aim to displease everyone equally*
  - But maintain a unified, coherent, simple design!

- **Expect to make mistakes**
  - A few years of real-world use will flush them out
  - Expect to evolve API

* Well, not equally – I said that back in 2004 because I thought it sounded funny, and it stuck; actually you should decide which uses are most important and favor them.

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
  - now there are tools (github, Jira, Bugzilla, IntelliJ's TODO facility, etc.)

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
- Do not be a slave to this or any other process
  - It's good only to the extent that it results in a better API and makes your job easier

Information Hiding & Minimizing Conceptual Weight

Which one do you prefer?

```
public class Point {
        public double x;
        public double y;
}
// vs.
public class Point {
        private double x;
        private double y;
        public double getX() { /* … */ }
        public double getY() { /* … */ }
}
```

Key design principle: Information hiding

- "When in doubt, leave it out.”

- Implementation details in APIs are harmful
  - Confuse users
  - Inhibit freedom to change implementation

Information hiding also for APIs

- Make classes, members as private as possible
  - You can add features, but never remove or change the behavioral contract for an existing feature
- Public classes should have no public fields (with the exception of constants)
- Minimize *coupling*
  - Allows modules to be, understood, used, built, tested, debugged, and optimized independently

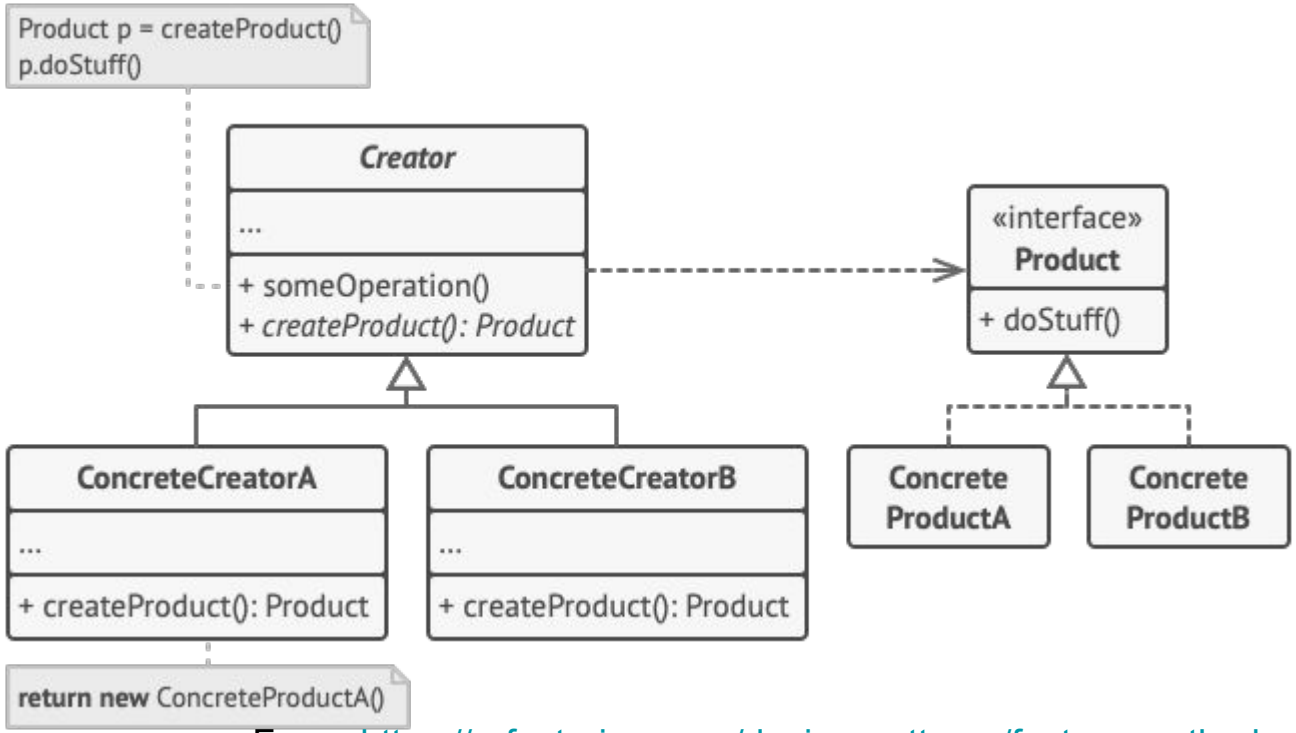
Which one do you prefer?

```
public class Rectangle {
   public Rectangle(Point e, Point f) …
}
// vs.
public class Rectangle {
   public Rectangle(PolarPoint e, PolarPoint f) …
}
```

Applying Information hiding: Factories

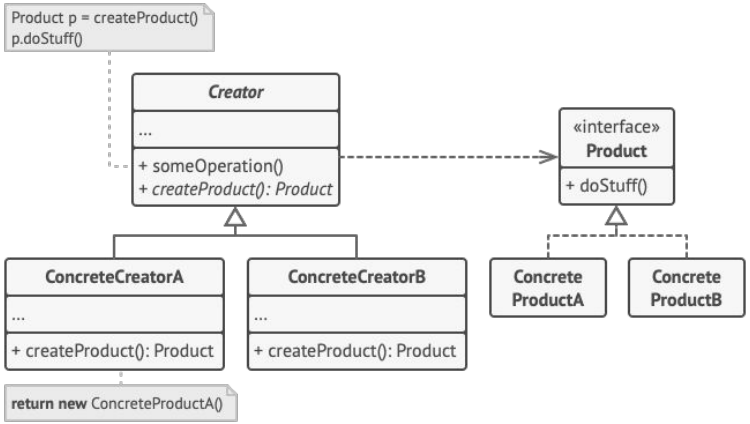
```
public class Rectangle {
        public Rectangle(Point e, Point f) …
}
// …
Point p1 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Point p2 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Rectangle r = new Rectangle(p1, p2);
```

Aside: The *Factory Method* Design Pattern



From: https://refactoring.guru/design-patterns/factory-method

Aside: The *Factory Method* Design Pattern



\+ Object creation separated from object
\+ Able to hide constructor from clients, control object creation
\+ Able to entirely hide implementation objects, only expose interfaces + factory
\+ Can swap out concrete class later
\+ Can add caching (e.g. `Integer.from()`)
\+ Descriptive method name possible

\- Extra complexity
\- Harder to learn API and write code

From: https://refactoring.guru/design-patterns/factory-method

Principle: Minimize conceptual weight

- API should be as small as possible but no smaller
  - **When in doubt, leave it out**
- Conceptual weight: How many concepts must a programmer learn to use your API?
  - APIs should have a "high power-to-weight ratio"

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

“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

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);
    ...
}
```

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

Boilerplate Code

```
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!
  }
}
```

- Generally done via cut-and-paste
- Ugly, annoying, and error-prone

Boilerplate Code

Generally created via cut-and-paste

Ugly, annoying, and error-prone

Sign of API not supporting common use cases directly

Consider creating APIs for most common use cases, hiding internals

Principle: Make it easy to do what's common, make it 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

Tradeoffs

How to balance

- Low conceptual weight
- Avoiding boilerplate code

?

Be Aware: Unintentionally Leaking Implementation Details

- Subtle leaks of implementation details through
  - Documentation: e.g., do not specify `hashCode()` return
  - Implementation-specific return types / exceptions: e.g., Phone number API that throws SQL exceptions
  - Output formats: e.g., `implements Serializable`
- Lack of documentation Implementation/StackOverflow becomes specification no hiding

But: Don’t overspecify method behavior

- 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

Be Aware: Unintentionally Leaki Implementation Details

- Subtle leaks of implementation details
  - Documentation: e.g., **do not specify has**
  - Implementation-specific return types / ex Phone number API that throws SQL exce
  - Output formats: e.g., `implements Seria`
- Lack of documentation Implementation becomes specification no hiding

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

Choosing names 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: e.g., **mail** suggests send, cc, bcc, inbox, outbox, folder, etc.

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
  - Avoid abbreviations
- Build *domain model* or glossary!

Discuss these names

- `get_x()` vs `getX()`
- `Timer` vs `timer`
- `isEnabled()` vs. `enabled()`
- `computeX()` vs. `generateX()`?
- `deleteX()` vs. `removeX()`?

Good names drive good design

- Be consistent
  - `computeX()` vs. `generateX()`?
  - `deleteX()` vs. `removeX()`?
- Avoid cryptic abbreviations
  - Good: `Font, Set, PrivateKey, Lock, ThreadFactory, TimeUnit, Future<T>`
  - Bad: `DynAnyFactoryOperations, _BindingIteratorImplBase, ENCODING_CDR_ENCAPS, OMGVMCID`

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!**

Another way names drive development

- Names may remind you of another API
- Consider **copying** its vocabulary and structure
- People who know other API will have an easy time learning yours
- You may be able to develop it more quickly
- You may be able to use types from the other API
- You may even be able to share implementation

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`

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

`addRow` `removeRow`

`addColumn` `removeColumn`

What's wrong here?

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

What's wrong here?

```
var timeoutID = setTimeout(function[, delay, arg1, arg2, ...]);
var timeoutID = setTimeout(function[, delay]);
var timeoutID = setTimeout(code[, delay]);

setTimeout(function () {
    // nice fast code here
},2000) // run after 2 seconds

setTimeout(`writeResults(${query.str})`, 100)
```

Don’t mislead your user

- Names have implications
- **Don’t violate *the principle of least astonishment***
- 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 outright

- **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…

Use consistent parameter ordering

- An egregious example from C:
  - `char* strncpy(char* dest, char* src,  size_t n);`
  - `void    bcopy(void* src,  void* dest, size_t n);`

Use consistent parameter ordering

- An egregious example from C:
  - `char* strncpy(char* dest, char* src, size_t n);`
  - `void bcopy(void* src, void* dest, size_t n);`
- Some good examples:
  - `java.util.Collections` – first parameter always collection to be modified or queried
  - `java.util.concurrent` – time always specified as long delay, TimeUnit unit

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
  - Write out realistic client code and compare
- Discuss names with colleagues; it really helps.

Other API Design Suggestions

Apply principles of user-centered design

e.g., "Principles of Universal Design"

- Equitable use: Design is useful and marketable to people with diverse abilities
- **Flexibility in use:** Design accommodates a wide range of individual preferences
- **Simple and intuitive use**: Use of the design is easy to understand
- Perceptible information: Design communicates necessary information effectively to user
- Tolerance for error
- Low physical effort
- Size and space for approach and use

Principle: Favor composition over inheritance

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);
    …
}
public class Properties {
    private final HashTable data = new HashTable();
    public String put(String key, String value) {
        data.put(key, value);
    }
    …
```

Principle: 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

Bad: `Date`, `Calendar`

Good: `LocalDate`, `Instant`, `TimerTask`

Antipattern: Long lists of parameters

- Especially with repeated parameters of the same type

```
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
  DWORD dwStyle, int x, int y, int nWidth, int nHeight,
  HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
  LPVOID lpParam);
```

- Long lists of identically typed params harmful
  - Programmers transpose parameters by mistake; programs still compile and run, but misbehave
- Three or fewer parameters is ideal
- Techniques for shortening parameter lists: Break up method, parameter objects, Builder Design Pattern

What's wrong here?

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);

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

Principle: Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);

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

Throw exceptions on exceptional conditions

- Don’t force client to use exceptions for control flow
- Conversely, don’t fail silently

```
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);
  }
}
```

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

// fails silently: “if the array is too
   short to hold all the threads, the
   extra threads are silently ignored”
```

Java: Avoid checked exceptions if possible

- Overuse of checked exceptions causes boilerplate

```
try {
   Foo f = (Foo) g.clone();
} catch (CloneNotSupportedException e) {
    // Do nothing. This exception can't happen.
}
```

Antipattern: returns require exception handling

- 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();
}
```

- Do not return a String if a better type exists

Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```
public class Throwable {
    public void printStackTrace(PrintStream s);
}
```

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E  comp
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
        at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.j
```

Don't let your output become your de facto API

- Document the fac
  the future
- Provide programm
  string form

```
public class Throwable
   public void printSt
}
```

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

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

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*

Contracts and Documentation

- APIs should be self-documenting
  - Good names drive good design
- Document religiously anyway
  - All public classes
  - All public methods
  - All public fields
  - All method parameters
  - Explicitly write behavioral specifications
- Documentation is integral to the design and development process

REST APIs

REST API

API of a web service

Uniform interface over HTTP requests

- Send parameters to URL, receive data (JSON, XML common)
- Stateless: Each request is self-contained

Language independent, distributed

REST API Design

All the same design principles apply

Document the API, input/output formats and error conditions!

CRUD Operations

Path correspond to nouns, not verbs, nesting common:

- `/articles`, `/state`, `/game`
  `/articles/:id/comments`

GET (receive), POST (submit new), PUT (update), and DELETE requests sent to those paths

Parameters for filtering, searching, sorting, e.g., `/articles?sort=date`

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json()); // JSON input
app.get('/articles', (req, res) => {
  const articles = [];
  // code to retrieve an article...
  res.json(articles);
});
app.post('/articles', (req, res) => {
  // code to add a new article...
  res.json(req.body);
});
app.put('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to update an article...
  res.json(req.body);
});
app.delete('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to delete an article...
  res.json({ deleted: id });
});
app.listen(3000, () => console.log('server started'));
```

REST Specifics

- JSON common for data exchange: Define and validate schema -- many libraries help
- Return HTTP standard errors (400, 401, 403, 500, …)
- Security mechanism through SSL/TLS and other common practices
- Caching common
- Consider versioning APIs `/v1/articles`, `/v2/articles`

Lecture summary

- APIs took off in the past thirty years, and gave us super-powers
- Good APIs are a blessing; bad ones, a curse
- API Design is hard
- Following an API design process greatly improves API quality
- Most good principles for good design apply to APIs
  - Don't adhere to them slavishly, but…
  - Don't violate them without good reason