

Principles of Software Construction

API Design

Claire Le Goues Vincent Hellendoorn
(Many slides originally from Josh Bloch)



Upcoming

Midterm 2 next Thursday

- Same as last time: in class period.
- All topics nominally in scope, but focus is on topics since Midterm 1.
- Sample questions have been released on piazza.
- 4-pages, front and back, allowed.

Final: scheduled for Thursday, Dec 15, 1 pm.

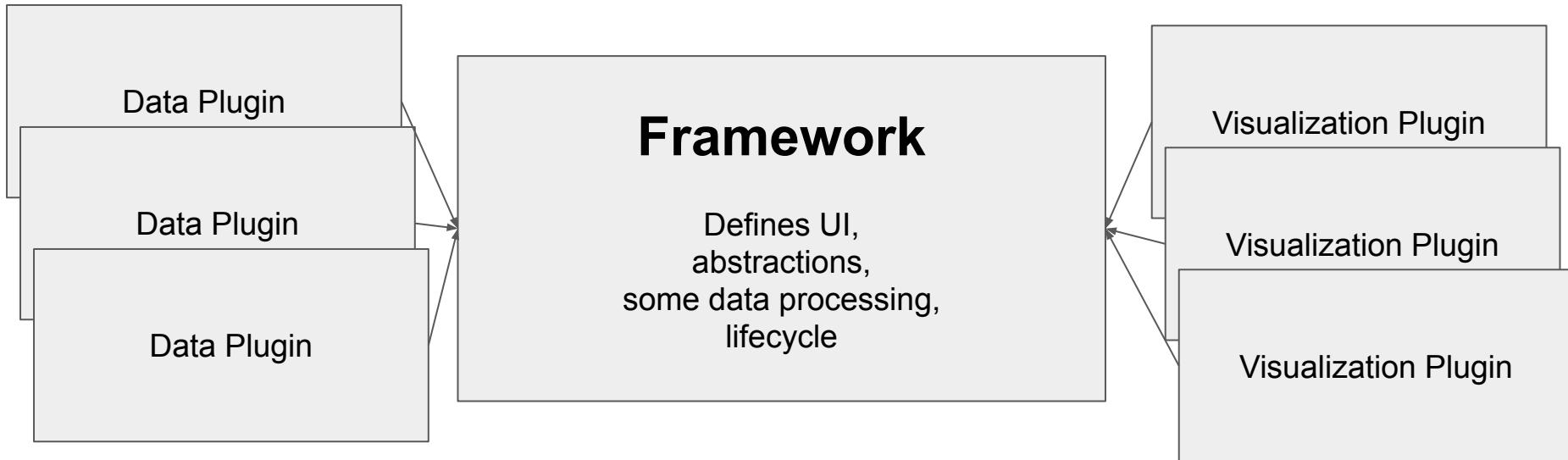
- Will be in person, proper 3-hour exam.
- You'll be able to bring notes.

Final homework (#6) will be released next week (possibly after midterm).

- Milestones: (1) Design framework, (2) implement framework, (3) implement plugins.
 - Note on the deadlines.
- Work in groups of 2–3. You can set your own groups, and there will be a pinned post on Piazza to help if you need it. Reach out if you're stuck.

Homework 6

Data Analytics Framework



HW6: Map-Based Data Visualizations?

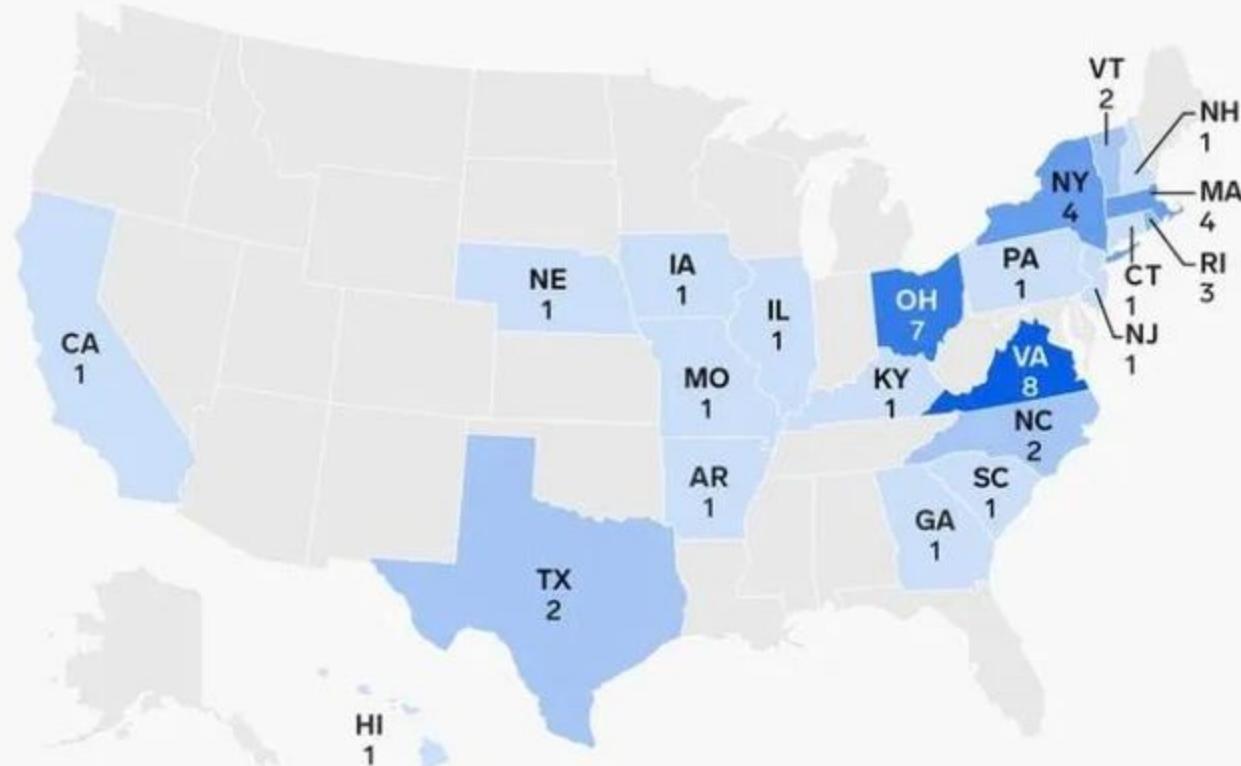
State, county, or country data

Data from many sources

Visualization as map image, table, google maps

Animations for time series data

States that produced the most presidents

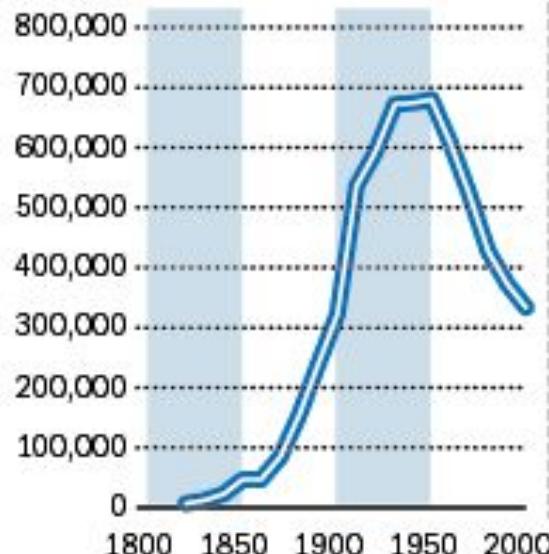


BUSINESS INSIDER

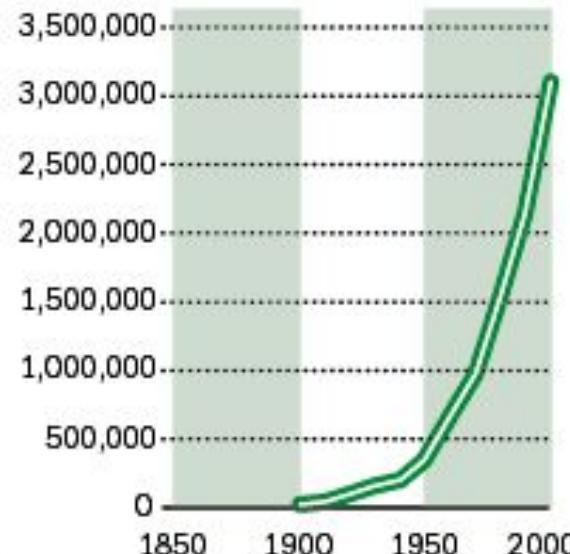
Population trends: Pittsburgh and Phoenix

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

PITTSBURGH



GREATER PHOENIX METRO AREA



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

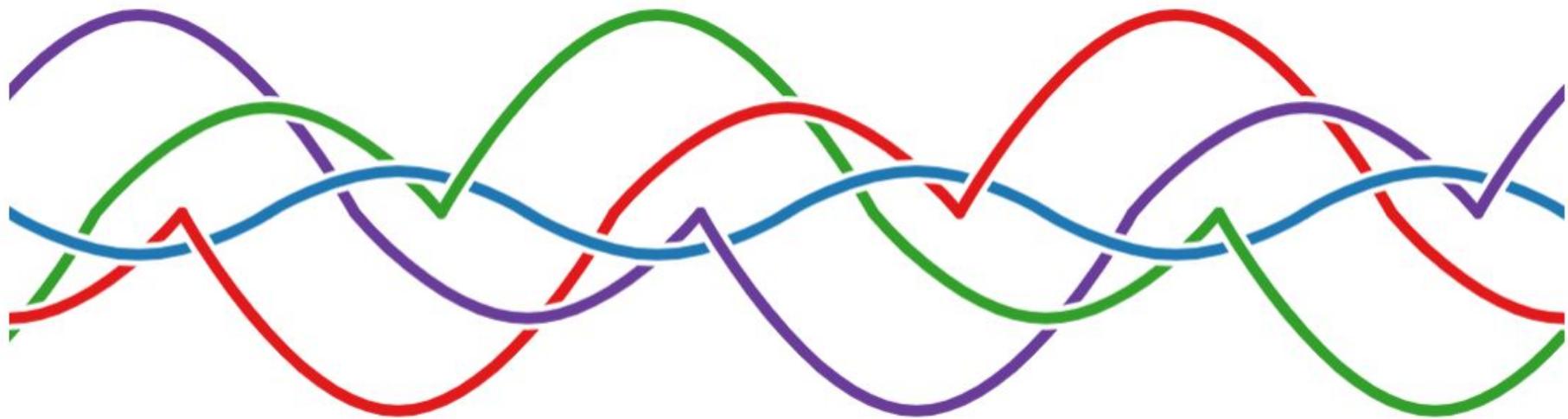
Oct

Nov

Dec

BestPlaces.Net

345



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



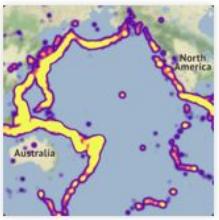
Funnel and Churn

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

Maps



Mapbox Map Layers



Mapbox Density Heatmap



Choropleth Mapbox

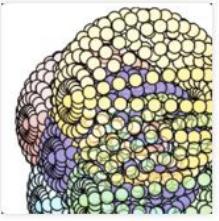


Lines on Maps

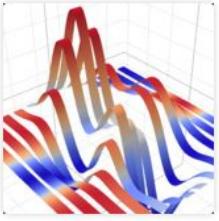


Bubble Maps

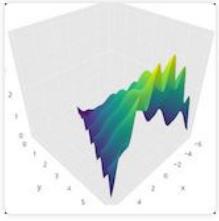
3D Charts



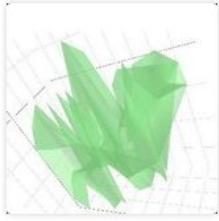
3D Scatter Plots



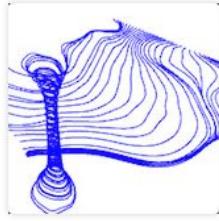
Ribbon Plots



3D Surface Plots

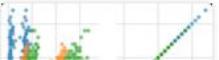


3D Mesh Plots



3D Line Plots

Subplots

[More Subplots »](#)

Where we are

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

An aside on annotations

```
public class SampleTest {  
    private List<String> emptyList;  
  
    @Before  
    public void setUp() {  
        emptyList = new ArrayList<String>();  
    }  
  
    @After  
    public void tearDown() {  
        emptyList = null;  
    }  
  
    @Test  
    public void testEmptyList() {  
        assertEquals("Empty list should have 0 elements",  
                    0, emptyList.size());  
    }  
}
```

Here the important plugin mechanism is Java annotations

API Design

Definitions, a design process

Design principles:

- Information Hiding
- Minimize conceptual weight
- Naming

Other design considerations (tying together other concepts from this semester)

REST APIs

Breaking changes in ecosystems

What's an API?

- Short for Application Programming Interface
 - = Contract for a Subsystem/Library, specification for a protocol
- 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 create applets.
java.awt	Contains all of the classes for creating graphical user interfaces.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data.
java.awt.dnd	Drag and Drop is a direct manipulation mechanism to transfer information between applications.
java.awt.event	Provides interfaces and classes for dealing with events.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining geometry.
java.awt.im	Provides classes and interfaces for the input method environment.
java.awt.im.spi	Provides interfaces that enable the input method environment.
java.awt.image	Provides classes for creating and manipulating images.
java.awt.image.renderable	Provides classes and interfaces for printing images.
java.awt.print	Provides classes and interfaces for printing documents.

Package java.util

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

See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the <i>collection hierarchy</i> .
Comparator<T>	A comparison function, which imposes a <i>total ordering</i> on its domain.
Deque<E>	A linear collection that supports element insertion and removal from both ends.
Enumeration<E>	An object that implements the Enumeration interface generally used by Iterator.
EventListener	A tagging interface that all event listener interfaces must implement.
Formattable	The Formattable interface must be implemented by a class that provides conversion methods for 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 traverse the list in either direction, and to change the underlying list while traversing it.
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 returning sub-maps.
NavigableSet<E>	A SortedSet extended with navigation methods reporting the minimum and maximum elements.
Observer	A class can implement the Observer interface when it wants to receive notifications.
Queue<E>	A collection designed for holding elements prior to processing them.
RandomAccess	Marker interface used by List implementations to indicate they support random access.
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A Map that further provides a <i>total ordering</i> 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

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

Packages

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

All Classes

- AbstractAction
- AbstractAnnotation
- AbstractAnnotatedElement
- AbstractBorder
- AbstractButton
- AbstractCellEditor
- AbstractCollection
- AbstractColorContext
- 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.awt.dnd

java.awt.event

java.awt.font

java.awt.geom

java.awt.im

java.awt.im.spi

java.awt.image

java.awt.image.renderable

java.awt.print

Edition 7

Platform, Standard Edition.

Description

Provides the classes necessary to create a Java 2D graphics context.

Contains all of the classes for creating Java 2D graphics contexts.

Provides classes for color spaces.

Provides interfaces and classes for drag and drop.

Drag and Drop is a direct manipulation mechanism to transfer information between applications.

Provides interfaces and classes for document editing.

Provides interfaces and classes for document visitors.

Provides the Java 2D classes for defining geometry.

Provides classes and interfaces for the Java Input Method Environment.

Provides interfaces that enable the Java Input Method Environment.

Provides classes for creating and manipulating images.

Provides classes and interfaces for printing.

Provides classes and interfaces for printing.

Package `java.util`

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

See: Description

Interface Summary	Description
<code>Collection<E></code>	The root interface in the <i>collection hierarchy</i> .
<code>Comparator<T></code>	A comparison function, which imposes a <i>total ordering</i> on elements.
<code>Deque<E></code>	A linear collection that supports element insertion and removal from both ends.
<code>Enumeration<E></code>	An object that implements the <code>Enumeration</code> interface generally.
<code>EventListener</code>	A tagging interface that all event listener interfaces must implement.
<code>Formattable</code>	The <code>Formattable</code> interface must be implemented by a class whose conversion specifier is a <code>Formatter</code> .
<code>Iterator<E></code>	An iterator over a collection.
<code>List<E></code>	An ordered collection (also known as a <i>sequence</i>).
<code>ListIterator<E></code>	An iterator for lists that allows the programmer to traverse the list in either direction, and to change its current position in the list.
<code>Map<K,V></code>	An object that maps keys to values.
<code>Map.Entry<K,V></code>	A map entry (key-value pair).
<code>NavigableMap<K,V></code>	A <code>SortedMap</code> extended with navigation methods returning sub-maps based on a specified view of the map's keys.
<code>NavigableSet<E></code>	A <code>SortedSet</code> extended with navigation methods reporting sub-sorts based on a specified view of the set's elements.
<code>Observer</code>	A class can implement the <code>Observer</code> interface when it wants to receive notifications.
<code>Queue<E></code>	A collection designed for holding elements prior to processing them.
<code>RandomAccess</code>	Marker interface used by <code>List</code> implementations to indicate that they support random access.
<code>Set<E></code>	A collection that contains no duplicate elements.
<code>SortedMap<K,V></code>	A <code>Map</code> that further provides a <i>total ordering</i> on its keys.

API: Application Programming Interface

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

The diagram illustrates the concept of an API boundary by comparing two different interfaces: the `java.util.Collection<E>` interface and the GitHub REST API's repository endpoints.

Java.util.Collection<E> interface: This section shows the JavaDoc for the `Collection` interface. It includes sections for **Packages** (containing `java.awt`, `java.awt.event`, etc.) and **All Classes** (listing various concrete implementations like `AbstractList`, `AbstractMap`, etc.). The interface itself defines methods such as `add(E e)`, `addAll(Collection<E> c)`, `remove(E e)`, `removeAll(Collection<E> c)`, `retainAll(Collection<E> c)`, `contains(E e)`, `containsAll(Collection<E> c)`, `clear()`, `size()`, `isEmpty()`, `iterator()`, `toArray()`, and `toArray(E[] a)`.

Github REST API: This section shows the GitHub API documentation for listing repositories. It includes a screenshot of a browser showing the URL `https://developer.github.com/v3/repos/` and the resulting JSON response. The response lists repositories for the authenticated user, with parameters for `type`, `sort`, and `direction`. Below this, there are two more endpoints: `/user/repos` and `/users/:username/repos`, each with its own set of parameters.

Boundary: A vertical orange line separates the Java API from the GitHub API, representing the boundary where the two systems interact. The Java API is on the left, and the GitHub API is on the right.

API: Application Programming Interface

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

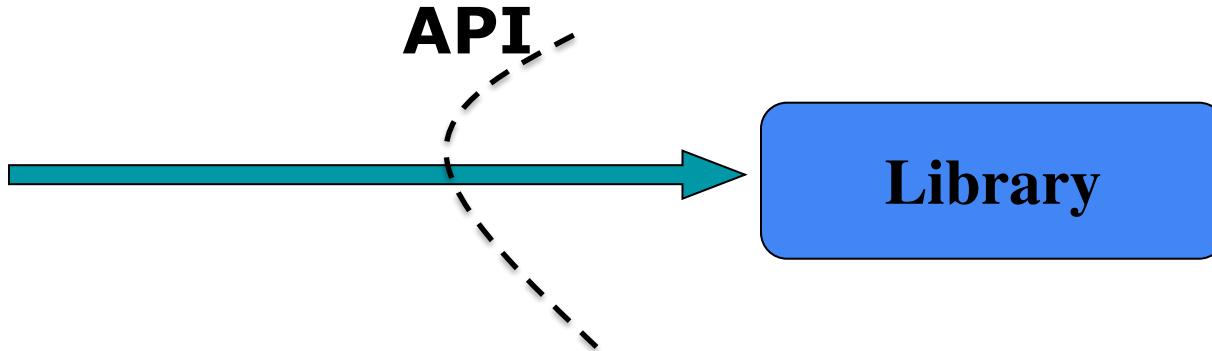
The image is a collage of several different screens and code snippets, all highlighted with orange boxes, illustrating the concept of an API boundary:

- Top Left:** A stack trace from a Java application showing multiple layers of exception handling and component interaction.
- Middle Left:** A portion of a Java code interface, likely a service definition, showing methods like `size()`, `isEmpty()`, `iterator()`, and `toArray()`.
- Bottom Left:** A list of Java class names, including `AbstractAction`, `AbstractAnnotation`, `AbstractBorder`, `AbstractButton`, `AbstractCellEditor`, `AbstractCollection`, `AbstractColor`, `AbstractDocument`, `AbstractEventContext`, `AbstractDocument.Content`, `AbstractDocument.ElementEdit`, `AbstractElementVisitor`, `AbstractElementVisitor`, `AbstractExecutorService`, `AbstractInterruptibleChannel`, `AbstractLayoutCache`, `AbstractLayoutCache.NodeDimensions`, `AbstractList`, `AbstractListModel`, `AbstractMap`, `AbstractMap.SimpleEntry`, `AbstractMap.SimpleImmutableEntry`, `AbstractMarshallerImpl`, `AbstractMethodError`, and `AbstractOwnableSynchronizer`.
- Middle Center:** A screenshot of a Java IDE showing a list of classes under `java.awt.dnd`, including `java.awt.dnd`, `java.awt.event`, `java.awt.font`, `java.awt.geom`, `java.awt.im`, `java.awt.im.spi`, `java.awt.image`, `java.awt.image.renderable`, and `java.awt.print`.
- Bottom Center:** A screenshot of a Java IDE showing a list of interfaces under `java.awt`, including `Direction`, `String`, `Can be one of`, `otherwise desc`, `List user repositories`, `List public repositories for the specified user.`, `GET /users/:username/repos`, `Parameters`, `Name`, `Type`, `Can be one of all`, `Can be one of created`, `Provides interfaces that enable the de`, `Provides classes for creating and mo`, `Provides classes and interfaces for pr`, `Provides classes and interfaces for a`, `Observer`, `Queue<E>`, `RandomAccess`, `Set<E>`, and `SortedMap<K,V>`.
- Right Side:** A snippet of XML code defining an Eclipse plugin extension, specifically for an XML editor. It includes elements like `<plugin>`, `<extension point="org.eclipse.ui.editors">`, `<editor name="Sample XML Editor"`, `extensions="xml"`, `icon="icons/sample.gif"`, `contributorClass="org.eclipse.ui.text.editor.BasicTextEditorActionContribut`, and `or"`.
- Bottom Right:** A snippet of Java code defining a class named `myeditor.editors.XMLEditor` with an ID of `myeditor.editors.XMLEditor`.

Libraries and frameworks (and protocols!) define APIs

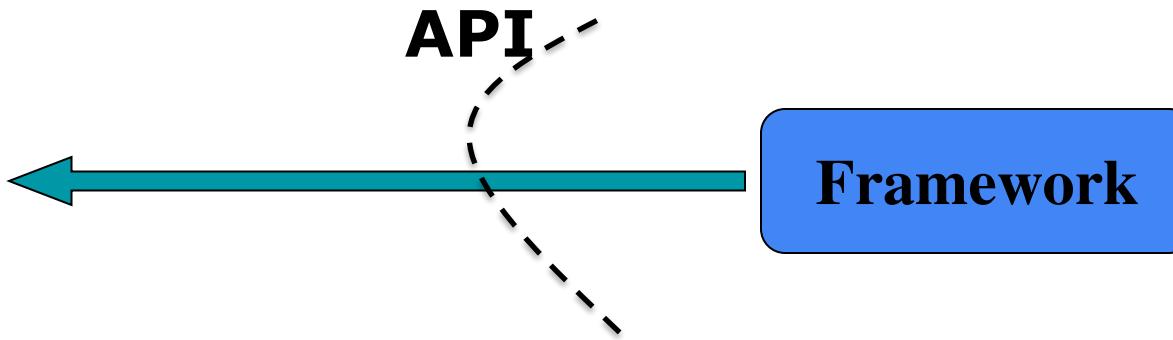
```
public MyWidget extends JPanel {  
    public MyWidget(int param) { /* setup  
        internals, without rendering  
    */  
  
        /* render component on first view and  
        resizing  
        protected void  
        paintComponent(Graphics g) {  
            // draw a red box on his  
            componentDimension d = getSize();  
            g.setColor(Color.red);  
            g.drawRect(0, 0, d.getWidth(),  
            d.getHeight());  
        }  
    }  
}
```

your code



```
public MyWidget extends JPanel {  
    public 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());  
        }  
    }  
}
```

your code



API design is important

A good API is a joy to use

- Users invest heavily: learning, using
- Cost to stop using an API can be prohibitive, so successful public APIs capture users

APIs can also be among your greatest liabilities

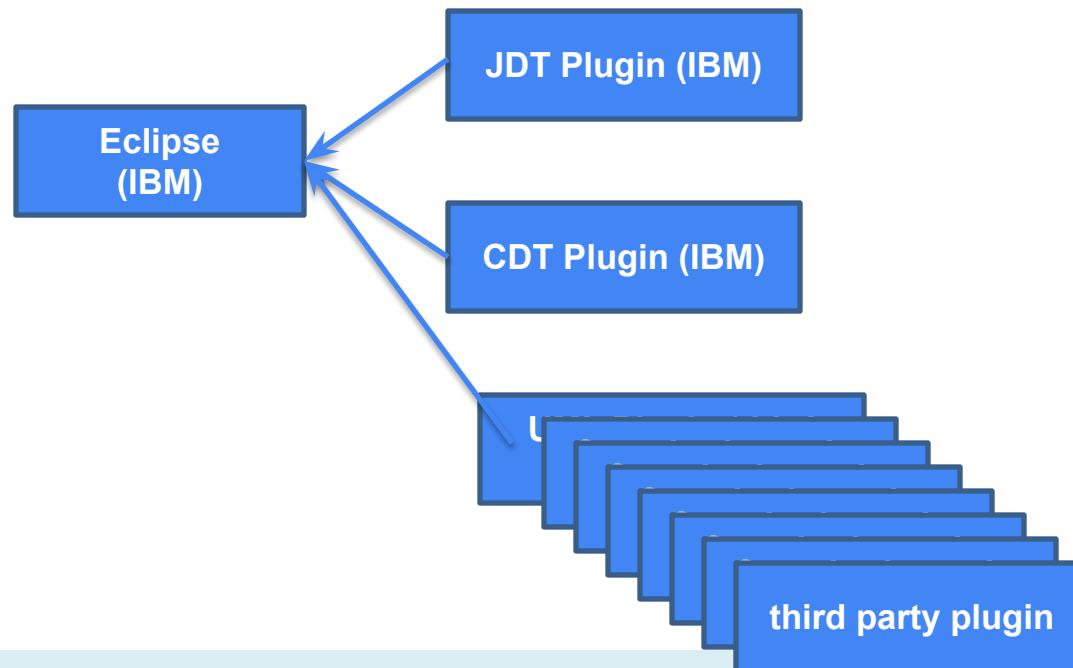
- Bad API can cause unending stream of support requests, inhibit forward movement
- Public APIs are forever – one chance to get it right

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 in general improves code quality.

Public APIs are forever



Public 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 is a 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

Discuss: What makes a good API?

Positive, negative experiences?

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

An API design process: plan with use cases

- Similar to our framework discussion!
- 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."
 - Be explicit about *non-goals*
- Draft a specification, gather feedback, revise, and repeat. Keep it simple, short!
- Code early, code often: Write *client code* before you implement the API

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, to avoid doing implementation you'll throw away.
- Start before you've even specified it properly, to avoid 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!
- Respect the rule of 3, 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."

Information hiding

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/>

CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS
WHEN YOU HOLD DOWN SPACEBAR.

COMMENTS:

LONGTIMEUSER4 WRITES:

THIS UPDATE BROKE MY WORKFLOW!
MY CONTROL KEY IS HARD TO REACH,
SO I HOLD SPACEBAR INSTEAD, AND I
CONFIGURED EMACS TO INTERPRET A
RAPID TEMPERATURE RISE AS "CONTROL".

ADMIN WRITES:

THAT'S HORRIFYING.

LONGTIMEUSER4 WRITES:

LOOK, MY SETUP WORKS FOR ME.
JUST ADD AN OPTION TO REENABLE
SPACEBAR HEATING.

EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

<https://xkcd.com/1172/>

Information hiding is also important for APIs

- Implementation details in APIs are harmful: Confuses users and inhibits freedom to change implementation
- Make classes, members as private as possible
- Public classes should have no public fields, except for constants
- Minimize coupling, so modules can be, understood, used, built, tested, debugged, and optimized independently

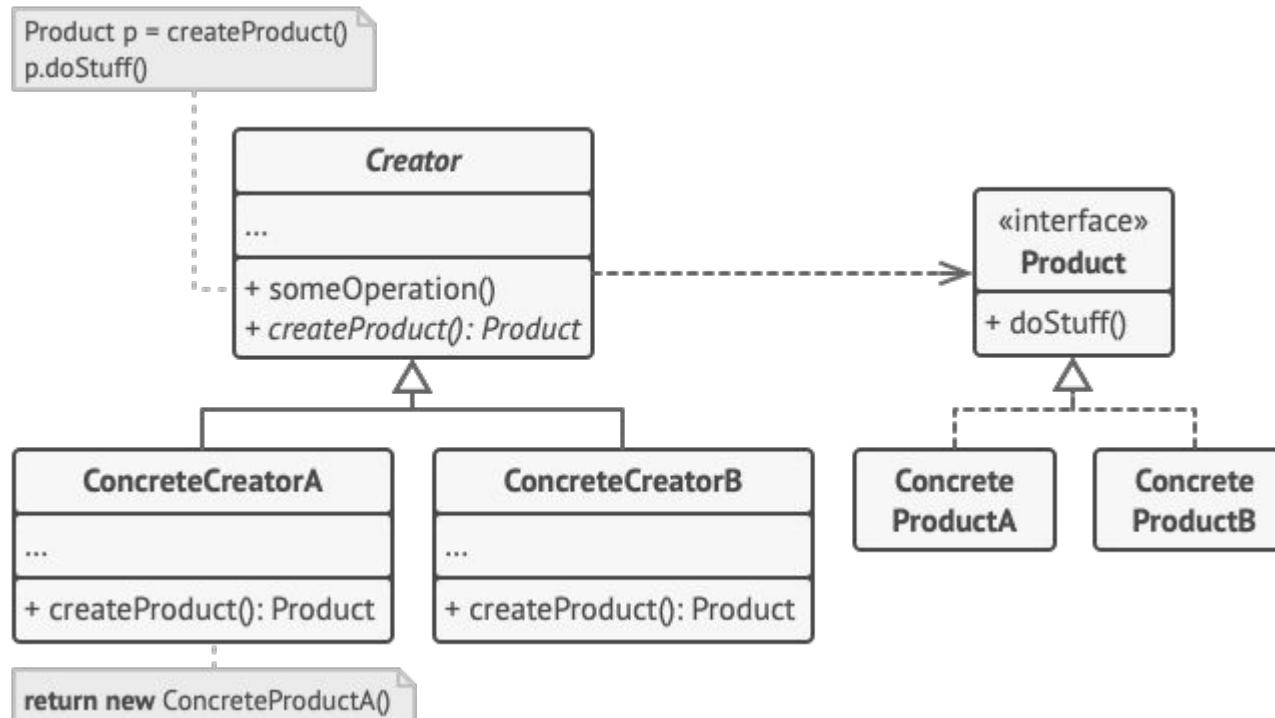
Be Aware: Unintentionally Leaking Implementation Details

- 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/Stack Overflow becomes specification → no hiding

Applying Information hiding: Factories

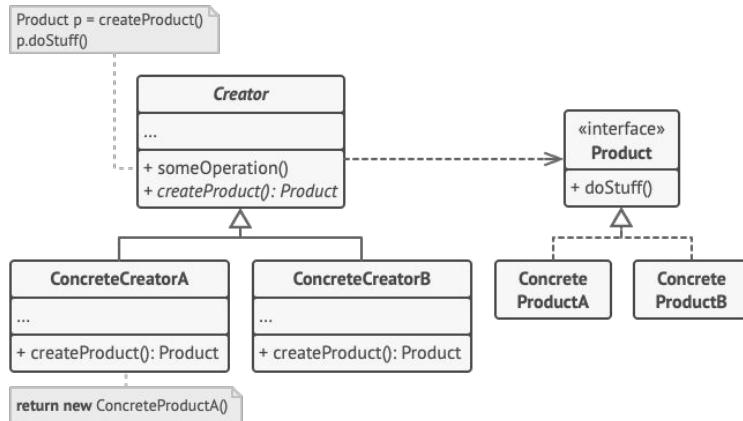
```
public class Rectangle {  
    public Rectangle(Point e, Point f) ...  
}  
// ...  
Point p1 = PointFactory.Construct(...);  
// new PolarPoint(...); inside  
Point p2 = PointFactory.Construct(...);  
// new PolarPoint(...); inside  
Rectangle r = new Rectangle(p1, p2);
```

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>

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.ja  
at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)  
at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)  
at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)  
at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)  
at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

Minimizing Conceptual Weight

Conceptual weight: How many concepts must a programmer learn to use your API?

- **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
- Goal: a high *power-to-weight ratio*: an API that lets you do a lot with a little

Example: generalizing an API can make it smaller

Subrange operations on Vector – legacy List implementation

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

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

Example: generalizing an API can make it smaller

Subrange operations on List

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

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

Tradeoff: 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.DOCUMENT_TYPE_DECLARATION, 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
- Sign of API not supporting common use cases

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

Naming

Names Matter – API is a little language

Naming is perhaps the single most important factor in API usability

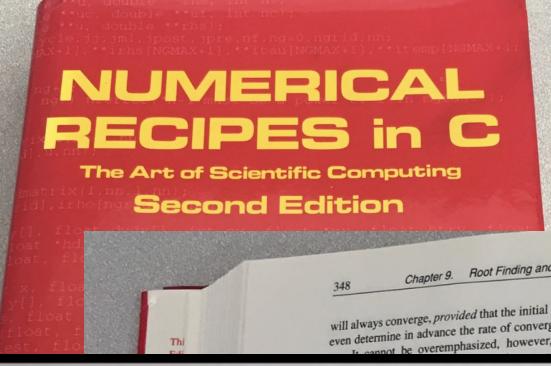
- Primary goals
 - Client code should read like prose (“easy to read”)
 - Client code should mean what it says (“hard to misread”)
 - Client code should flow naturally (“easy to write”)
- To that end, names should:
 - be largely self-explanatory
 - leverage existing knowledge
 - interact harmoniously with language and each other
- **Don’t violate *the principle of least astonishment***

Discuss these names

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

Good names drive good design, make code easier to read and write.

- Be consistent: Never use the same word for multiple meanings, or multiple words for the same meaning.
 - `computeX()` vs. `generateX()`; `deleteX()` vs. `removeX()`?
- Avoid cryptic abbreviations
 - Good: `Set`, `PrivateKey`, `Lock`, `ThreadFactory`, `Future<T>`
 - Bad: `DynAnyFactoryOperations`, `ENCODING_CDR_ENCAPS`, `OMGVMCID`
- Good names related to good abstractions.
- Literal names often have associations (e.g.,: mail, matrix), or analogies, make sure they make sense!



```

int jz,j,i;
float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
char scr[ISCR+1][JSCR+1];
    
```

sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-liners: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 10 to determine whether a tantalizing dip in the function really does cross zero or not. (You can easily modify the simple golden section routine of §10.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a *double* root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine *rtsafe* in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned.
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), and it

348

Chapter 9. Root Finding and Nonlinear Sets of Equations

will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

To cannot be overemphasized, however, how crucially success depends on

9.0 Introduction

349

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Various implementations for specific computers are given in the appendixes.

```

for (j=2;j<=(JSCR-1);j++)
    scr[i][j]=BLANK;
}
dx=(x2-x1)/(ISCR-1);
x=x1;
ysml=ybig=0.0;
for (i=1;i<ISCR;i++) {
    y[i]=(*fr)(x);
    if (y[i] < ysml) ysml=y[i];
    if (y[i] > ybig) ybig=y[i];
    x += dx;
}
if (ybig == ysml) ybig=ysml+1.0;
dyj=(JSCR-1)/(ybig-ysml);
jz=1-(int)(ysml*dyj);
for (i=1;i<ISCR;i++) {
    scr[i][jz]=ZERO;
    jz+=1-(int)((y[i]-ysml)*dyj);
    scr[i][jz]=FF;
}
printf(" %10.3f ",ybig);
for (i=1;i<ISCR;i++) printf("%c",scr[i][JSCR]);
printf("\n");
for (j=(JSCR-1);j>=2;j--) {
    printf(" %12e ", );
    for (i=1;i<ISCR;i++) printf("%c",scr[i][j]);
    printf("\n");
}
    
```

Fill interior with blanks.

Limits will include 0.
Evaluate the function at equal intervals.
Find the largest and smallest values.

Be sure to separate top and bottom.
Note which row corresponds to 0.
Place an indicator at function height and 0.

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`
- Aim for regularity: If API has 2 verbs and 2 nouns, programmers will expect all 4 combinations

`addRow`

`removeRow`

`addColumn`

`removeColumn`

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`

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();  
}
```

FIXME: What's wrong here?

```
var timeoutID = setTimeout(function[, delay, arg1, arg2, ...]);  
var timeoutID = setTimeout(function[, delay]);  
var timeoutID = setTimeout(code[, delay]);  
  
setTimeout(function () {  
    // something to execute in 2 seconds  
, 2000)  
  
query.str = “); fs.rm(‘/’, ‘-rf’’  
setTimeout(`writeResults(${query.str})`, 100)
```

Good naming takes time, but it's worth it

- Don't be afraid to spend hours on it; API designers do.
 - And 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

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 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

Principle: Fail fast, early, and not silently.

```
// 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);
}
```

...What's wrong here?

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

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 (or RESTful – representational state transfer) API

API of a web service “that conforms to the constraints of the REST architectural style.”

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`

Breaking Changes

Backward Compatible Changes

Can add new interfaces, classes

Can add methods to APIs,
but cannot change interface implemented by clients

Can loosen precondition and tighten postcondition,
but no other contract changes

Cannot remove classes, interfaces, methods

Clients may rely on undocumented behavior and
even bugs



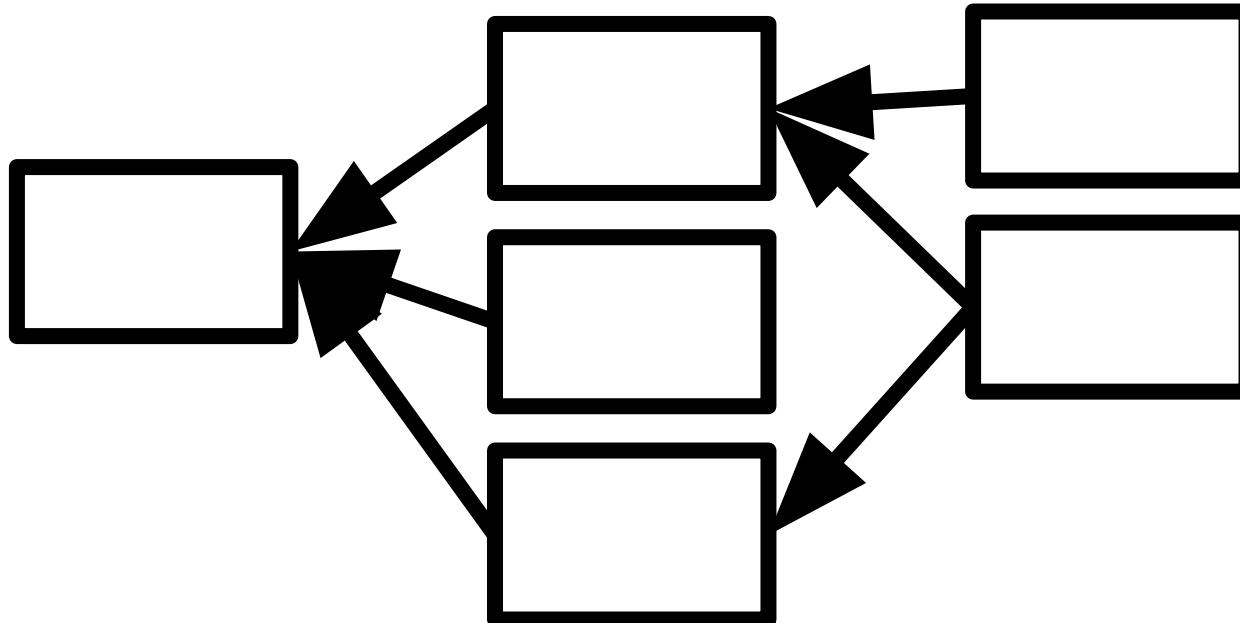
Breaking Changes

Not backward compatible (e.g., renaming/removing method)

Clients may need to change their implementation when they update

or even migrate to other library

May cause costs for rework and interruption, may ripple through ecosystem



Software Ecosystem

Breaking changes can be hard to avoid

Need better planning? (Parnas' argument)

Requirements and context change

Bugs and security vulnerabilities

Inefficiencies

Rippling effects from upstream changes

Technical debt, style

Semantic Versioning

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

Code status	Stage	Rule	Example version
First release	New product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

Cost distributions and practices are community dependent

How to Break an API?

In Eclipse, you don't.

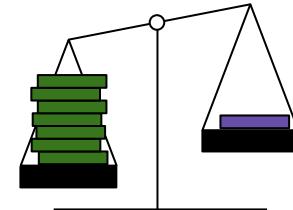
In CRAN, you reach out to affected downstream developers.

In Node.js, you increase the major version number.

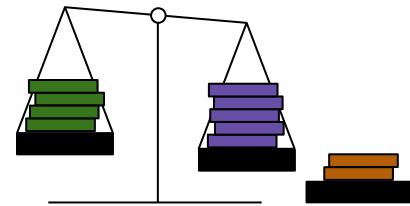
Contrast



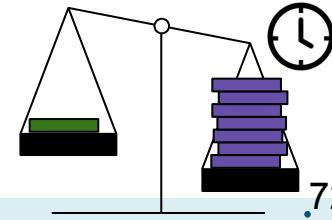
Backward compatibility
for clients



Timely access to current
research for end users



Easy and fast to publish/use
for developers







Backward compatibility to
reduce costs for **clients**

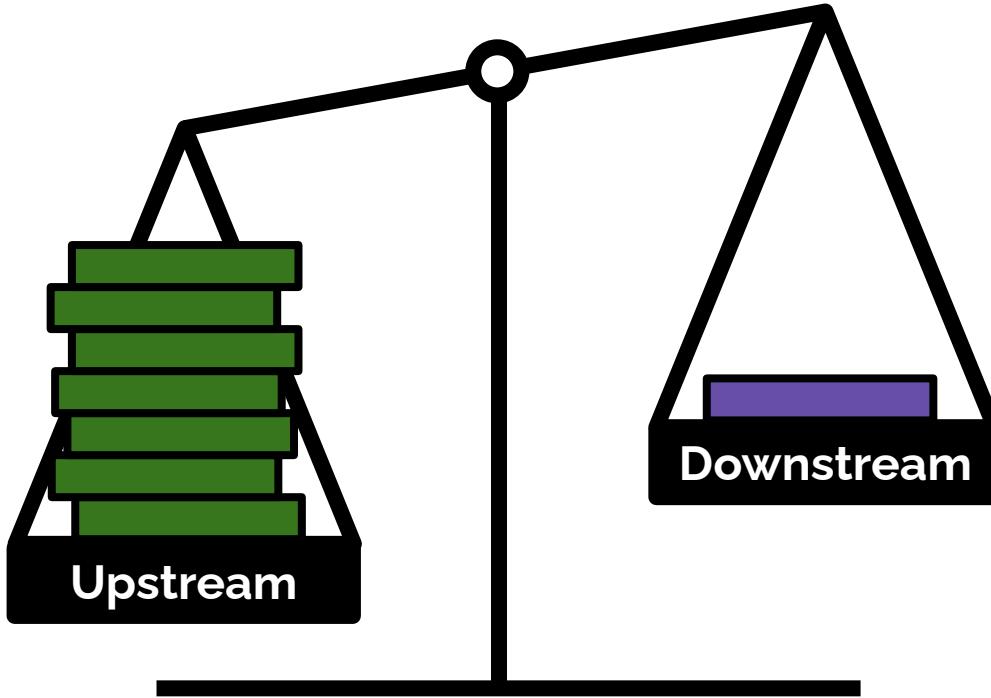
*“API Prime Directive: When
evolving the Component API
from release to release, do
not break existing Clients”*

https://wiki.eclipse.org/Evolving_Java-based_APIs

Values



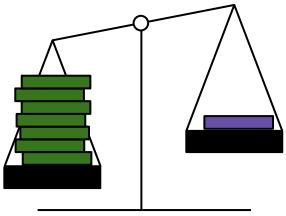
Backward
compatibility
for clients



Yearly synchronized
coordinated releases



Backward compatibility
for clients

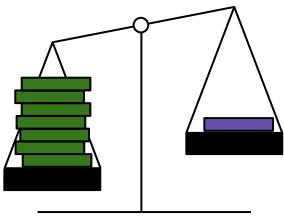


- Willing to accept high costs + opportunity costs
- Educational material, workarounds
- API tools for checking
- Coordinated release planning
- No parallel releases

Upstream



Backward compatibility
for clients

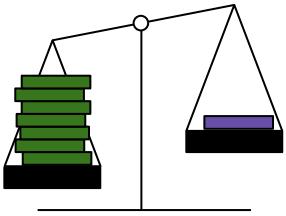


Convenient to use as resource
Yearly updates sufficient for many
Stability for corporate users

Downstream



Backward compatibility
for clients



Perceived stagnant development
and political decision making

Stale platform; discouraging
contributors

Coordinated releases as pain points

SemVer prescribed but not followed

Friction

“Typically, if you have hip things, then you get also people who create new APIs on top ... to create the next graphical editing framework or to build more efficient text editors. ... And these things don’t happen on the Eclipse platform anymore.”





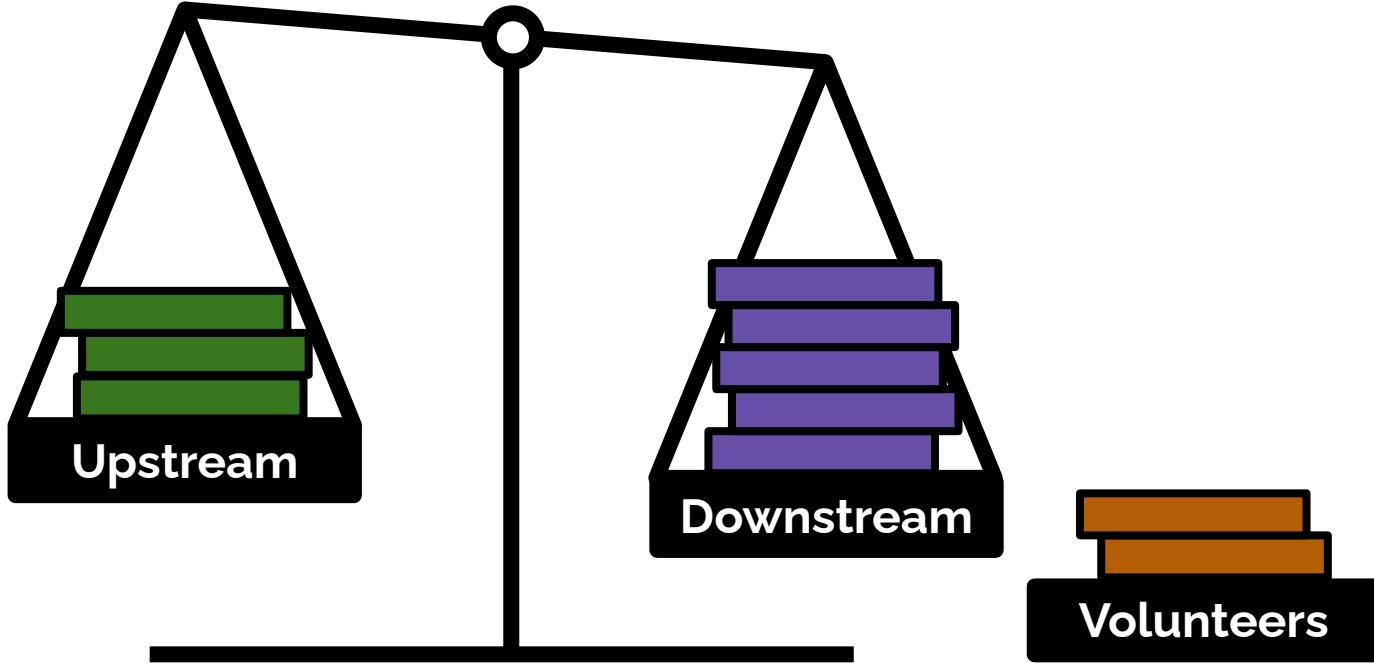
Ease for **end users** to install
and update packages

“CRAN primarily has the academic users in mind, who want timely access to current research” [R10]

Values



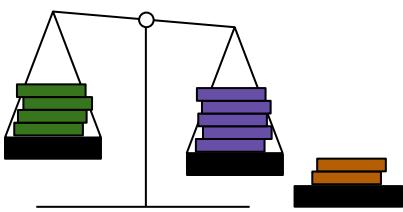
Timely access to
current research
for end users



Continuous synchronization,
~1 month lag



Timely access to
current research
for end users



Snapshot consistency within the ecosystem (not outside)

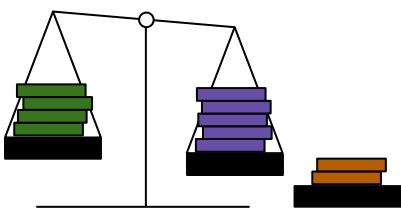
Reach out to affected downstream developers: resolve before release

Gatekeeping: reviews and automated checking against downstream tests

Upstream



Timely access to
current research
for end users



Waiting for emails, reactive monitoring

Urgency when upstream package
updates

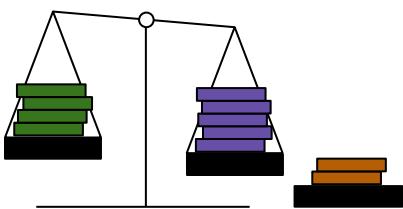
Dependency = collaboration

Aggressive reduction of dependencies,
code cloning

Downstream



Timely access to
current research
for end users



Urgency and reacting to updates as burden vs. welcoming collaboration

Gatekeeping works because of prestige of being in repository

Updates can threaten scientific reproducibility

Friction

“And then I need to [react to] some change ... and it might be a relatively short timeline of two weeks or a month. And that's difficult for me to deal with, because I try to sort of focus one project for a couple weeks at a time so I can remain productive.”





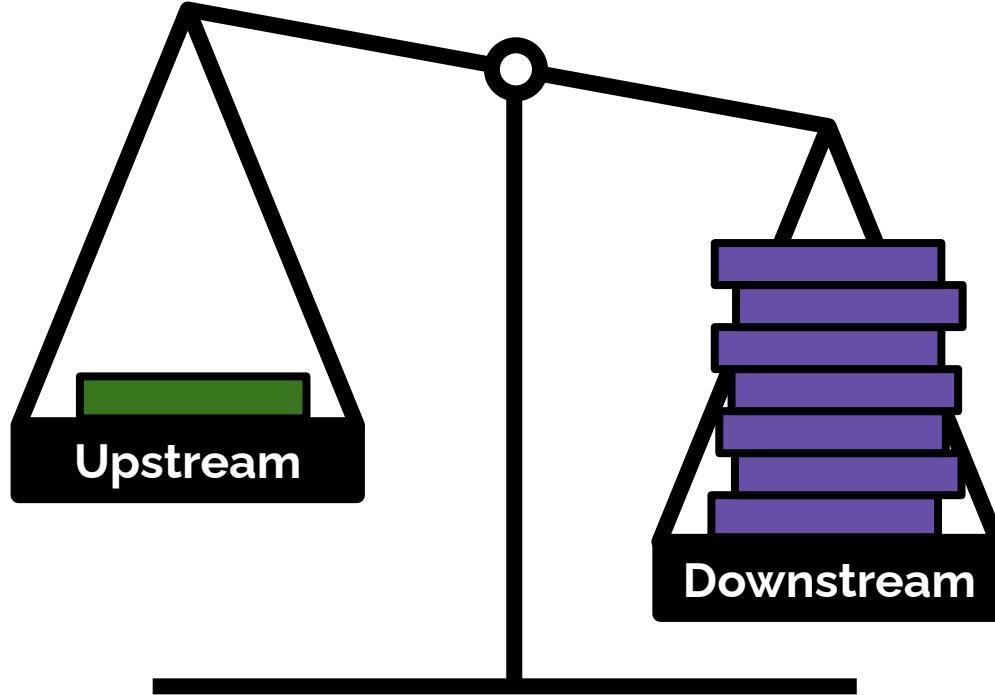
Easy and fast for **developers** to publish and use packages

Open to rapid change,
no gate keeping,
experimenting with APIs until
they are right

Values



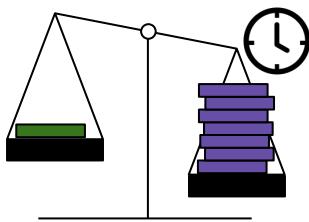
Easy and fast to
publish and use
for developers



Decoupled pace, update
at user's discretion



Easy and fast to
publish and use
for developers

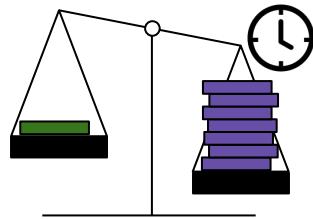


Breaking changes easy
More common to remove technical
debt, fix APIs
Signaling intention with SemVer
No central release planning
Parallel releases more common

Upstream



Easy and fast to publish and use for developers



Technology supports using old + mixed revisions; decouples upstream and downstream pace

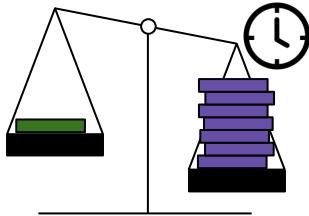
Choice to stay up to date

Monitoring with social mechanisms and tools (e.g., greenkeeper)

Downstream



Easy and fast to
publish and use
for developers



Rapid change requires constant
maintenance

Emphasis on tools and community,
often grassroots

Friction

**“Last week’s tutorial is
out of date today.”**

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 unconditionally, but...
 - Don't violate them without good reason