Principles of Software Construction

API Design

Claire Le Goues Vincent Hellendoorn
(Many slides originally from Josh Bloch, some from Christian Kaestner)

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

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.

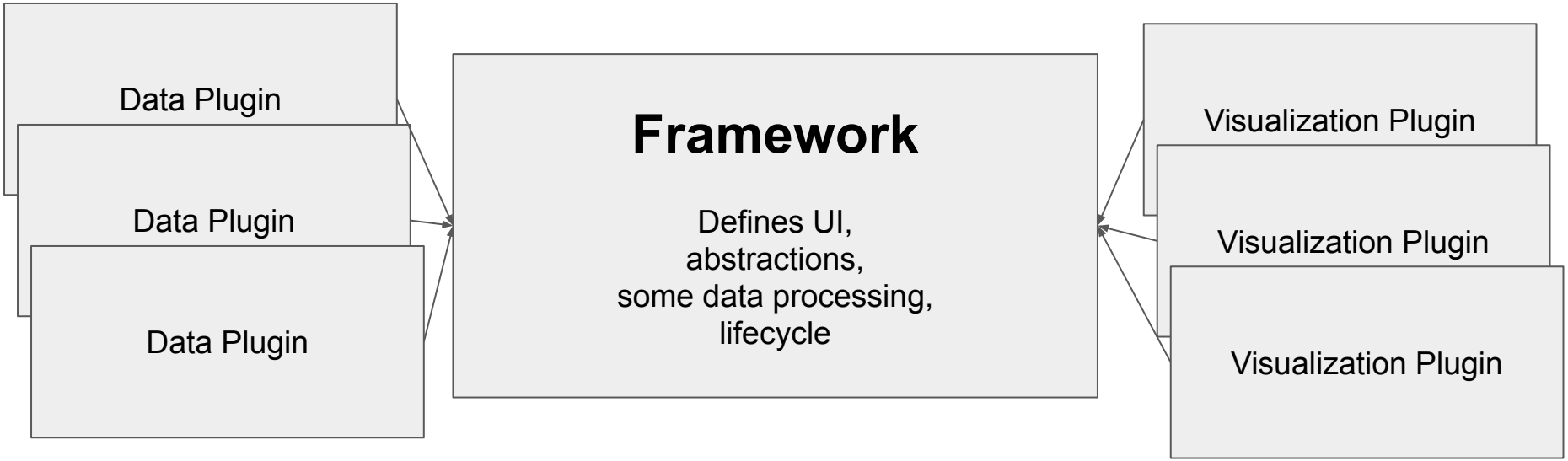
- Will be in person, proper 3-hour exam.
- You'll be able to bring notes.

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



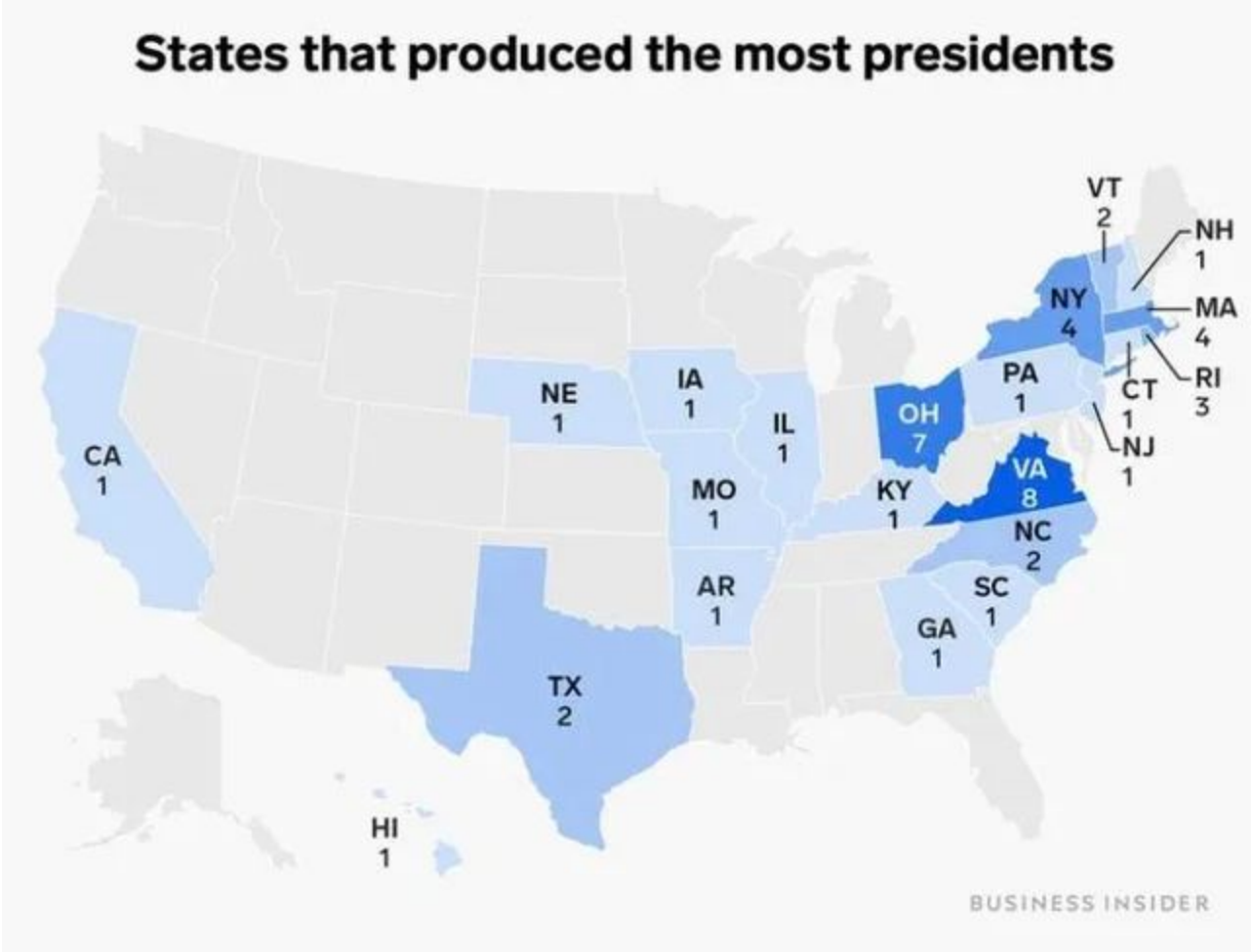
HW6: Map-Based Data Visualizations?

State, county, or country data

Data from many sources

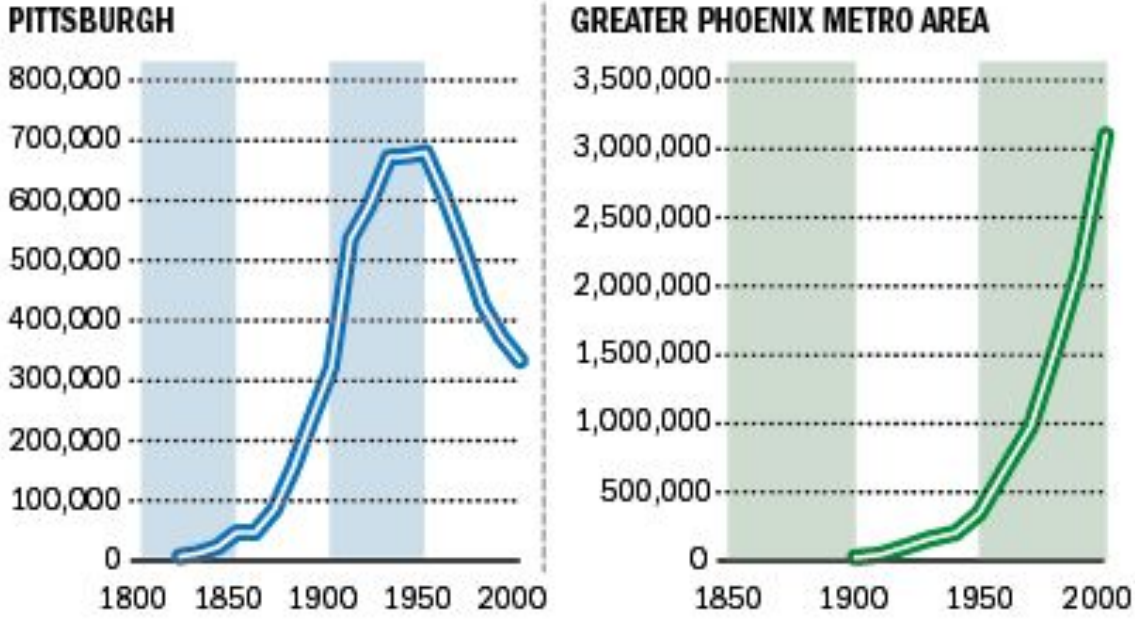
Visualization as map image, table, google maps

Animations for time series data

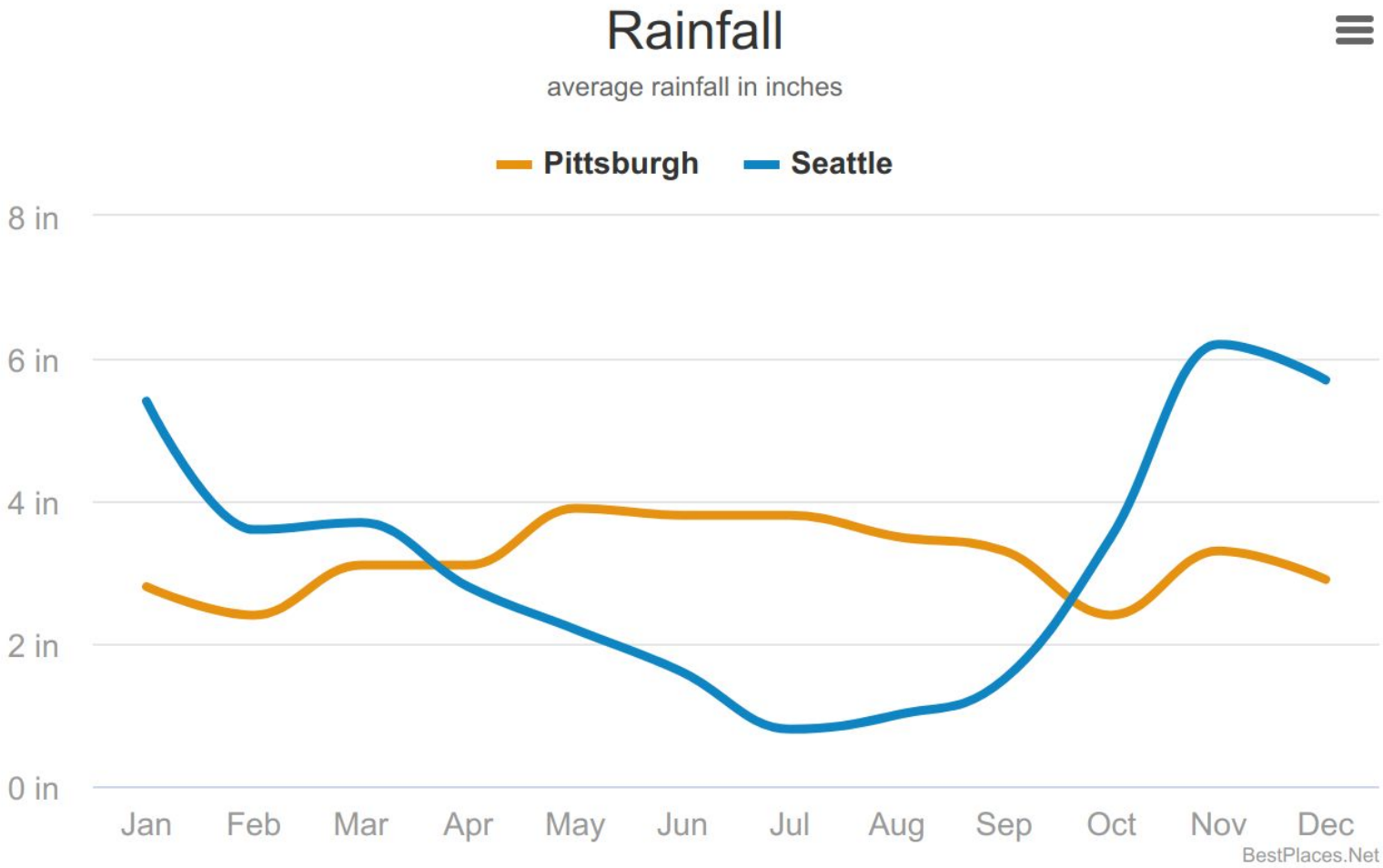


Population trends: Pittsburgh and Phoenix

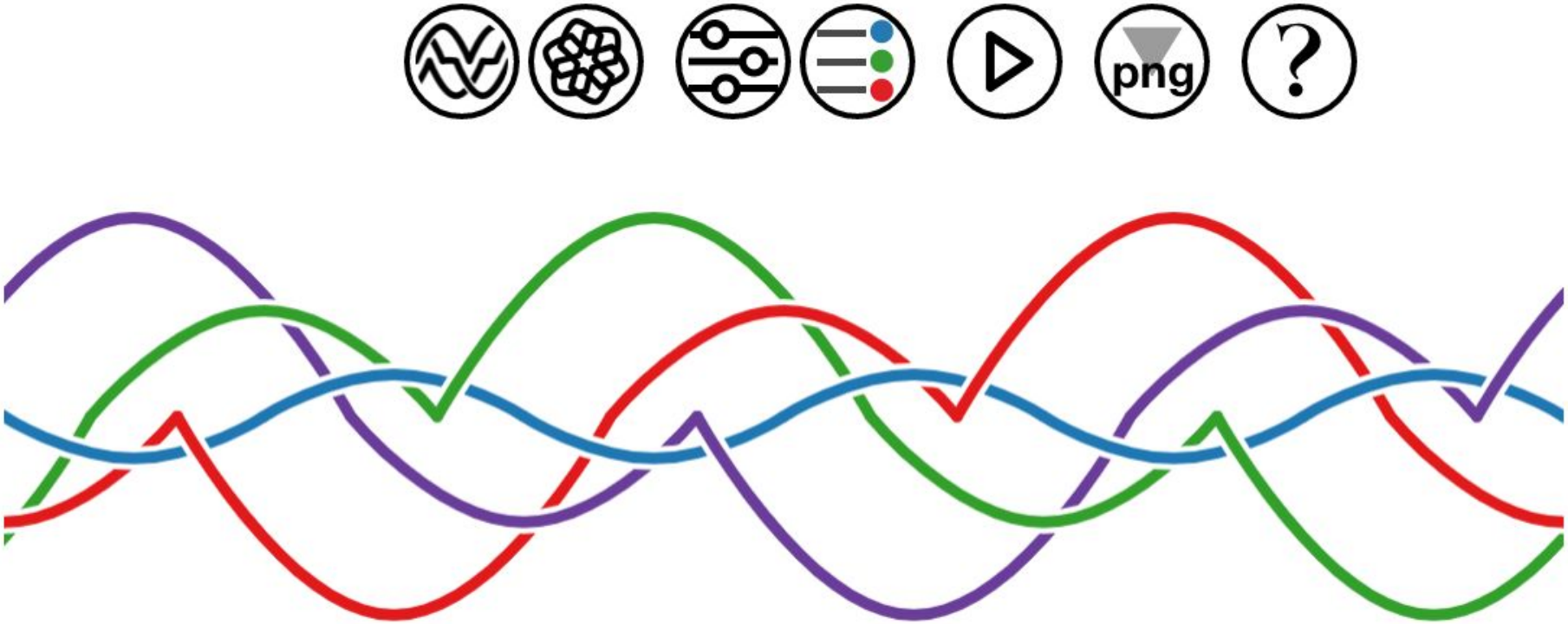
Population trends in Pittsburgh and the greater Phoenix metropolitan area (roughly Maricopa County) over the past 150-200 years.



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

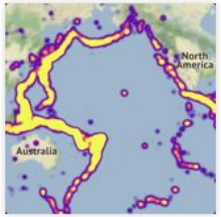

Candlestick Charts


Funnel and Ch

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

Maps


Mapbox Map Layers



Mapbox Density Heatmap

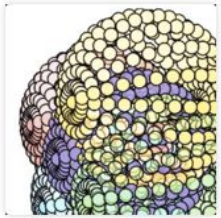

Choropleth Mapbox

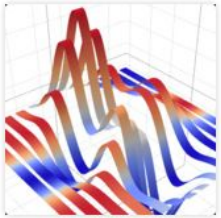

Lines on Maps

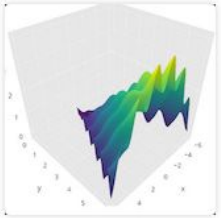

Bubble Maps

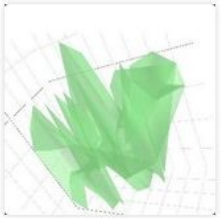
3D Charts

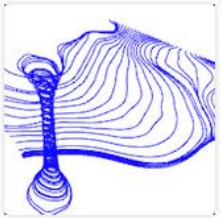
More 3D Charts »


3D Scatter Plots


Ribbon Plots

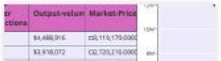

3D Surface Plots


3D Mesh Plots


3D Line Plots

Subplots

More Subplots »



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

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

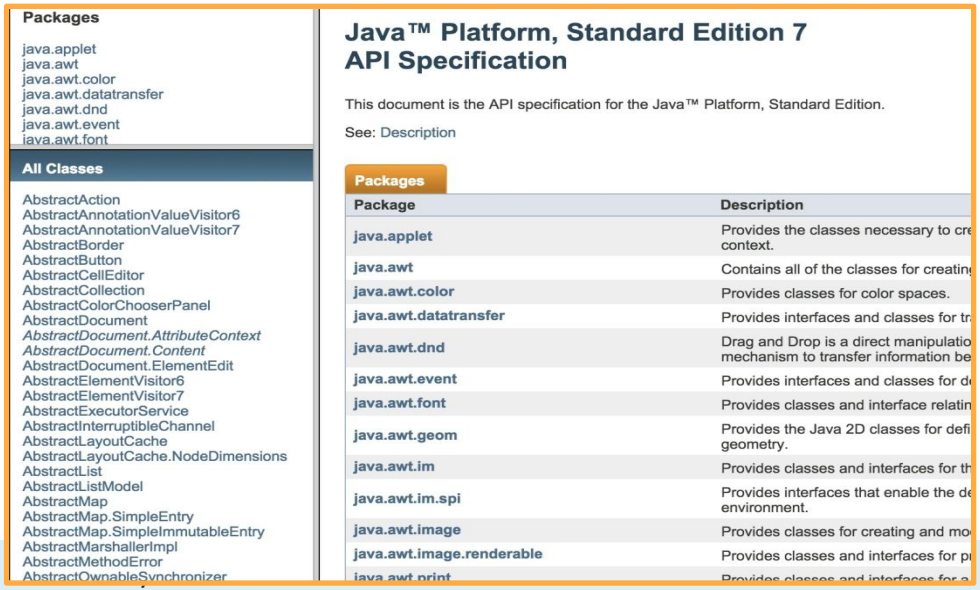
Breaking changes in ecosystems

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

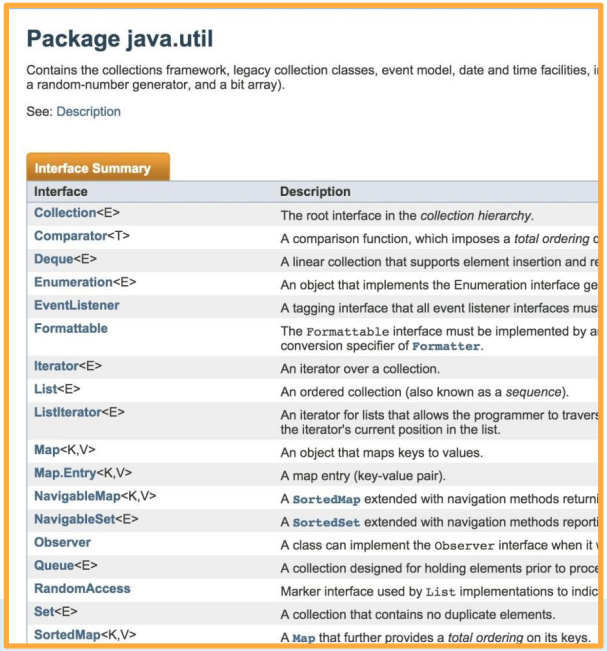
Java™ Platform, Standard Edition 7 API Specification

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

See: Description

Packages

Package	Description
java.applet	Provides the classes necessary to 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).

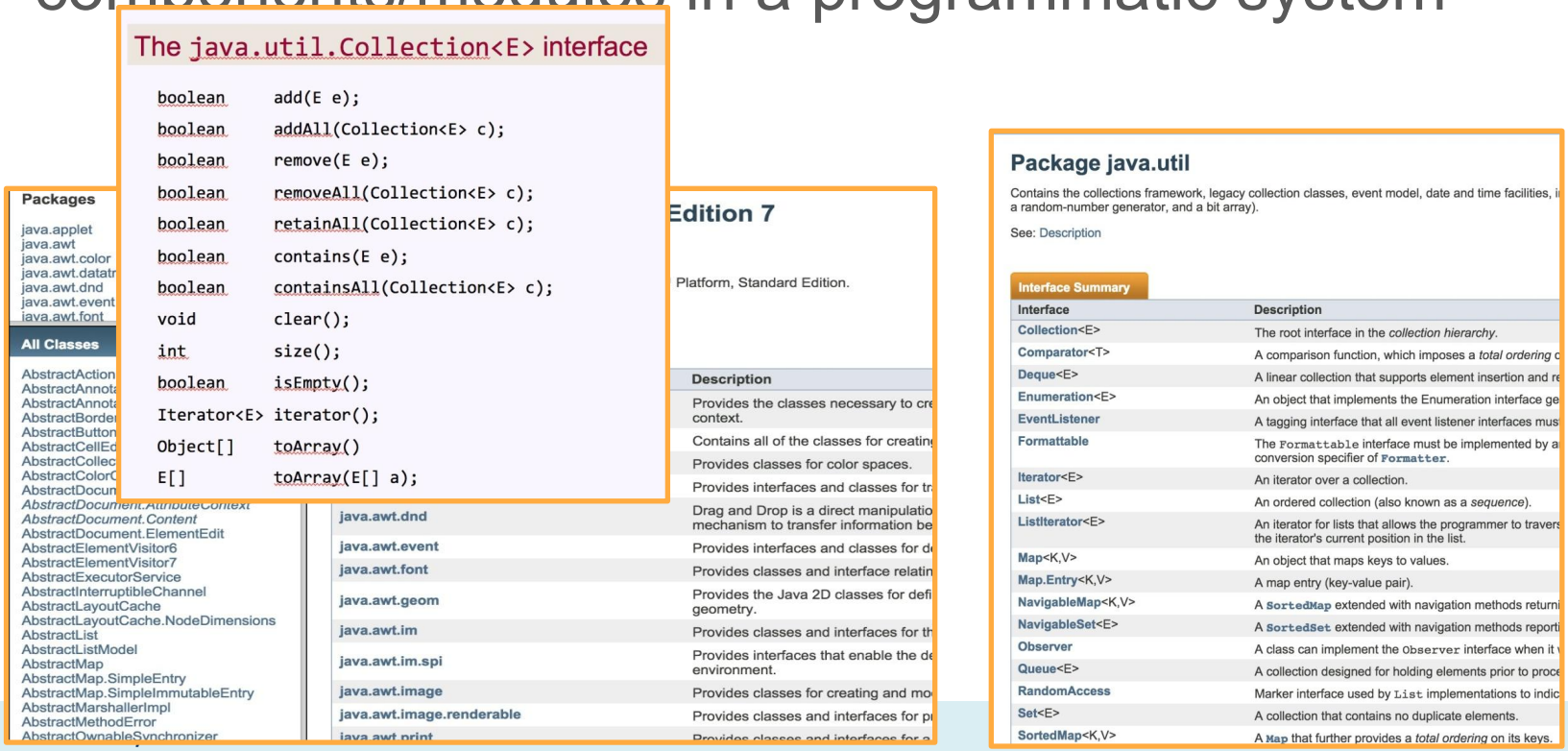
See: Description

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).

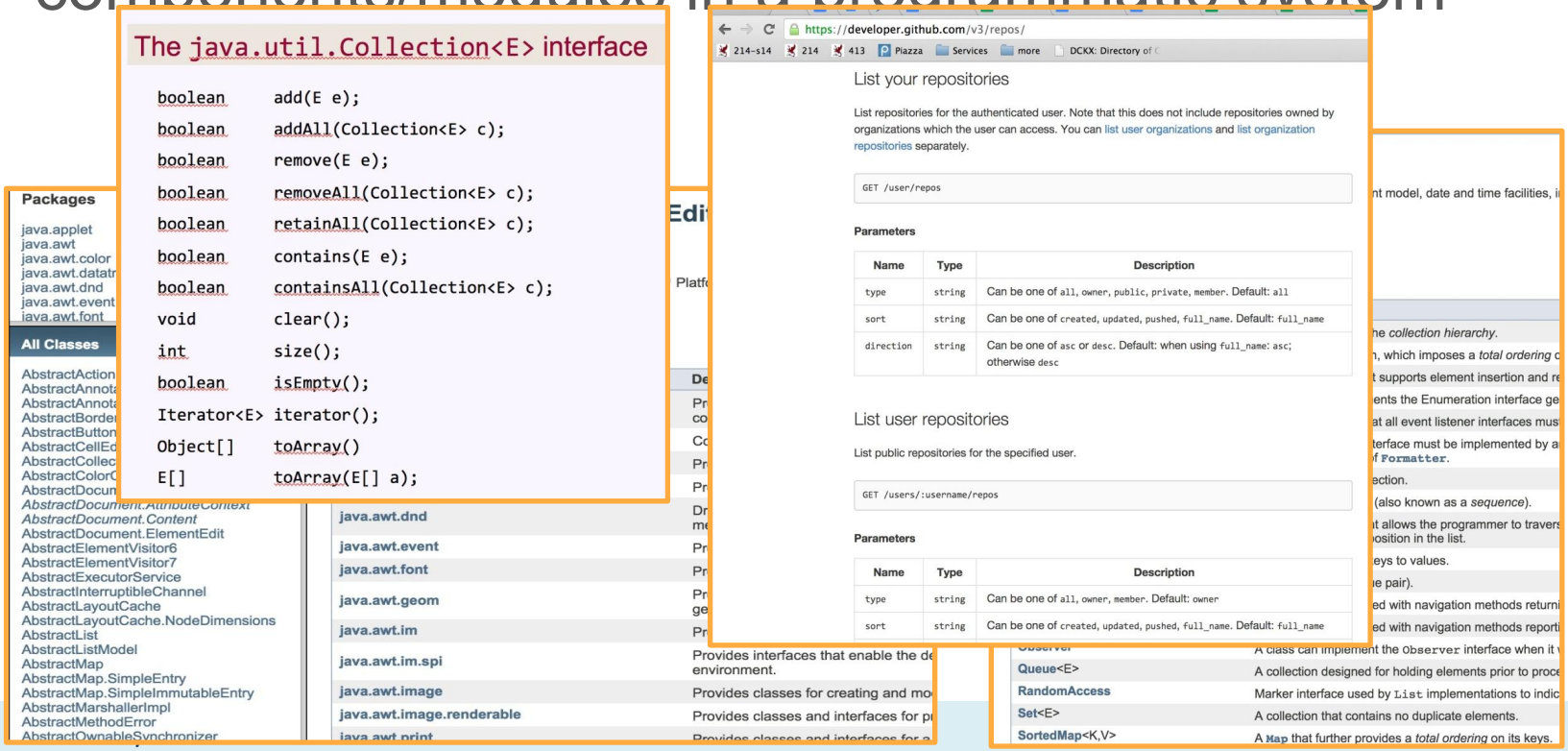
See: Description

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.

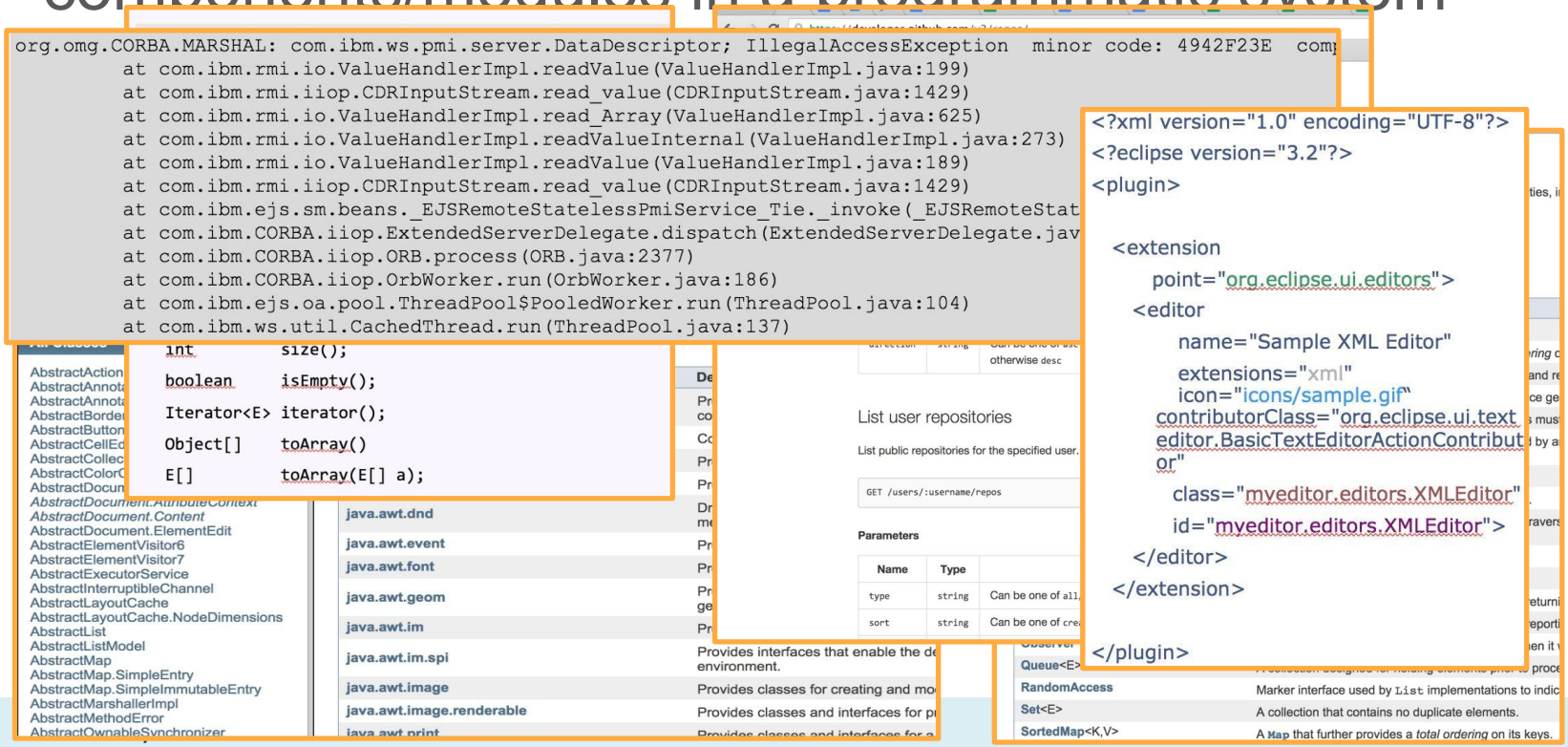
API: Application Programming Interface

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

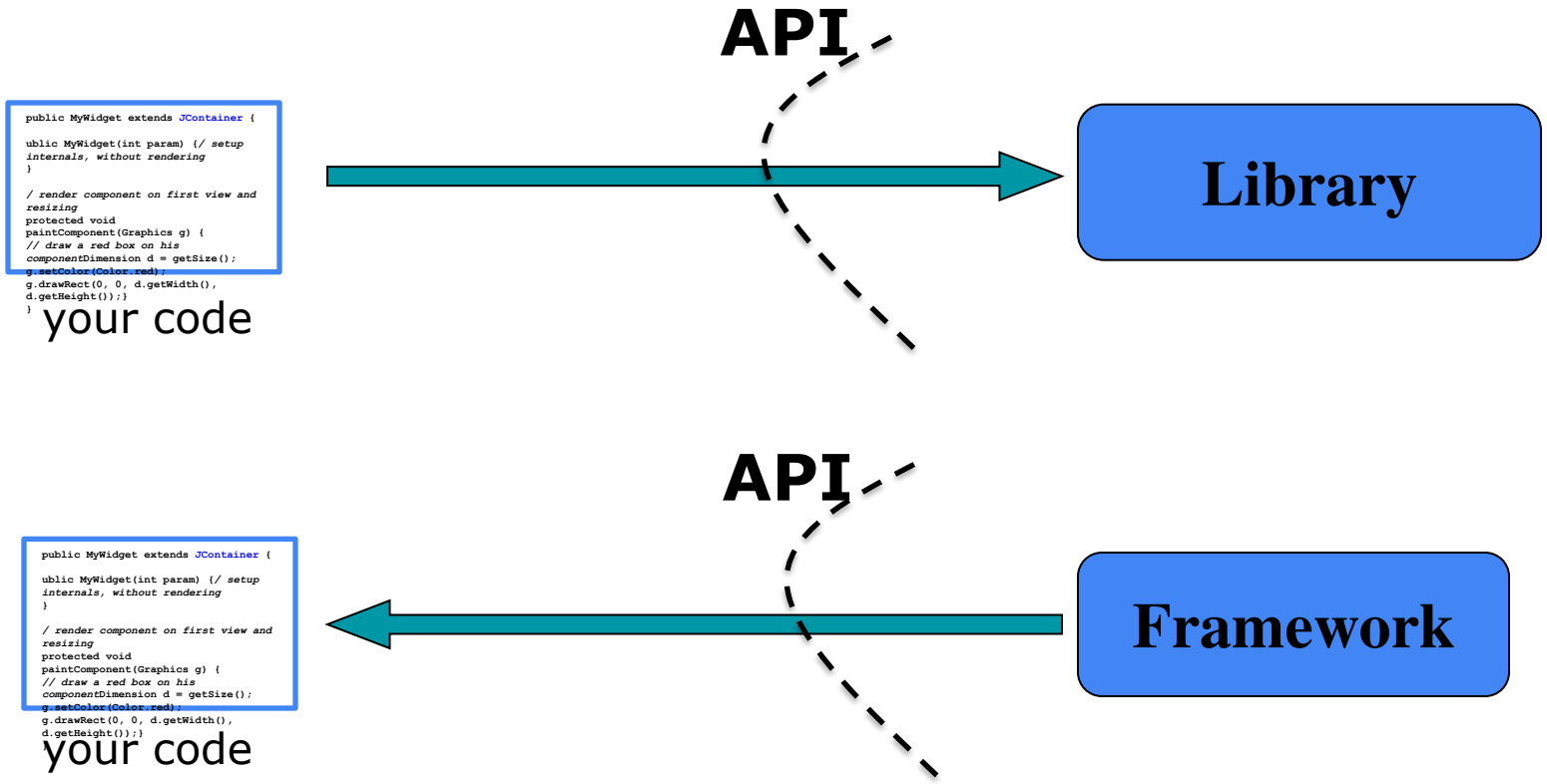


API: Application Programming Interface

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



Libraries and frameworks (and protocols!) define APIs



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

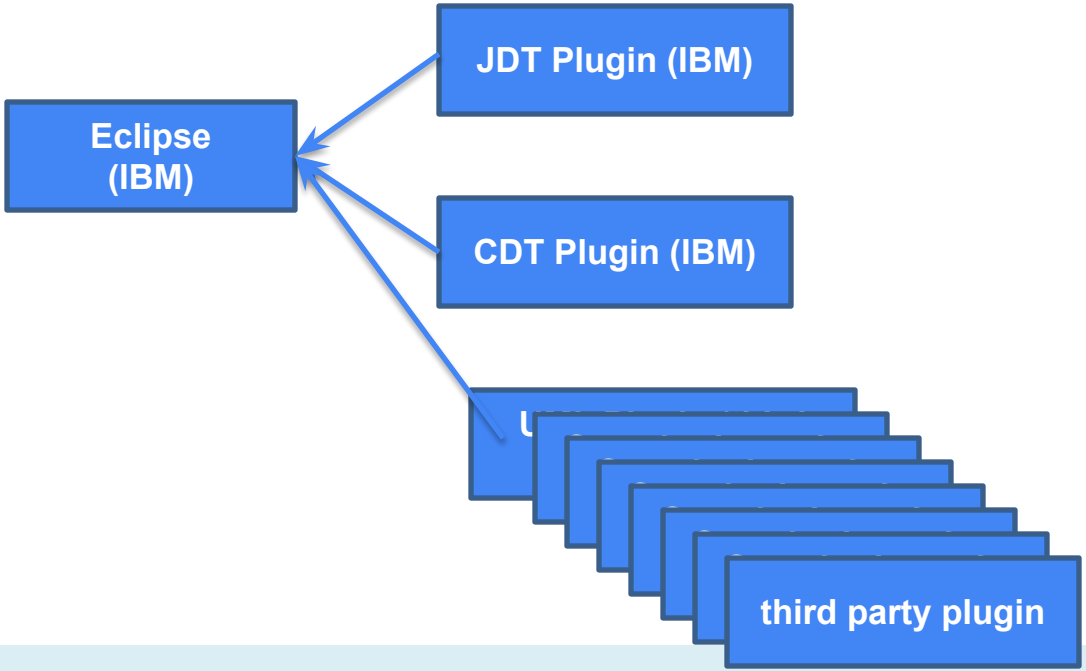
- Bad API can cause unending stream of support requests, inhibit forward movement
- Public APIs are forever – one chance to get it right

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

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

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/

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

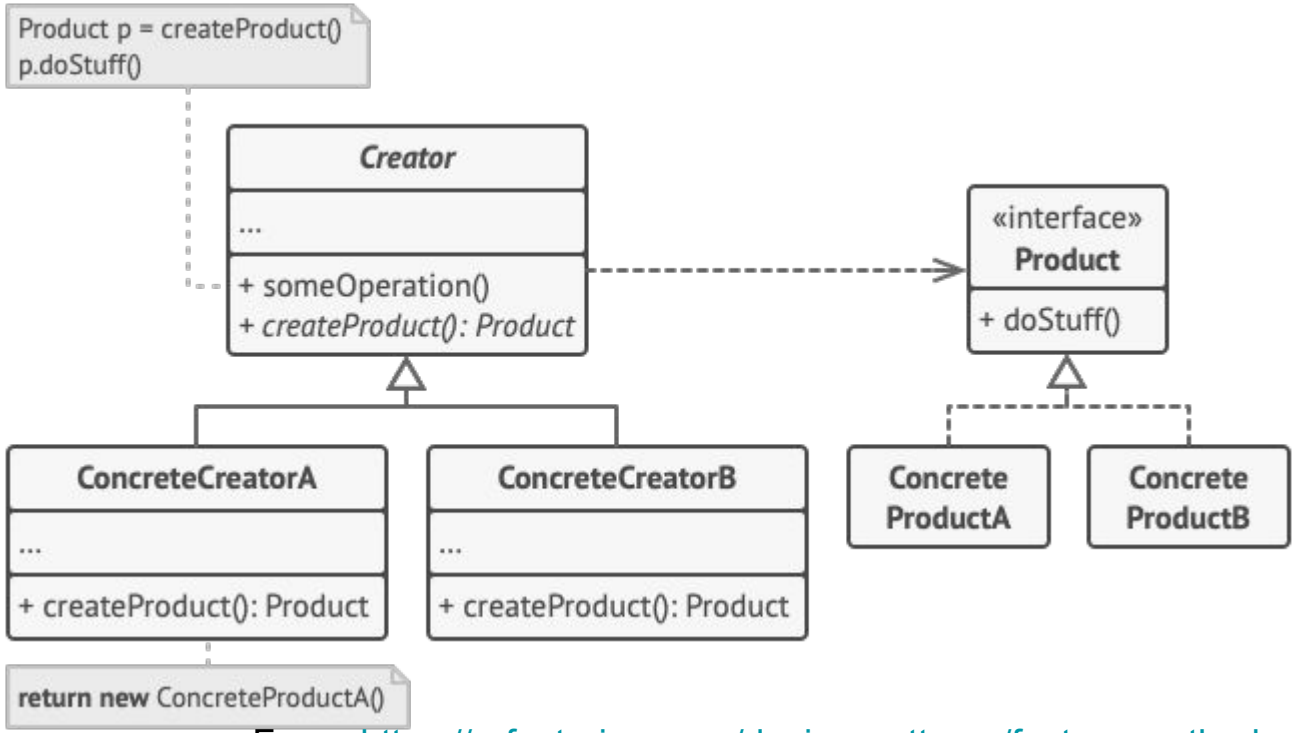
Be Aware: Unintentionally Leaking Implementation Details

- 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

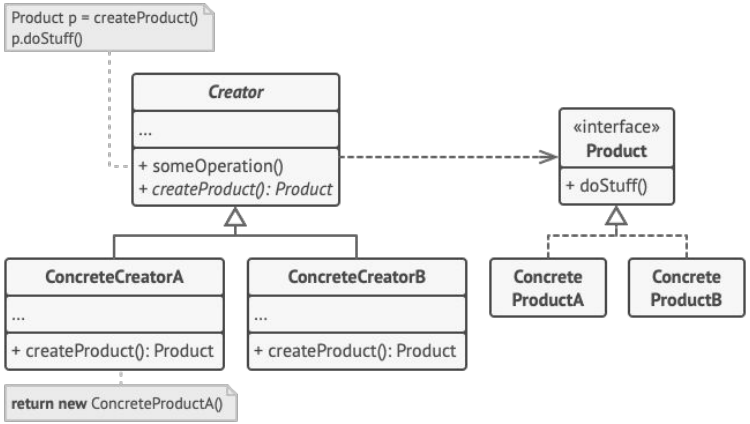
```
public class Rectangle {
        public Rectangle(Point e, Point f) …
}
// …
Point p1 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Point p2 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Rectangle r = new Rectangle(p1, p2);
```

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.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
- 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

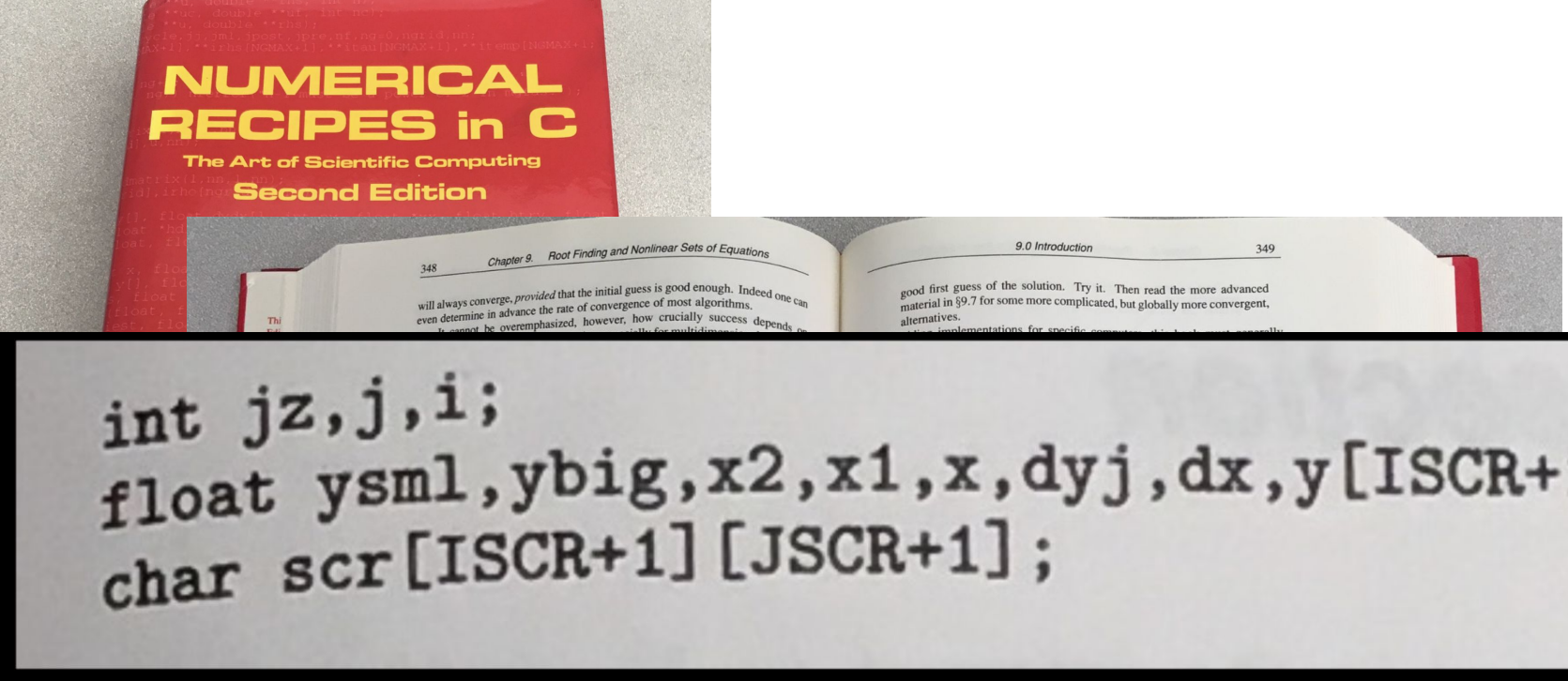
- Primary goals
  - **Client code should read like prose** (“easy to read”)
  - **Client code should mean what it says** (“hard to misread”)
  - **Client code should flow naturally** (“easy to write”)
- To that end, names should:
  - be largely self-explanatory
  - leverage existing knowledge
  - interact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



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.

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.

```
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)

```
    for (j=2;j<=(JSCR-1);j++)          Fill interior with blanks.
        scr[i][j]=BLANK;
}
dx=(x2-x1)/(ISCR-1);
x=x1;
ysml=ybig=0.0;                         Limits will include 0.
for (i=1;i<=ISCR;i++) {                Evaluate the function at equal intervals.
    y[i]=(*fx)(x);                         Find the largest and smallest val-
    if (y[i] < ysml) ysml=y[i];            ues.
    if (y[i] > ybig) ybig=y[i];
    x += dx;
}
if (ybig == ysml) ybig=ysml+1.0;       Be sure to separate top and bottom.
dyj=(JSCR-1)/(ybig-ysml);
jz=1-(int) (ysml*dyj);                 Note which row corresponds to 0.
for (i=1;i<=ISCR;i++) {                Place an indicator at function height and
    scr[i][jz]=ZERO;                       0.
    j=1+(int) ((y[i]-ysml)*dyj);
    scr[i][j]=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--) {            Display.
    printf("%12s"," ");
    for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
    printf("\n");
```

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 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

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

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