Principles of Software Construction: Objects, Design, and Concurrency
API Design

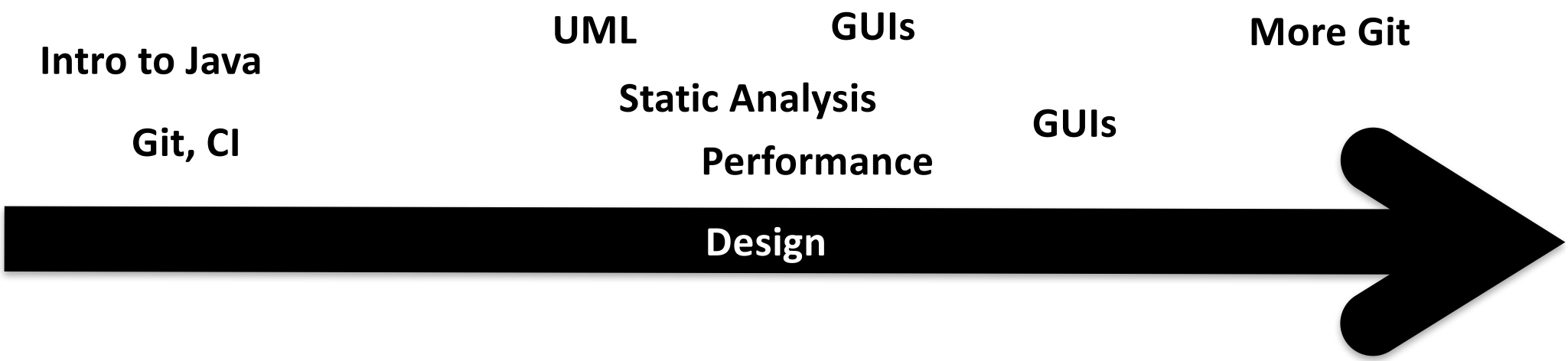
Christian Kaestner **Bogdan Vasilescu**

Many slides stolen with permission from Josh Bloch (thanks!)

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 4c due **tonight**
- Homework 4b feedback available soon
- Homework 5 released tomorrow
  - Work in teams



Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	Distributed Systems in a Nutshell
	Design for Reuse at Scale: Frameworks and APIs	

Agenda

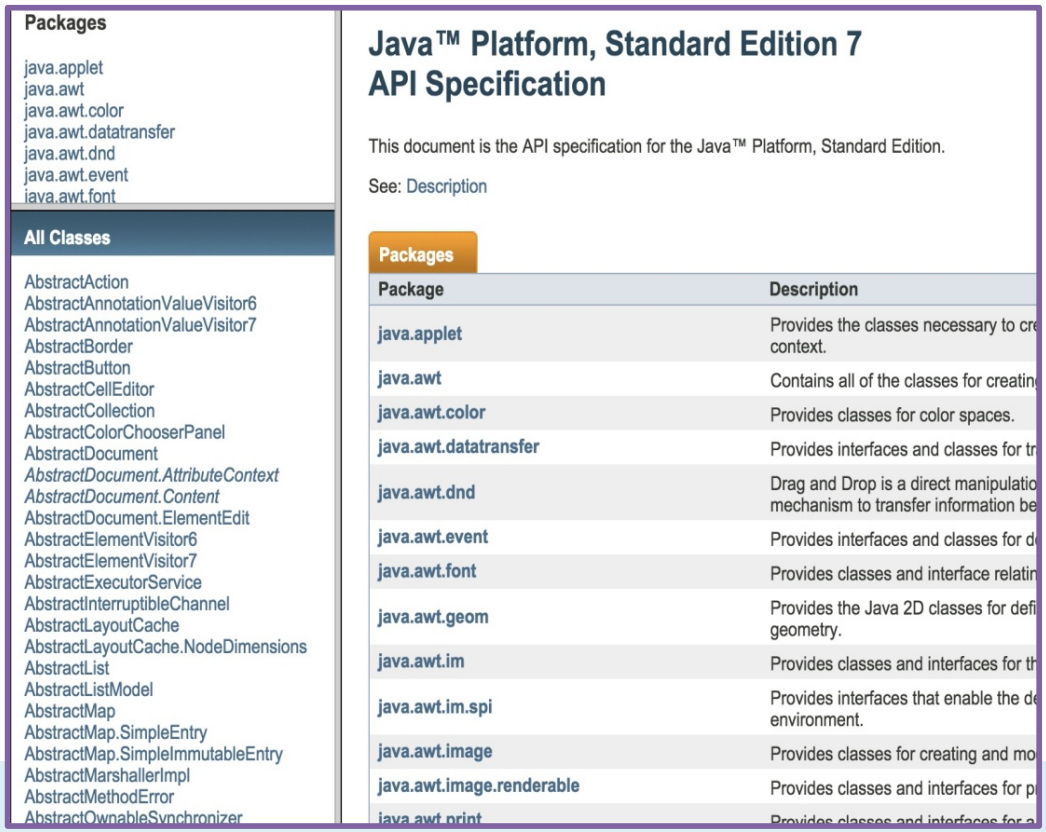
- Introduction to APIs: Application Programming Interfaces
- An API design process
- Key design principle: Information hiding
- Concrete advice for user-centered design

Learning goals

- Understand and be able to discuss the similarities and differences between API design and regular software design
  - Relationship between libraries, frameworks and API design
  - Information hiding as a key design principle
- Acknowledge, and plan for failures as a fundamental limitation on a design process
- Given a problem domain with use cases, be able to plan a coherent design process for an API for those use cases, e.g., "Rule of Threes"

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

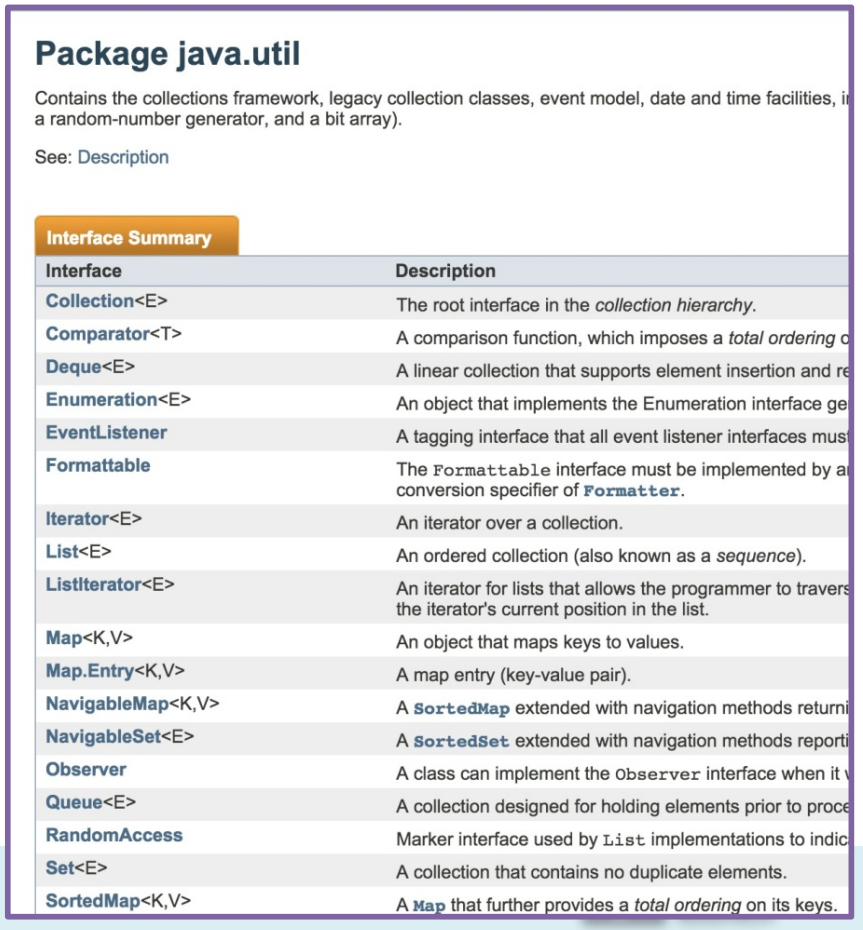
Java™ Platform, Standard Edition 7 API Specification

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

See: Description

Packages

Package	Description
java.applet	Provides the classes necessary to cr 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 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

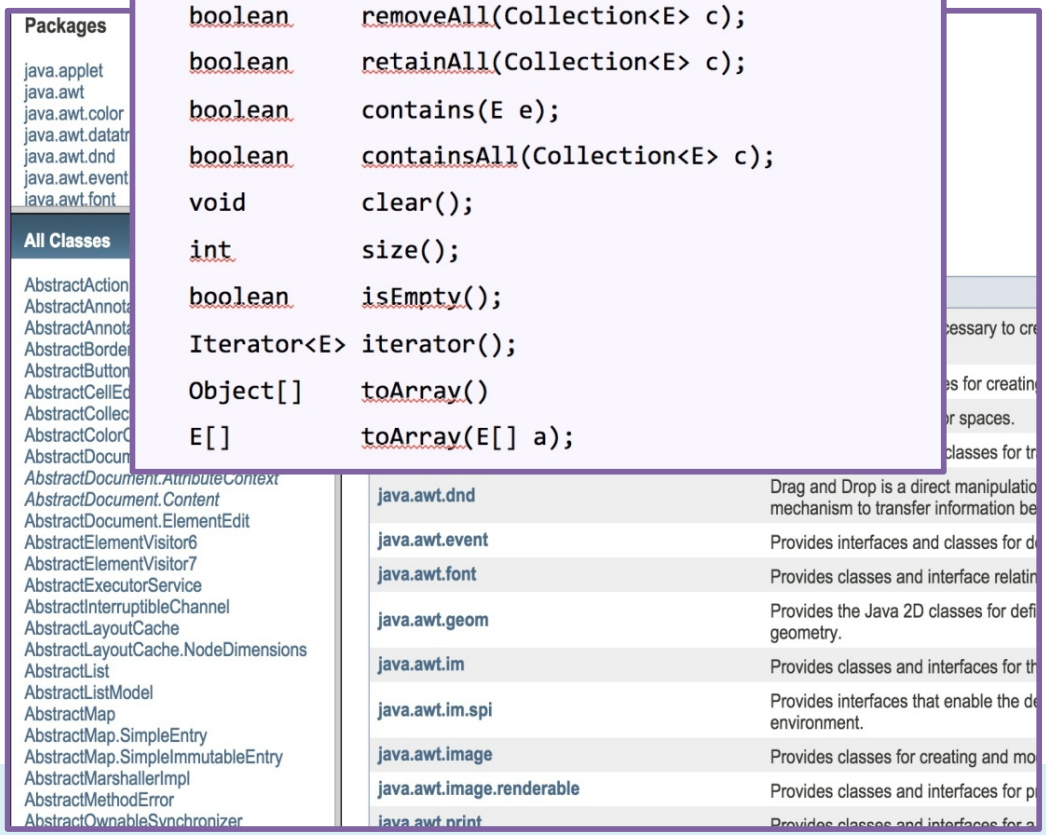
Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* c
Deque<E>	A linear collection that supports element insertion and re
Enumeration<E>	An object that implements the Enumeration interface ge
EventListener	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
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.

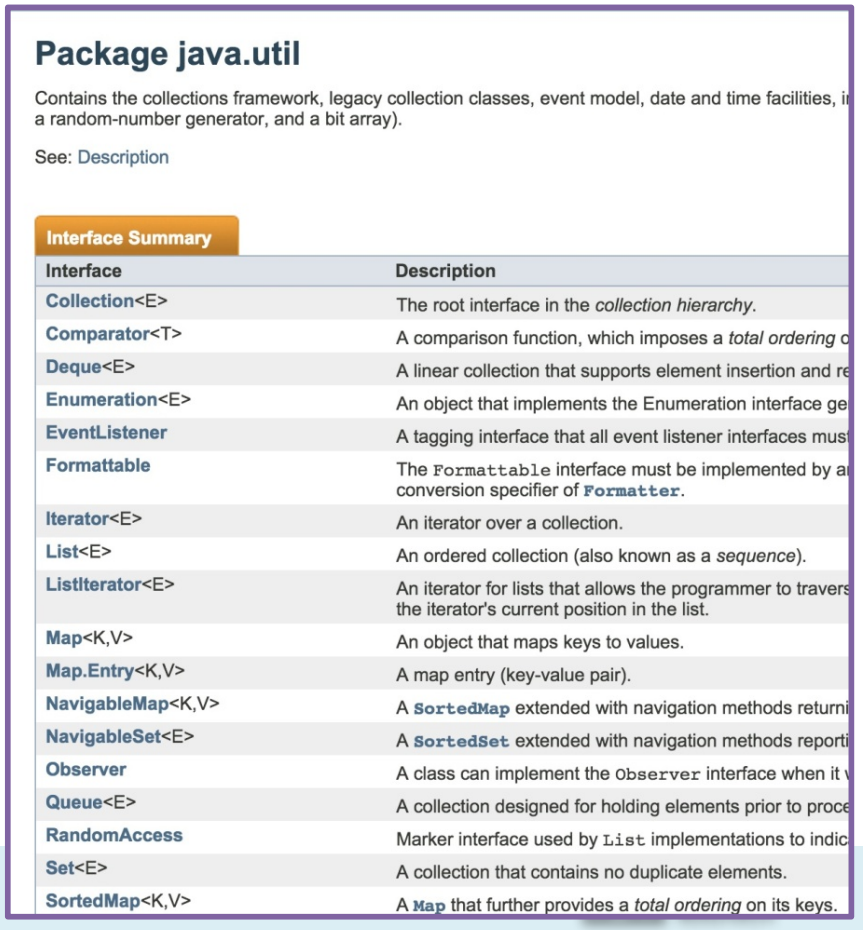
API: Application Programming Interface

- An API defines the boundary between
  s 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);
```





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).

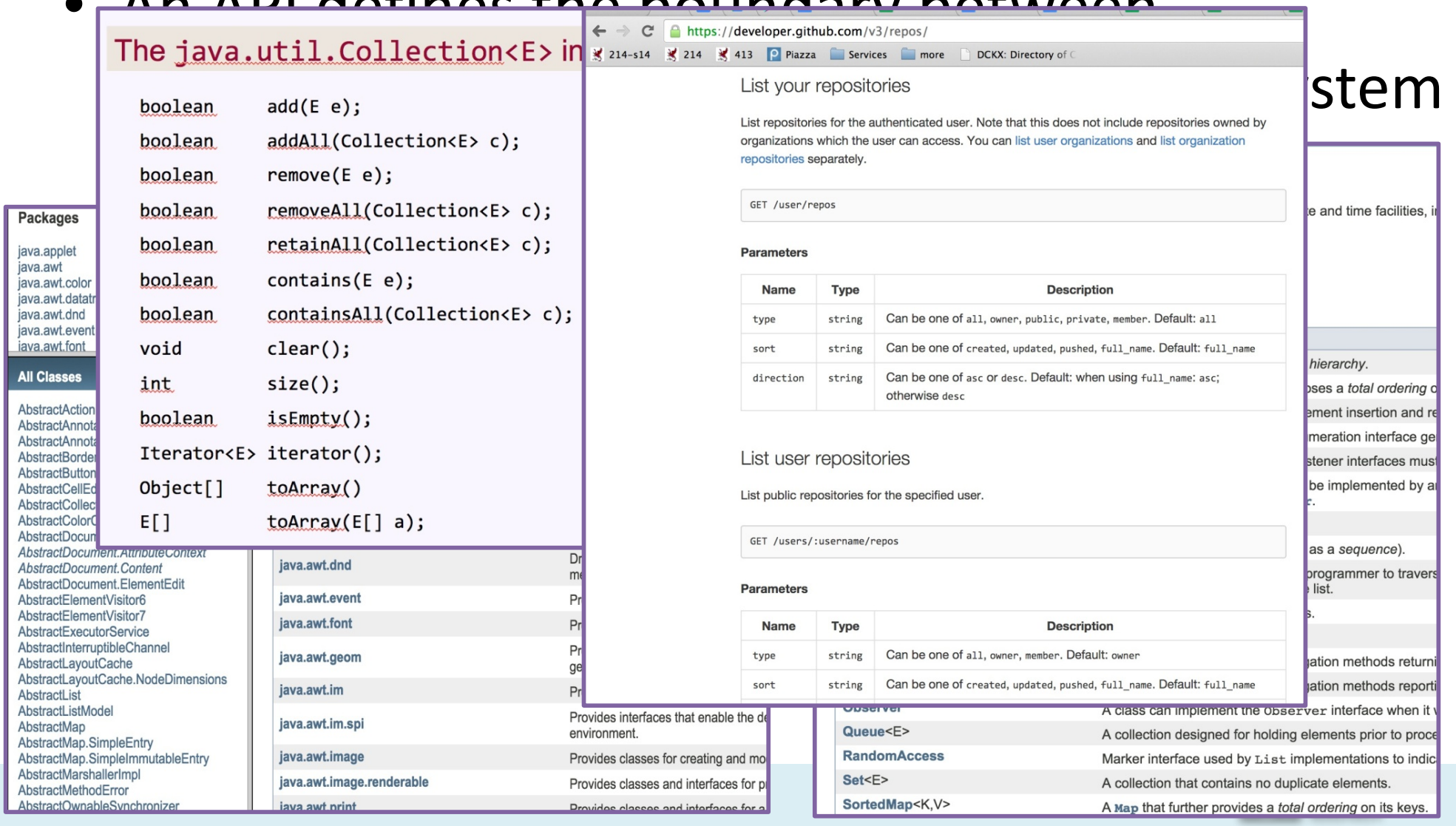
See: Description

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 must
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
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.

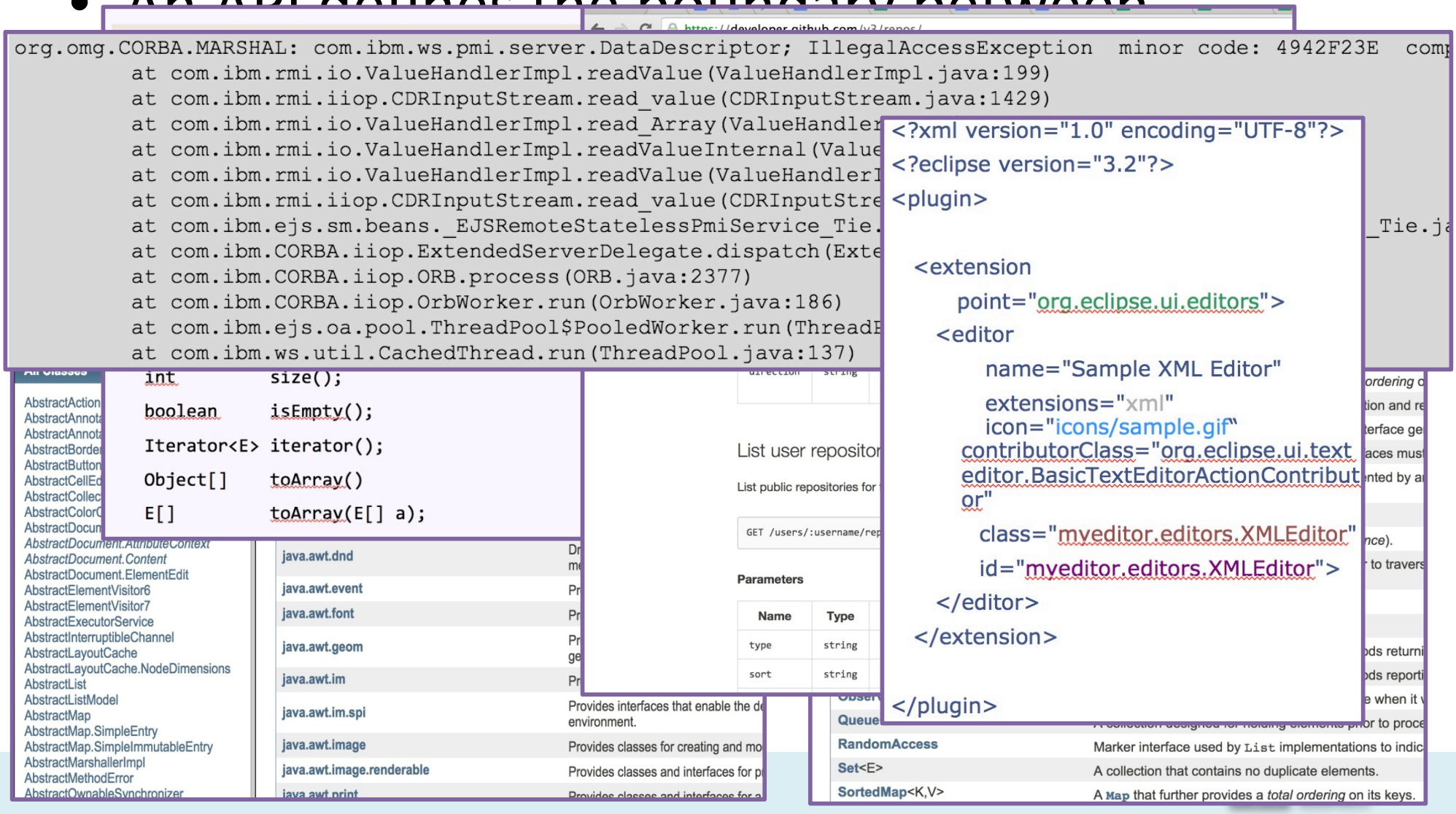
API: Application Programming Interface

- An API defines the boundary between

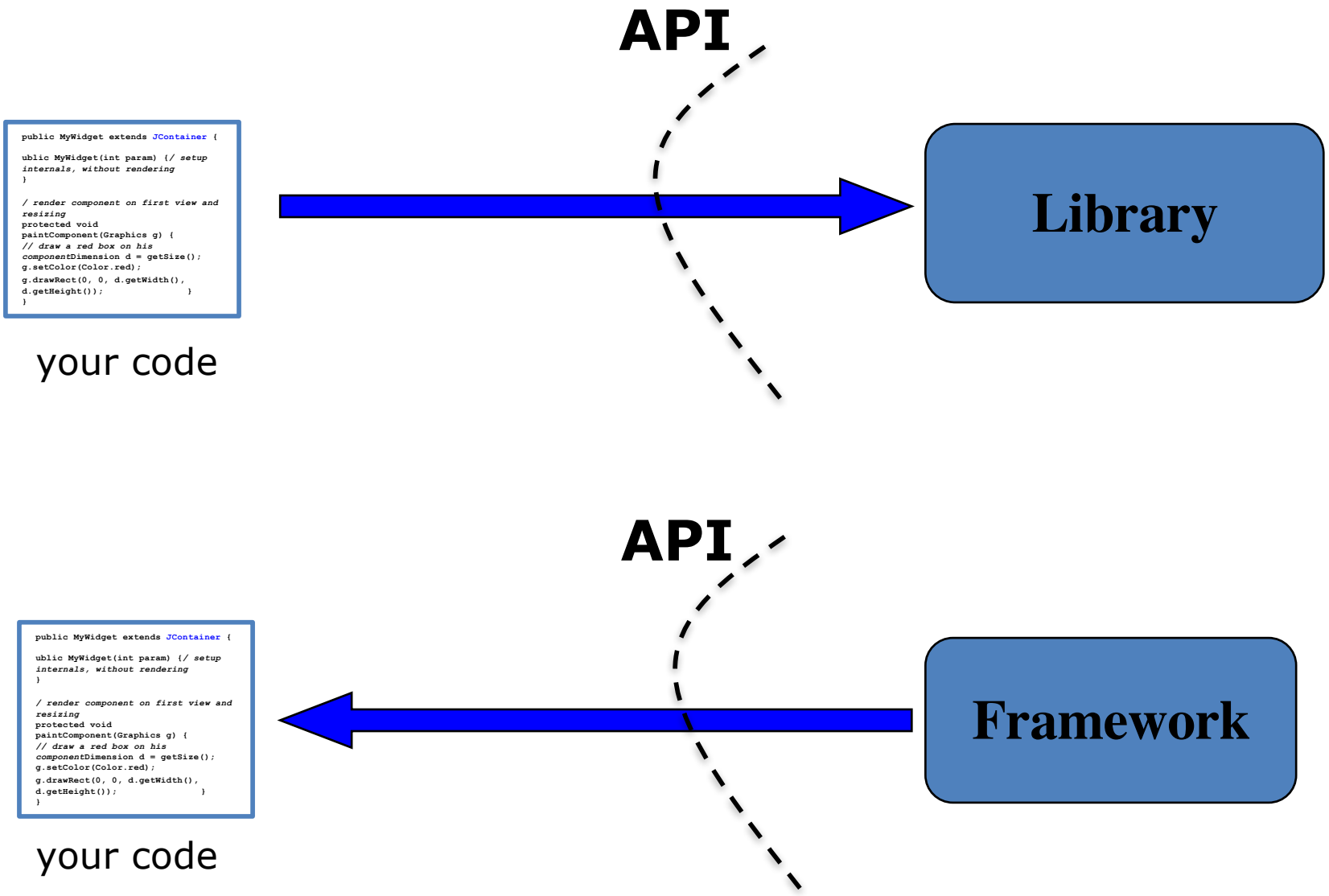


API: Application Programming Interface

- An API defines the boundary between



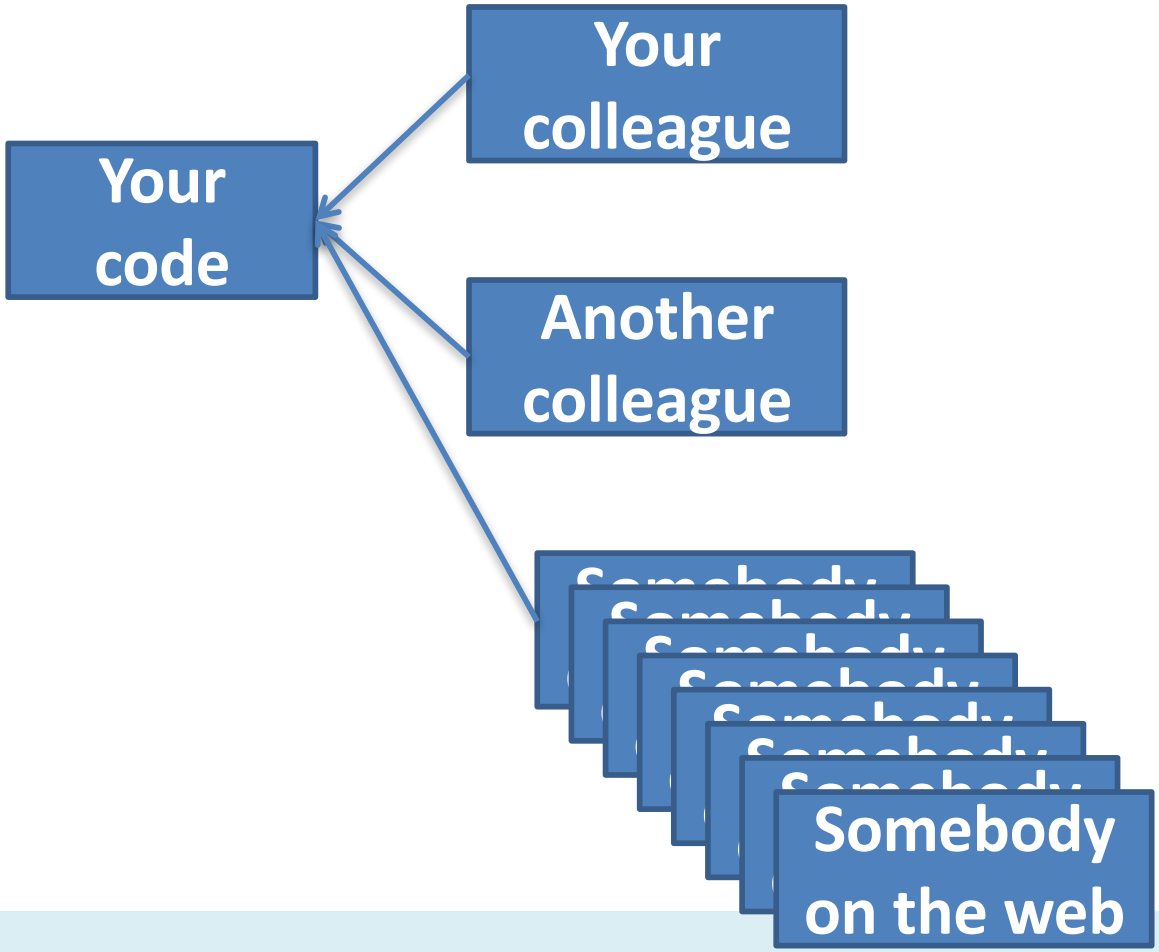
Libraries and frameworks both define APIs



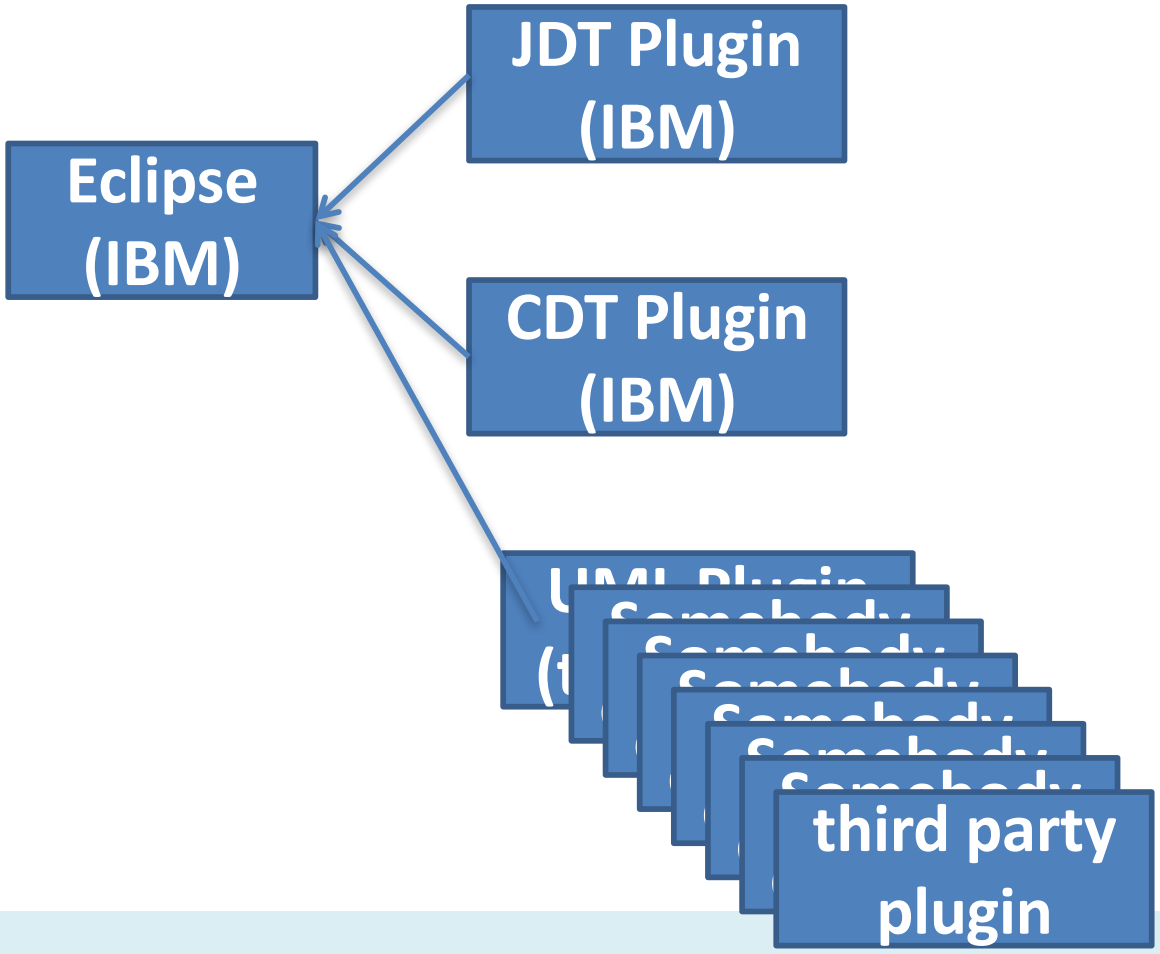
Why is API design important?

- APIs can be among your greatest assets
  - Users invest heavily: acquiring, writing, learning
  - Cost to **stop** using an API can be prohibitive
  - Successful public APIs capture users
- Can also be among your greatest liabilities
  - Bad API can cause unending stream of support calls
  - Can inhibit ability to move forward

Public APIs are forever

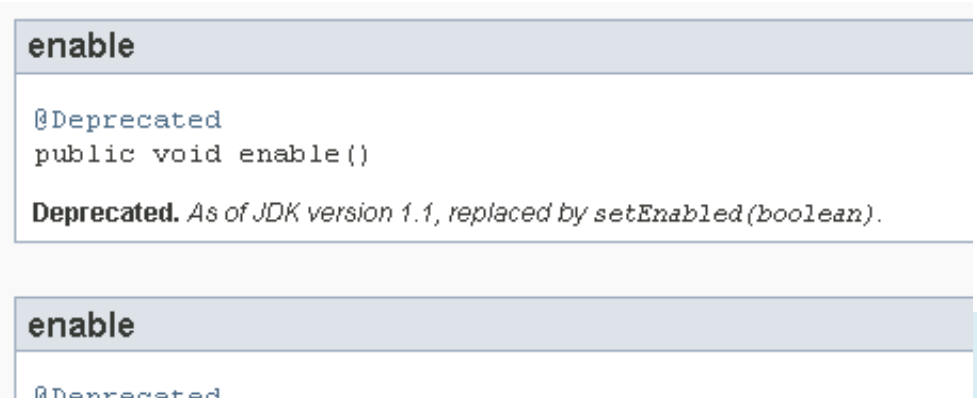


Public APIs are forever



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

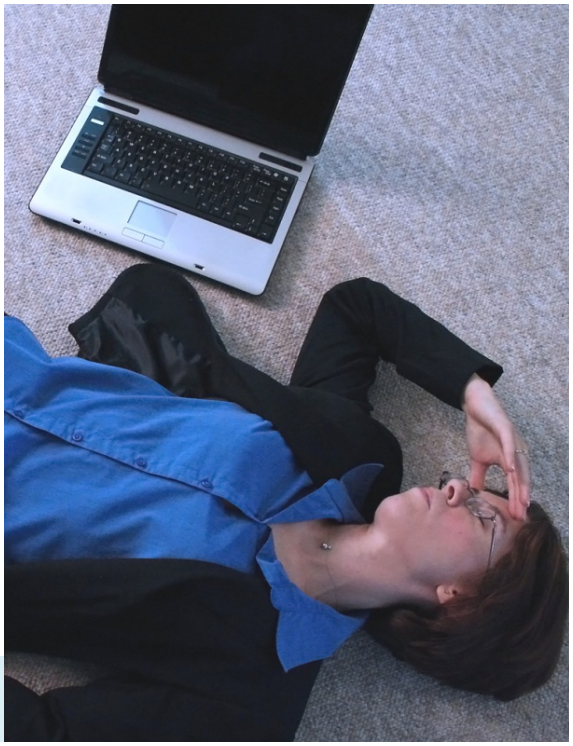


awt.Component,
deprecated since Java 1.1
still included in 7.0

Good vs Bad APIs

- Lots of reuse
  - including from yourself
- Lots of users/customers
- User buy-in and lock-in

- Lost productivity, inefficient reuse
- Maintenance and customer support liability



Characteristics of a good API

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

Outline for today

- The Process of API Design
- Key design principle: Information hiding
- Concrete advice for user-centered 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 (via Josh Bloch), *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."

Outline

- The Process of API Design
- Key design principle: Information hiding
- Concrete advice for user-centered design

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

Key design principle: Information hiding

- 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

Applying Information Hiding: Fields vs Getter/Setter Functions

```
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() { /* … */ }
}
```

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: Interface vs. Class Types

```
public class Rectangle {
 public Rectangle(Point e, Point f) …
}
```

vs.

```
public class Rectangle {
 public Rectangle(PolarPoint e, PolarPoint f) …
}
```

Still…

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

Still…

```
public class Rectangle {
 public Rectangle(Point e, Point f) …
}
…
Point p1 = new PolarPoint(…);
Point p2 = new PolarPoint(…);
Rectangle r = new Rectangle(p1, p2);
…
Point p3 = new PolarPoint(…);
Point p4 = new PolarPoint(…);
Rectangle r2 = new Rectangle(p3, p4);
…
```

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);
```

Applying Information hiding: Factories

- Consider implementing a factory method instead of a constructor
- Factory methods provide additional flexibility
  - Can be overridden
  - Can return instance of any subtype; hides dynamic type of object
  - Can have a descriptive method name

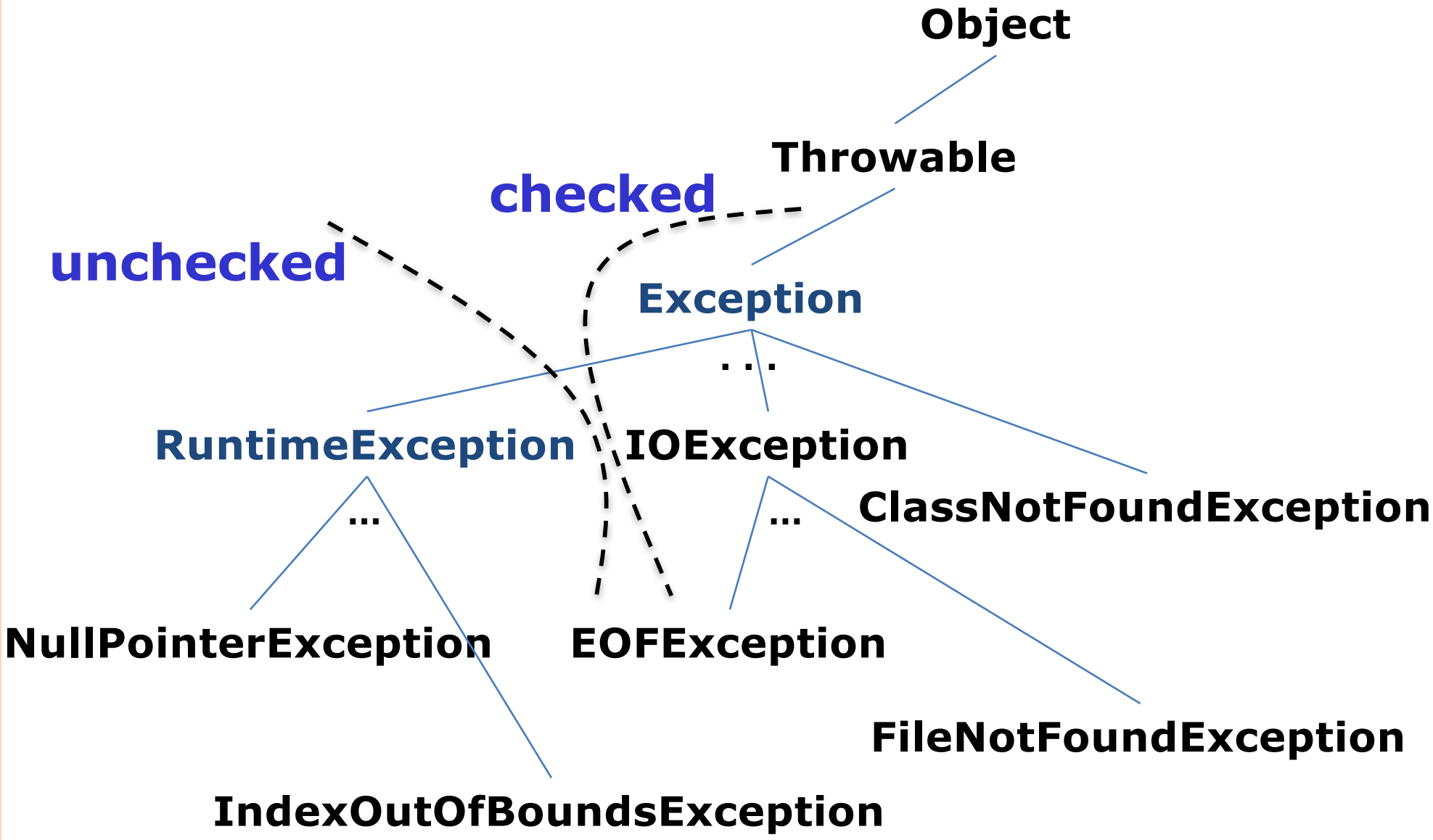
Applying Information Hiding: Hide Client Boilerplate Code

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

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out)throws IOException{
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```

The exception hierarchy in Java



Applying Information Hiding: Hide Client Boilerplate Code

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

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out)throws IOException{
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```

Won't compile

Applying Information Hiding: Hide Information Details

- Subtle leaks of implementation details through
  - Documentation
    - E.g., **do not specify hash functions**
  - Implementation-specific return types / exceptions
    - E.g., Phone number API that throws SQL exceptions
  - Output formats
    - E.g., `implements Serializable`
- Lack of documentation → Implementation becomes specification → no hiding

Outline

- The Process of API Design
- Key design principle: Information hiding
- Concrete advice for user-centered design

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

Achieving flexibility in use while remaining simple and intuitive: 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"
- Good examples:
  - `java.util.*`
  - `java.util.Collections`

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();
 }
```

Unintuitive behavior: side effects

- User of API should not be surprised by behavior, aka "the principle of least astonishment"
  - It's worth extra implementation effort
  - It's even worth reduced performance

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

Good names drive good design

- Do what you say you do:
  - "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 (2)

- Follow language- and platform-dependent conventions, e.g.,
  - Typographical:
    - `get_x() vs. getX()`
    - `timer vs. Timer, HTTPServlet vs HttpServlet`
    - `edu.cmu.cs.cs214`
  - Gramatical (see next)

Good names drive good design (3)

- 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`

Good names drive good design (4)

- Use clear, specific naming conventions
  - `getX()` and `setX()` for simple accessors and mutators
  - `isX()` for simple boolean accessors
  - `computeX()` for methods that perform computation
  - `createX()` or `newInstance()` for factory methods
  - `toX()` for methods that convert the type of an object
  - `asX()` for wrapper of the underlying object

Good names drive good design (5)

- 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`

Do not violate Liskov's behavioral subtyping rules

- Use inheritance only for true subtypes
- Examples:

```
1)
class Stack extends Vector …


2)
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);
    …
}
```

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);
    }
    …
}
```

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: `TimerTask`

Mutability and performance

- `Component.getSize()` returns `Dimension`
- `Dimension` is mutable
- Each `getSize` call must allocate `Dimension`
- Causes millions of needless object allocations
- Alternative added in Java1.2 but old client code still slow: `getX()`, `getY()`
- Document mutability
  - Carefully describe state space
  - Make clear when it's legal to call which method

Overload method names judiciously

- Avoid ambiguous overloads for subtypes
  - Recall the subtleties of method dispatch:

```
public class Point() {
        private int x;
        private int y;
        public boolean equals(Point p) {
            return this.x == p.x && this.y == p.y;
        }
    }
```

- If you must be ambiguous, implement consistent behavior

```
public class TreeSet implements SortedSet {
  public TreeSet(Collection c);  // Ignores order.
  public TreeSet(SortedSet s);   // Respects order.
}
```

Use appropriate parameter & return types

- Favor interface types over classes for input
  - Provides flexibility, performance
- Use most specific possible input parameter type
  - Moves error from runtime to compile time
- Don't use `String` if a better type exists
  - Strings are cumbersome, error-prone, and slow
- Don't use floating point for monetary values
  - Binary floating point causes inexact results!
- Use `double` (64 bits) rather than `float` (32 bits)
  - Precision loss is real, performance loss negligible

Use consistent parameter ordering

- 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

Avoid 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
  - Create helper class to hold parameters
  - Builder 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);
}
```

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 to indicate exceptional conditions

- Don't force client to use exceptions for control flow

```
private byte[] a = new byte[CHUNK_SIZE];

void processBuffer (ByteBuffer buf) {
  try {
    while (true) {
      buf.get(a);
      processBytes(a, CHUNK_SIZE);
    }
  } catch (BufferUnderflowException e) {
    int remaining = buf.remaining();
    buf.get(a, 0, remaining);
    processBytes(a, remaining);
  }
}
```

- Conversely, don't fail silently

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

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.
}
```

Avoid return values that demand exceptional processing

- Return zero-length array or empty collection, not null

```
package java.awt.image;
public interface BufferedImageOp {
  // Returns the rendering hints for this operation,
  // or null if no hints have been set.
  public RenderingHints getRenderingHints();
}
```

- 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

```
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)
```

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);
  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

Summary

- Accept the fact that you, and others, will make mistakes
  - Use your API as you design it
  - Get feedback from others
  - Think in terms of use cases (domain engineering)
  - Hide information to give yourself maximum flexibility later
  - Design for inattentive, hurried users
  - Document religiously

BONUS: API REFACTORING

1. Sublist operations in `Vector`

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

- Not very powerful - supports only search
- Hard to use without documentation

Sublist operations refactored

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

- Extremely powerful - supports *all* operations
- Use of interface reduces conceptual weight
  - High power-to-weight ratio
- Easy to use without documentation

2. Thread-local variables

```
// Broken - inappropriate use of String as capability.
// Keys constitute a shared global namespace.
public class ThreadLocal {
    private ThreadLocal() { } // Non-instantiable

    // Sets current thread's value for named variable.
    public static void set(String key, Object value);

    // Returns current thread's value for named variable.
    public static Object get(String key);
}
```

Thread-local variables refactored (1)

```
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    public static class Key { Key() { } }

    // Generates a unique, unforgeable key
    public static Key getKey() { return new Key(); }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

- Works, but requires boilerplate code to use

```
static ThreadLocal.Key serialNumberKey = ThreadLocal.getKey();
ThreadLocal.set(serialNumberKey, nextSerialNumber());
System.out.println(ThreadLocal.get(serialNumberKey));
```

Thread-local variables refactored (2)

```
public class ThreadLocal<T> {
    public ThreadLocal() { }
    public void set(T value);
    public T get();

}
```

- Removes clutter from API and client code

```
static ThreadLocal<Integer> serialNumber =
    new ThreadLocal<Integer>();
serialNumber.set(nextSerialNumber());
System.out.println(serialNumber.get());
```