Principles of Software Construction: Objects, Design, and Concurrency

Part 4: Design for large-scale reuse

API design (and some libraries and frameworks…)

Charlie Garrod Michael Hilton

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4c due Thursday
- Homework 5 coming soon
  - Team sign-up deadline…
- Required reading due today
  - Effective Java, Items 40, 48, 50, and 52
- Midterm exam in class next Thursday (02 November)
  - Review session Wednesday, 01 Nov. 7-9 p.m. in HH B103

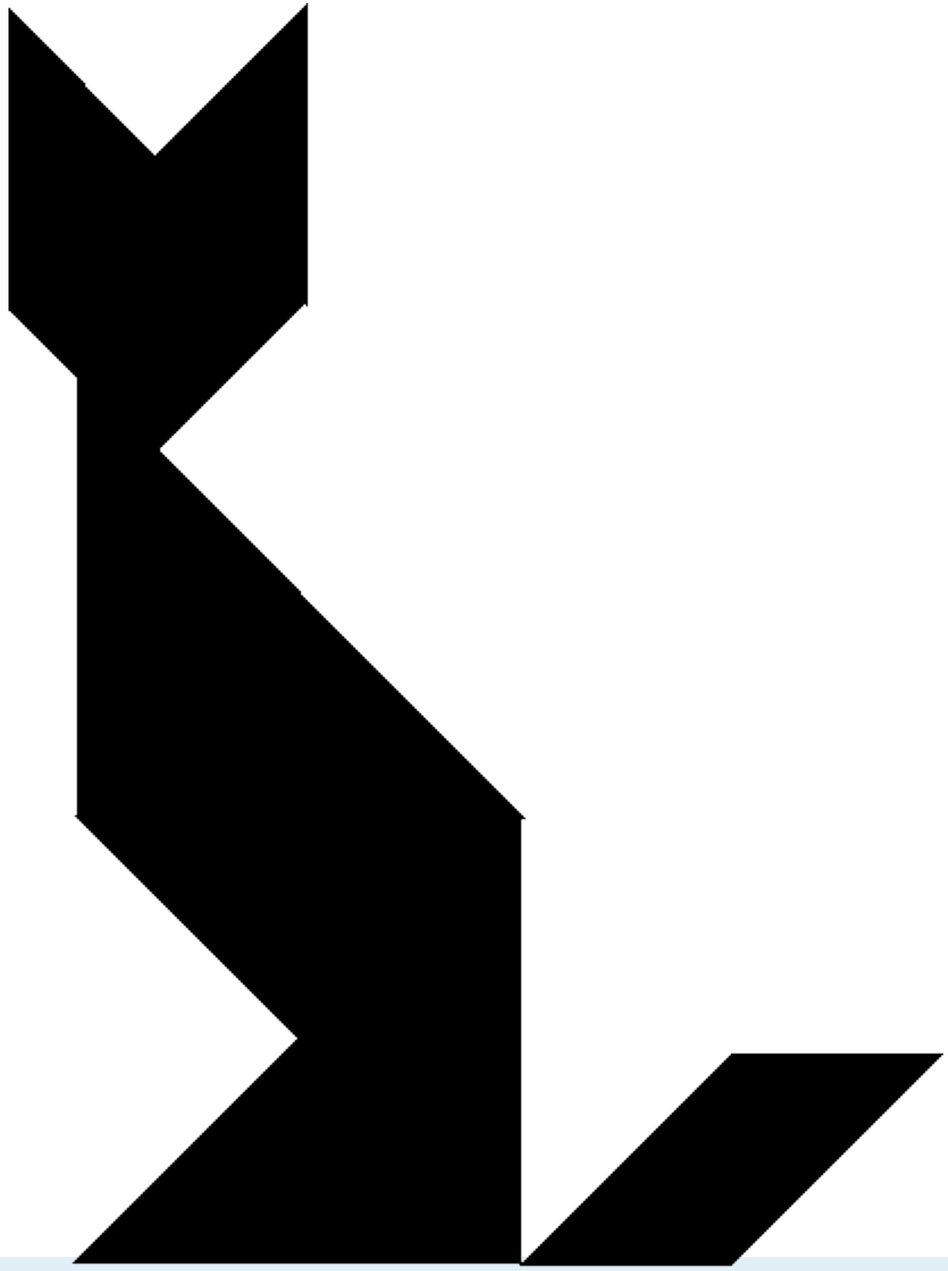
Key concepts from last Thursday

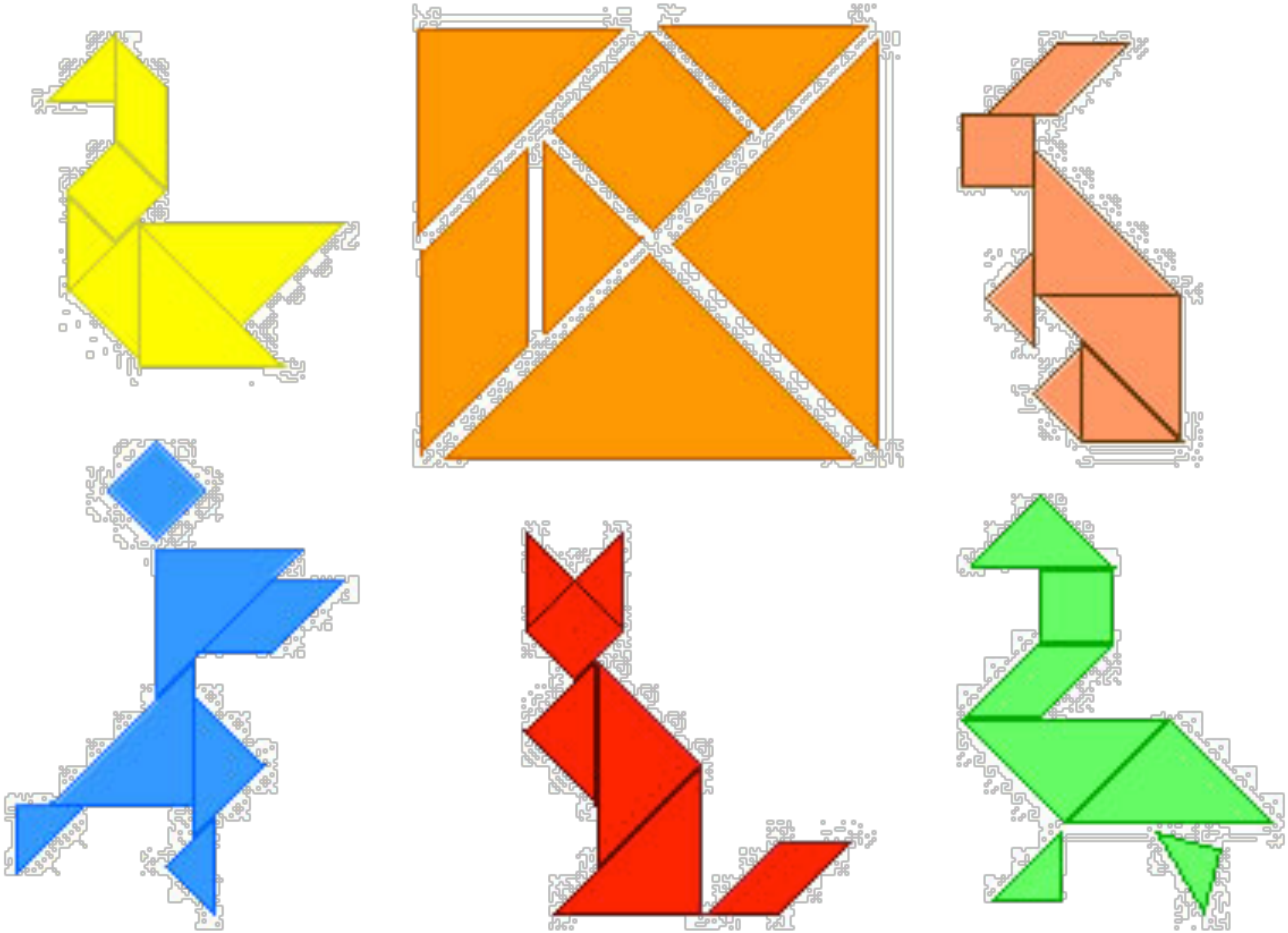
Key concepts from last Thursday

- Libraries vs. frameworks
- Whitebox vs blackbox frameworks

Framework design considerations

- Once designed there is little opportunity for change
- Key decision: Separating common parts from variable parts
  - What problems do you want to solve?
- Possible problems:
  - Too few extension points: Limited to a narrow class of users
  - Too many extension points: Hard to learn, slow
  - Too generic: Little reuse value





(one modularization: tangrams)

The use vs. reuse dilemma

- Large rich components are very useful, but rarely fit a specific need
- Small or extremely generic components often fit a specific need, but provide little benefit

"maximizing reuse minimizes use"

C. Szyperski

Domain engineering

- Understand users/customers in your domain
  - What might they need? What extensions are likely?
- Collect example applications before designing a framework
- Make a conscious decision what to support
  - Called *scoping*
- e.g., the Eclipse policy:
  - Interfaces are internal at first
    - Unsupported, may change
  - Public stable extension points created when there are at least two distinct customers

Typical framework design and implementation

- Define your domain
  - Identify potential common parts and variable parts
- Design and write sample plugins/applications
- Factor out & implement common parts as framework
- Provide plugin interface & callback mechanisms for variable parts
  - Use well-known design principles and patterns where appropriate…
- Get lots of feedback, and iterate

Not discussed here (yet!?)

- Framework implementation details
  - Mechanics of running the framework
  - Mechanics of loading plugins

This week: API design

- An API design process
- The key design principle: information hiding
- Concrete advice for user-centered design

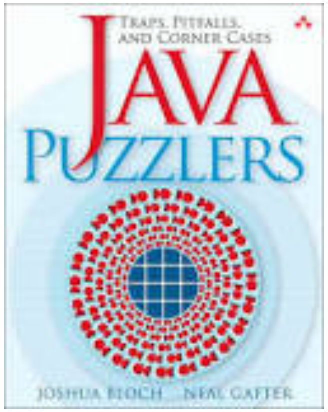
Based heavily on "How to Design a Good API and Why it Matters" by Josh Bloch.

If you have "Java" in your résumé you should own *Effective Java*.


1. “Time for a Change” (2002)

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```



What does it print?

(a) `0.9`
(b) `0.90`
(c) It varies
(d) None of the above

```
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```

What does it print?

(a) `0.9`

(b) `0.90`

(c) It varies

(d) None of the above: `0.8999999999999999`

Decimal values can't be represented exactly by `float` or `double`

Another look

```
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```

How do you fix it?

```
// You could fix it this way...
import java.math.BigDecimal;
public class Change {
    public static void main(String args[]) {
        System.out.println(
            new BigDecimal("2.00").subtract(
                new BigDecimal("1.10")));
    }
}
```

Prints 0.90

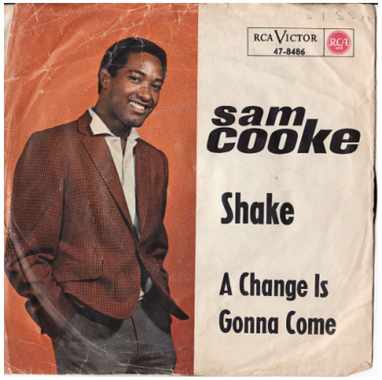
```
// ...or you could fix it this way
public class Change {
    public static void main(String args[]) {
        System.out.println(200 - 110);
    }
}
```

Prints 90

The moral

- Avoid `float` and `double` where exact answers are required
  - For example, when dealing with money
- Use `BigDecimal`, `int`, or `long` instead

2. "A Change is Gonna Come"



If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```

What does it print?

(a) `0.9`
(b) `0.90`
(c) `0.8999999999999999`
(d) None of the above

```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```

What does it print?

(a) `0.9`

(b) `0.90`

(c) `0.8999999999999999`

(d) None of the above:

`0.899999999999999911182158029987476766109466552734375`

We used the wrong `BigDecimal` constructor

Another look

The spec says:

```
public BigDecimal(double val)
```

Translates a double into a BigDecimal which is the exact decimal representation of the double's binary floating-point value.

```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```

How do you fix it?

```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal("2.00");
        BigDecimal cost = new BigDecimal("1.10");
        System.out.println(payment.subtract(cost));
    }
}
```

Prints `0.90`

The moral

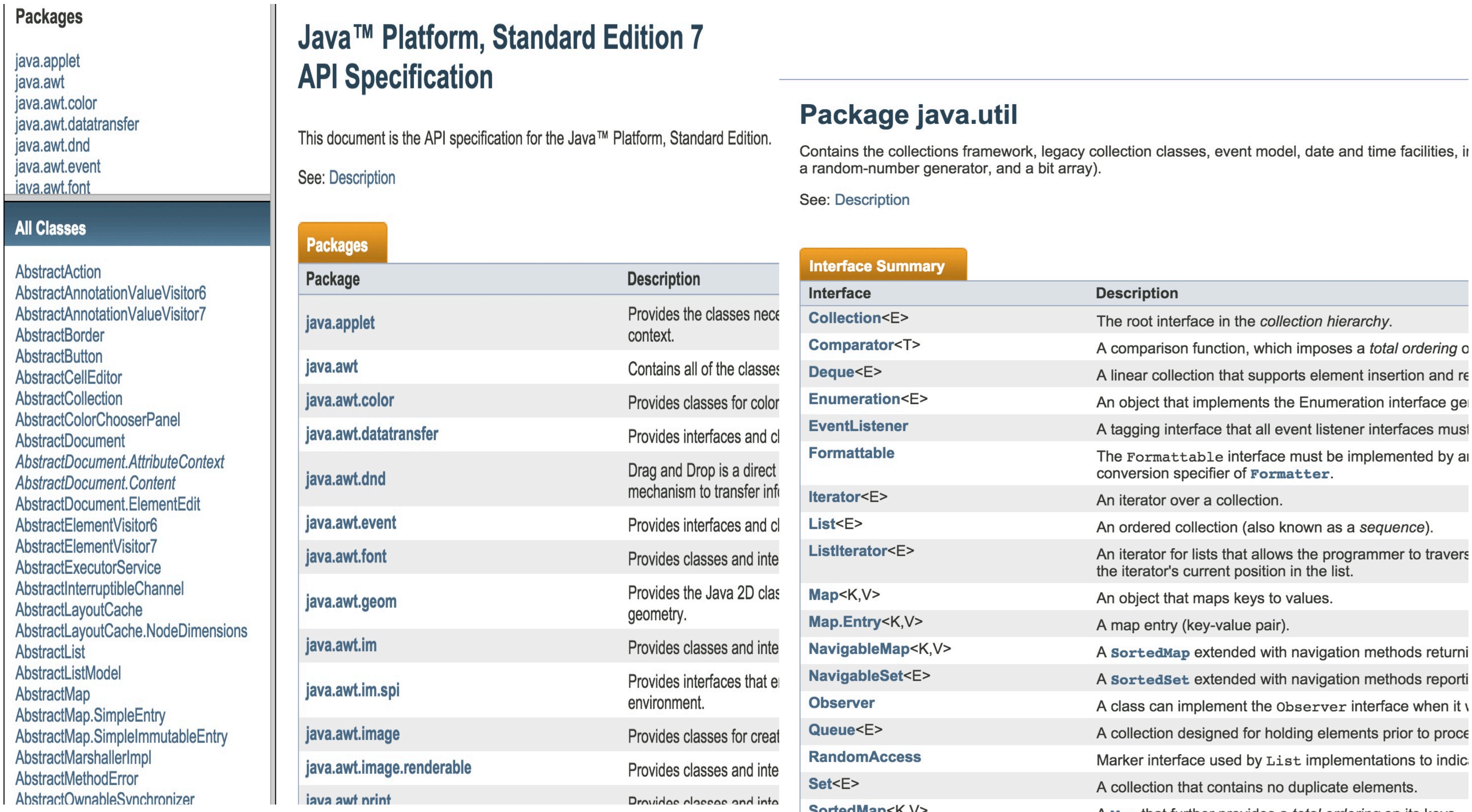
- Use `new BigDecimal(String)`,
  not `new BigDecimal(double)`
- `BigDecimal.valueOf(double)` is better, but not perfect
  – Use it for non-constant values.
- For API designers
  – Make it easy to do the commonly correct thing
  – Make it hard to misuse
  – Make it possible to do exotic things

Learning goals for today

- 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
  - "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 nece context.
java.awt	Contains all of the classes
java.awt.color	Provides classes for color
java.awt.datatransfer	Provides interfaces and cl
java.awt.dnd	Drag and Drop is a direct mechanism to transfer info
java.awt.event	Provides interfaces and cl
java.awt.font	Provides classes and inte
java.awt.geom	Provides the Java 2D clas geometry.
java.awt.im	Provides classes and inte
java.awt.im.spi	Provides interfaces that e environment.
java.awt.image	Provides classes for creat
java.awt.image.renderable	Provides classes and inte
java.awt.print	Provides classes and inte

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).

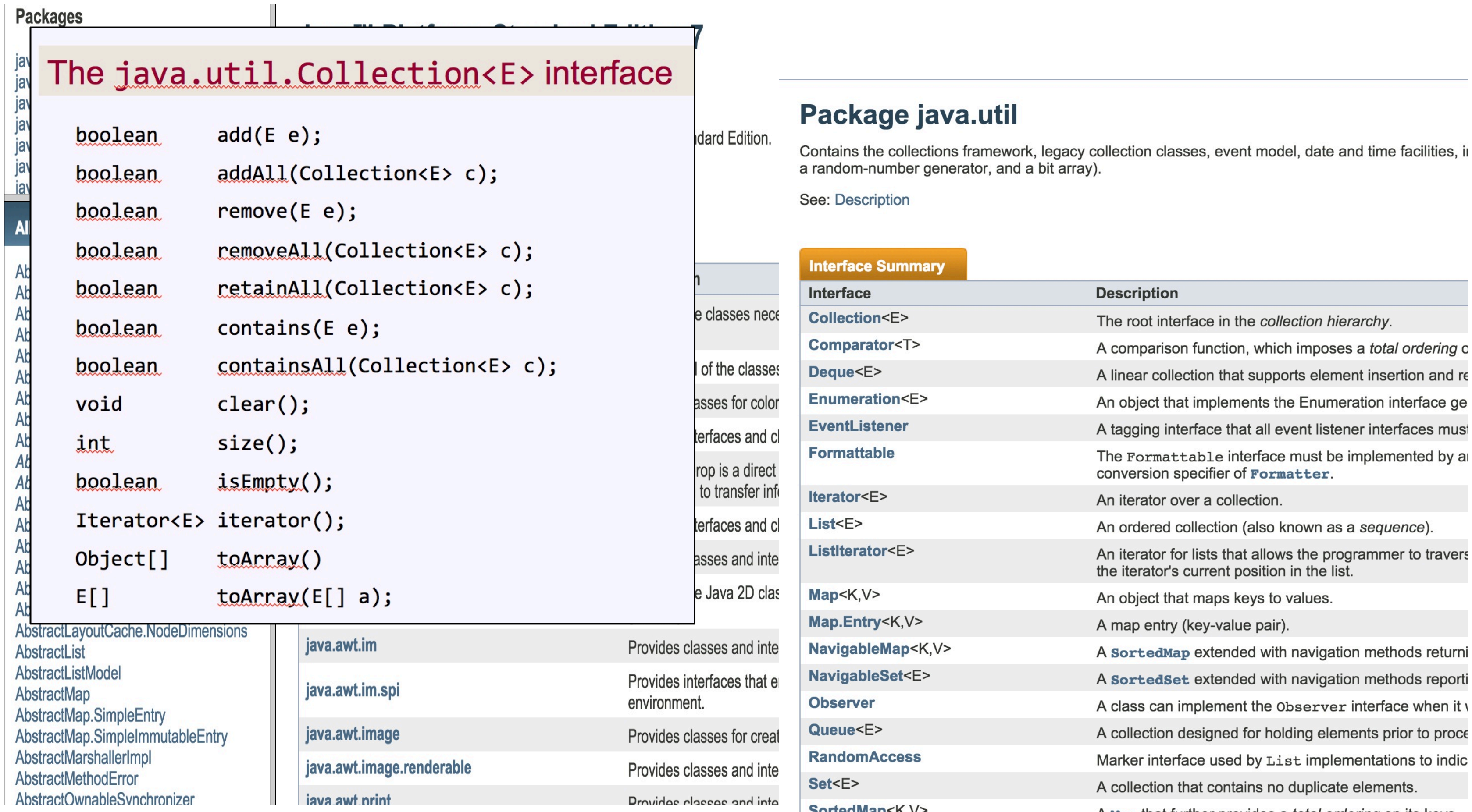
See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* o
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 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

AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer

java.awt.im	Provides classes and inte
java.awt.im.spi	Provides interfaces that e environment.
java.awt.image	Provides classes for creat
java.awt.image.renderable	Provides classes and inte

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).

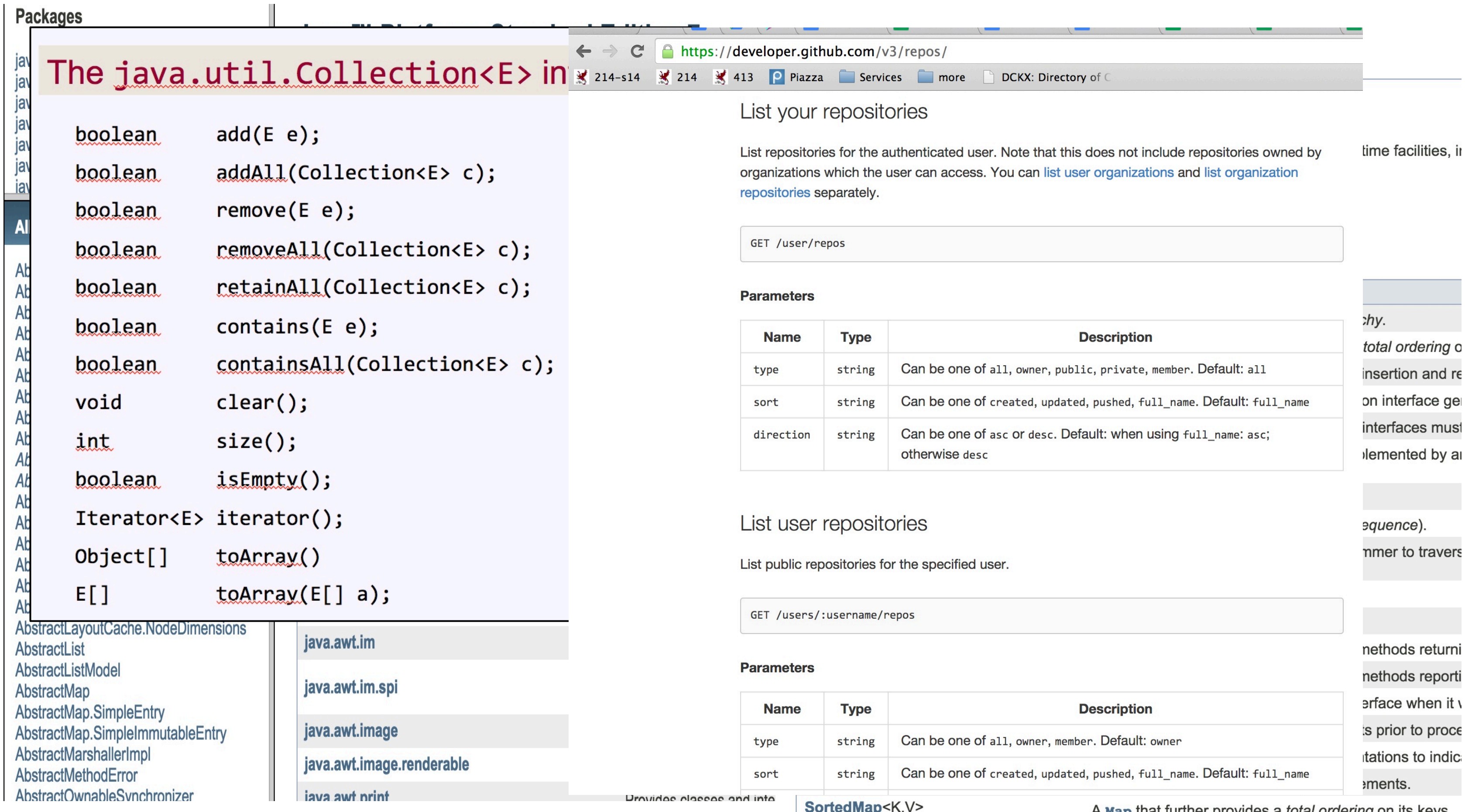
See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* o
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.

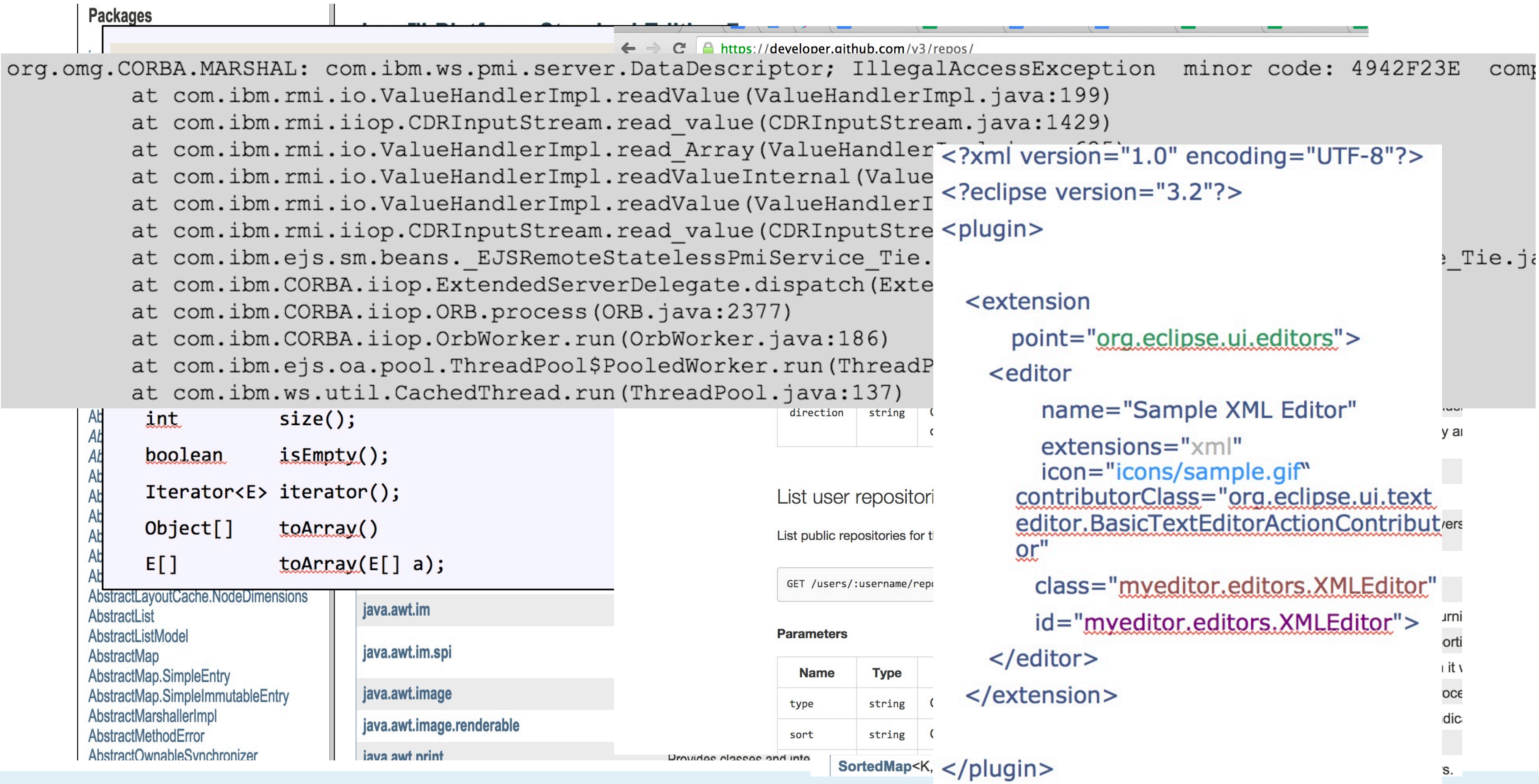
API: Application Programming Interface

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



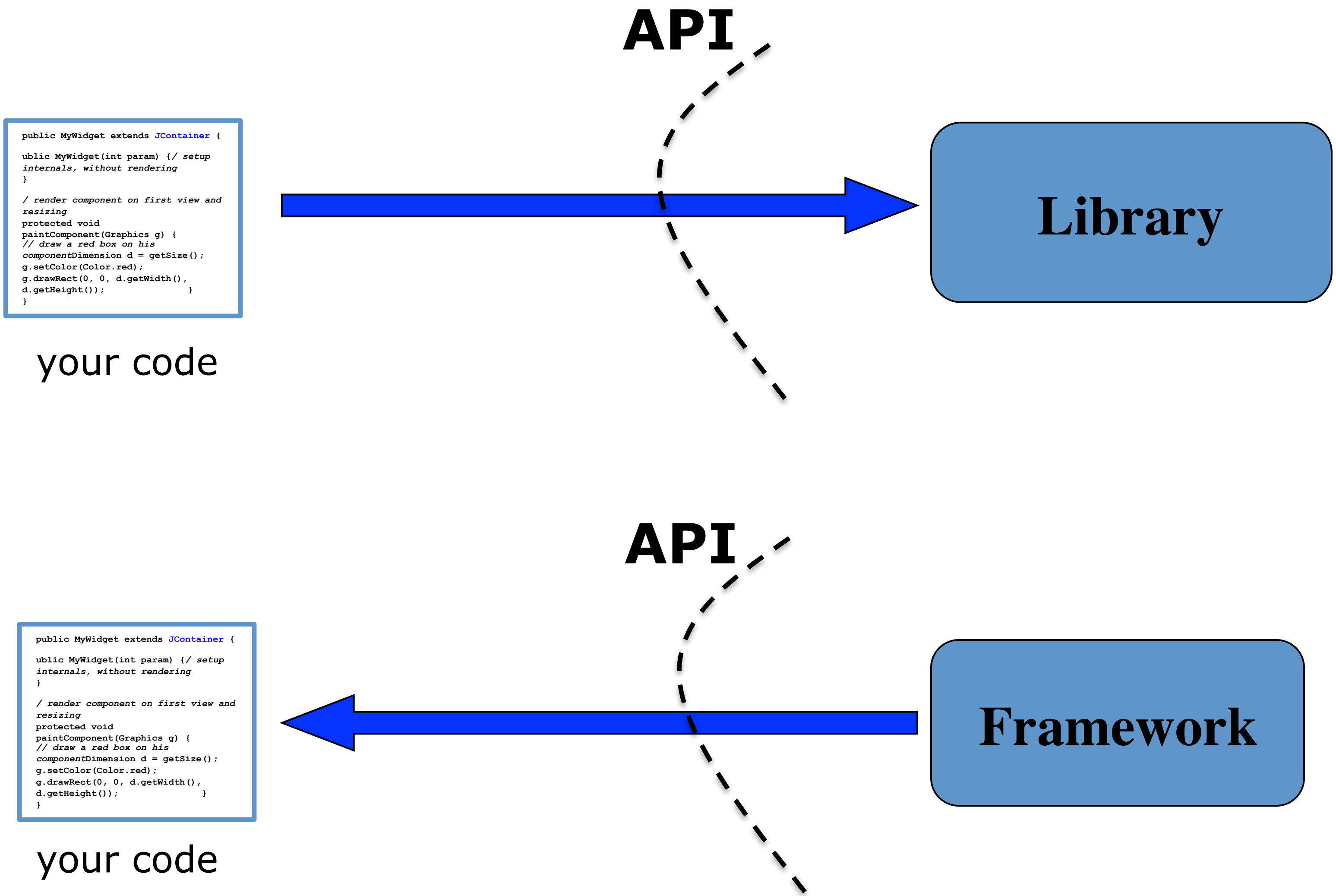
API: Application Programming Interface

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



Institute for SOFTWARE RESEARCH

Libraries and frameworks both define APIs



Motivation to create a public API

- Good APIs are a great asset
  - Distributed development among many teams
    - Incremental, non-linear software development
    - Facilitates communication
  - Long-term buy-in from clients & customers
- Poor APIs are a great liability
  - Lost productivity from your software developers
  - Wasted customer support resources
  - Lack of buy-in from clients & customers

Evolutionary problems: Public APIs are forever

- "One chance to get it right"
- You can add features, but never remove or change the behavioral contract for an existing feature
  - You can neither add nor remove features from an interface*

Motivation to create a public API

- Good APIs are a great asset
  - Distributed development among many teams
    - Incremental, non-linear software development
    - Facilitates communication
  - Long-term buy-in from clients & customers
- Poor APIs are a great liability
  - Lost productivity from your software developers
  - Wasted customer support resources
  - Lack of buy-in from clients & customers

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
  - Keep an issues list
- 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();
    ...  // Remainder omitted
}
```


institute for SOFTWARE RESEARCH

Review from a ***very*** senior engineer

```
API                  vote    notes
=====================================================================
Array                yes      But remove binarySearch* and toList
BasicCollection      no       I don't expect lots of collection classes
BasicList            no       see List below
Collection           yes      But cut toArray
Comparator           no
DoublyLinkedList     no       (without generics this isn't worth it)
HashSet              no
LinkedList           no       (without generics this isn't worth it)
List                 no       I'd like to say yes, but it's just way
                              bigger than I was expecting
RemovalEnumeration   no
Table                yes      BUT IT NEEDS A DIFFERENT NAME
TreeSet              no

I'm generally not keen on the toArray methods because they add complexity

Similarly, I don't think that the table Entry subclass or the various
views mechanisms carry their weight.
```

An aside: Should `List<T>` contain a `.sort` method?

An aside: Should `List<T>` contain a `.sort` method?

- Before Java 1.8, had to use `Collections.sort`:

```
public static <T extends Comparable<? super T>> void sort(List<T> list) {
        Object[] a = list.toArray();
        Arrays.sort(a);
        ListIterator<T> i = list.listIterator();
        for (int j=0; j<a.length; j++) {
            i.next();
            i.set((T)a[j]);
        }
    }
```

Java 1.8 introduced *default* interface methods

- `List<T>.sort`:

```
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```

- `Collections.sort`:

```
public static <T extends Comparable<? super T>> void sort(List<T> list) {
    list.sort(null);
}
```

Respect the rule of three

- Via Will Tracz (via Josh Bloch), *Confessions of a Used Program Salesman*:
  - "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."

Documenting an API

- 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

Key design principle: Information hiding

- "When in doubt, leave it out."

Documenting an API

- 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
- Do not document implementation details

Key design principle: Information hiding (2)

- Minimize the accessibility of classes, fields, and methods
  - You can add features, but never remove or change the behavioral contract for an existing feature

Key design principle: Information hiding (3)

- Use accessor methods, not public fields
  - Consider:

```
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 (4)

- Prefer interfaces over abstract classes
  - Interfaces provide greater flexibility, avoid needless implementation details
  - Consider:

```
public interface Point {
   public double getX();
   public double getY();
}

public class PolarPoint() implements Point {
   private double r;       // Distance from origin.
   private double theta;  // Angle.
   public double getX() { return r*Math.cos(theta); }
   public double getY() { return r*Math.sin(theta); }
}
```

API design to be continued Thursday

Principles of Software Construction:
Objects, Design, and Concurrency

Part 4: Design for large-scale reuse

API design, continued

Charlie Garrod Michael Hilton

School of
Computer Science
institute for
SOFTWARE
RESEARCH

Administrivia

- Homework 4c due tonight
- Homework 5 coming soon
  – Team sign-up deadline Tuesday, 31 October
- Optional reading for today
  – Effective Java, Items 41 and 42
- Midterm exam in class next Thursday (02 November)
  – Review session Wednesday, 01 Nov. 7-9 p.m. in HH B103
- 15-214 --> 17-214
  – (Also 17-514)

Key concepts from Tuesday

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
  - Keep an issues list
- Code early, code often
  - Write *client code* before you implement the API

Respect the rule of three

- Via Will Tracz (via Josh Bloch), *Confessions of a Used Program Salesman*:
  - "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."

Key design principle: Information hiding

- "When in doubt, leave it out."

Today: API design, continued

- An API design process
- The key design principle: information hiding
- Concrete advice for user-centered design

Based heavily on "How to Design a Good API and Why it Matters" by Josh Bloch.

If you have "Java" in your résumé you should own *Effective Java*.

Key design principle: Information hiding (5)

- 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

Key design principle: Information hiding (6)

- Prevent subtle leaks of implementation details
  - Documentation
  - Lack of documentation
  - Implementation-specific return types
  - Implementation-specific exceptions
  - Output formats
  - `implements Serializable`

Minimize conceptual weight

- Conceptual weight: How many concepts must a programmer learn to use your API?
  - APIs should have a "high power-to-weight ratio"
- See `java.util.*`, `java.util.Collections`

`static <T> Collection<T>`	`synchronizedCollection(Collection<T> c)` Returns a synchronized (thread-safe) collection backed by the specified collection.
`static <T> List<T>`	`synchronizedList(List<T> list)` Returns a synchronized (thread-safe) list backed by the specified list.
`static <K,V> Map<K,V>`	`synchronizedMap(Map<K,V> m)` Returns a synchronized (thread-safe) map backed by the specified map.
`static <T> Set<T>`	`synchronizedSet(Set<T> s)` Returns a synchronized (thread-safe) set backed by the specified set.
`static <K,V> SortedMap<K,V>`	`synchronizedSortedMap(SortedMap<K,V> m)` Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
`static <T> SortedSet<T>`	`synchronizedSortedSet(SortedSet<T> s)` Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
`static <T> Collection<T>`	`unmodifiableCollection(Collection<? extends T> c)` Returns an unmodifiable view of the specified collection.
`static <T> List<T>`	`unmodifiableList(List<? extends T> list)` Returns an unmodifiable view of the specified list.
`static <K,V> Map<K,V>`	`unmodifiableMap(Map<? extends K,? extends V> m)` Returns an unmodifiable view of the specified map.
`static <T> Set<T>`	`unmodifiableSet(Set<? extends T> s)` Returns an unmodifiable view of the specified set.
`static <K,V> SortedMap<K,V>`	`unmodifiableSortedMap(SortedMap<K,? extends V> m)` Returns an unmodifiable view of the specified sorted map.
`static <T> SortedSet<T>`	`unmodifiableSortedSet(SortedSet<T> s)` Returns an unmodifiable view of the specified sorted set.

Apply principles of user-centered design

- Other programmers are your users
- e.g., "Principles of Universal Design"
  - Equitable use
  - Flexibility in use
  - Simple and intuitive use
  - Perceptible information
  - Tolerance for error
  - Low physical effort
  - Size and space for approach and use

Good names drive good design

- Do what you say you do:
  - "Don't violate the Principle of Least Astonishment"

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

Discuss these names:

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

Good names drive good design (2)

- Follow language- and platform-dependent conventions
  - Typographical:
    - `get_x()` vs. `getX()` vs. `x()`
    - `timer` vs. `Timer`, `HTTPServlet` vs `HttpServlet`
    - `edu.cmu.cs.cs214`
  - Grammatical (next slide):

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
- 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<String,String> data;
    public String put(String key, String value) {
        return data.put(key, value);
    }
    …
}
```

Minimize mutability

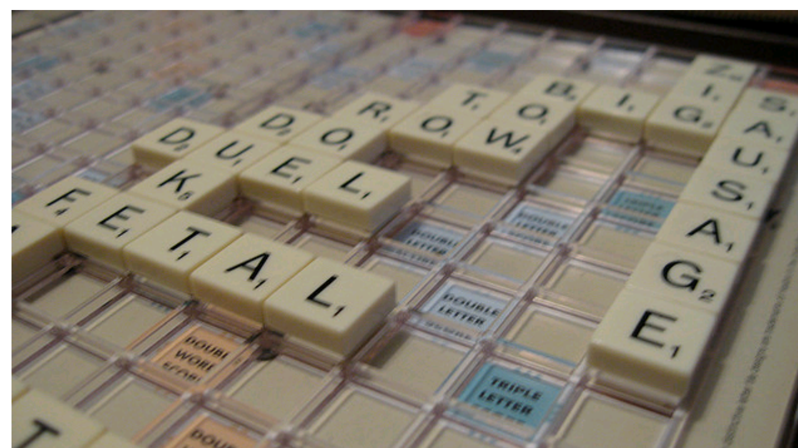
- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
    - See java.lang.String
  - Disadvantage: separate object for each value

Minimize mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
    - See java.lang.String
  - Disadvantage: separate object for each value
- Mutable objects require careful management of visibility and side effects
  - e.g. `Component.getSize()` returns a mutable `Dimension`
- Document mutability
  - Carefully describe state space

On a piece of paper (in groups of 2-3)

1. Write your Andrew IDs.
2. Argue that a Scrabble with Stuff board implementation should be *mutable*. Explicitly include design goals and design principles in your rationale, where possible.
3. Argue that a Scrabble with Stuff board implementation should be *immutable*. Explicitly include …



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 x == p.x && 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 and return types

- Favor interface types over classes for input
- Use most specific type for input type
- Do not return a `String` if a better type exists
- Do not use floating point for monetary values
- Use `double` (64 bits) instead of `float` (32 bits)

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 is always the collection to be modified or queried

`java.util.concurrent`: time is always specified as `long delay, TimeUnit unit`

Avoid long lists of parameters

- Especially avoid parameter lists 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);
```

- Instead:
  - Break up the method, or
  - Use a helper class to hold parameters, or
  - Use the *builder* design pattern

The *Effective Java*-style builder pattern


```java
NutritionFacts twoLiterDietCoke = new NutritionFacts.Builder(
    "Diet Coke", 240, 6).sodium(1).build();

public class NutritionFacts {
    public static class Builder {
        public Builder(String name, int servingSize,
                int servingsPerContainer) { ... }
        public Builder totalFat(int val)     { totalFat = val; …}
        public Builder saturatedFat(int val) { satFat = val; …}
        public Builder transFat(int val)     { transFat = val; …}
        public Builder cholesterol(int val)  { cholesterol = val; …
        ... // 15 more setters

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder) { ... }
}
```

Summary

- Accept the fact that you, and others, will make mistakes
  - Use your API as you design it
  - Get feedback from others
  - Hide information to give yourself maximum flexibility later
  - Design for inattentive, hurried users
  - Document religiously