Principles of Software Construction: Objects, Design, and Concurrency

Part 2: Design for large-scale reuse

API design (and some libraries and frameworks…)

Charlie Garrod **Bogdan Vasilescu**

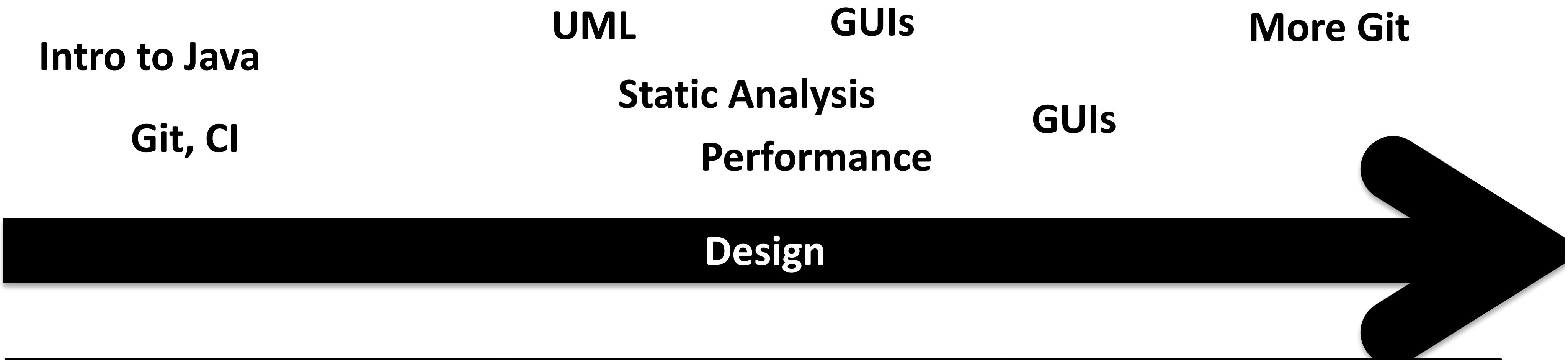
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4c due Thursday
  - Remember: up to 75% of lost points on 4a back
- Homework 5 coming soon
  - Team sign-up deadline next week
- Required reading due today
  - Effective Java: Items 51 (method names), 60 (avoid float and double), 62 (avoid strings), and 64 (prefer interfaces)
- Midterm exam in class next week Thursday (29 March)
  - Review session next week Wednesday

https://commons.wikimedia.org/wiki/File:1_carcassonne_aerial_2016.jpg



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

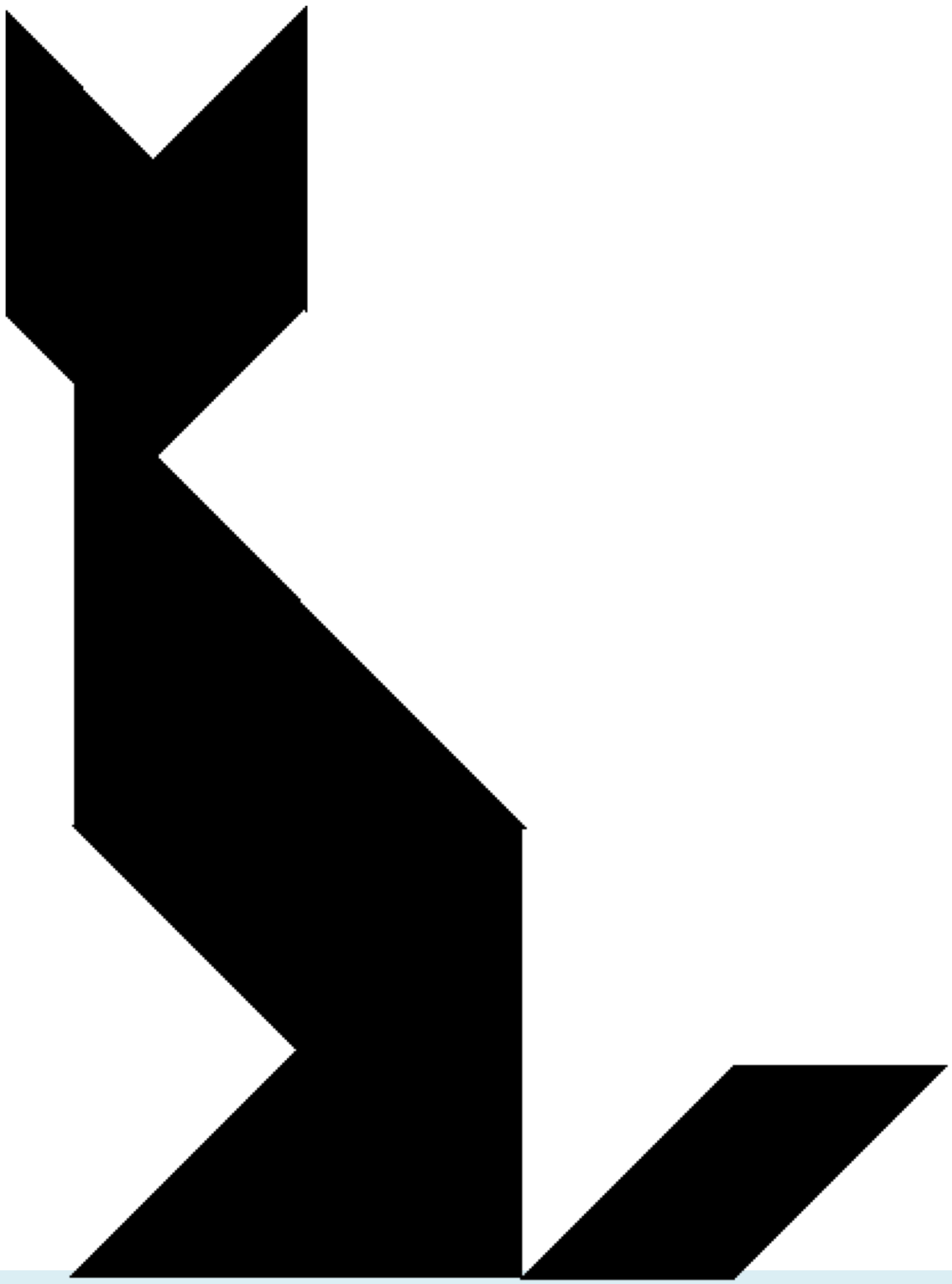
Key concepts from last Thursday

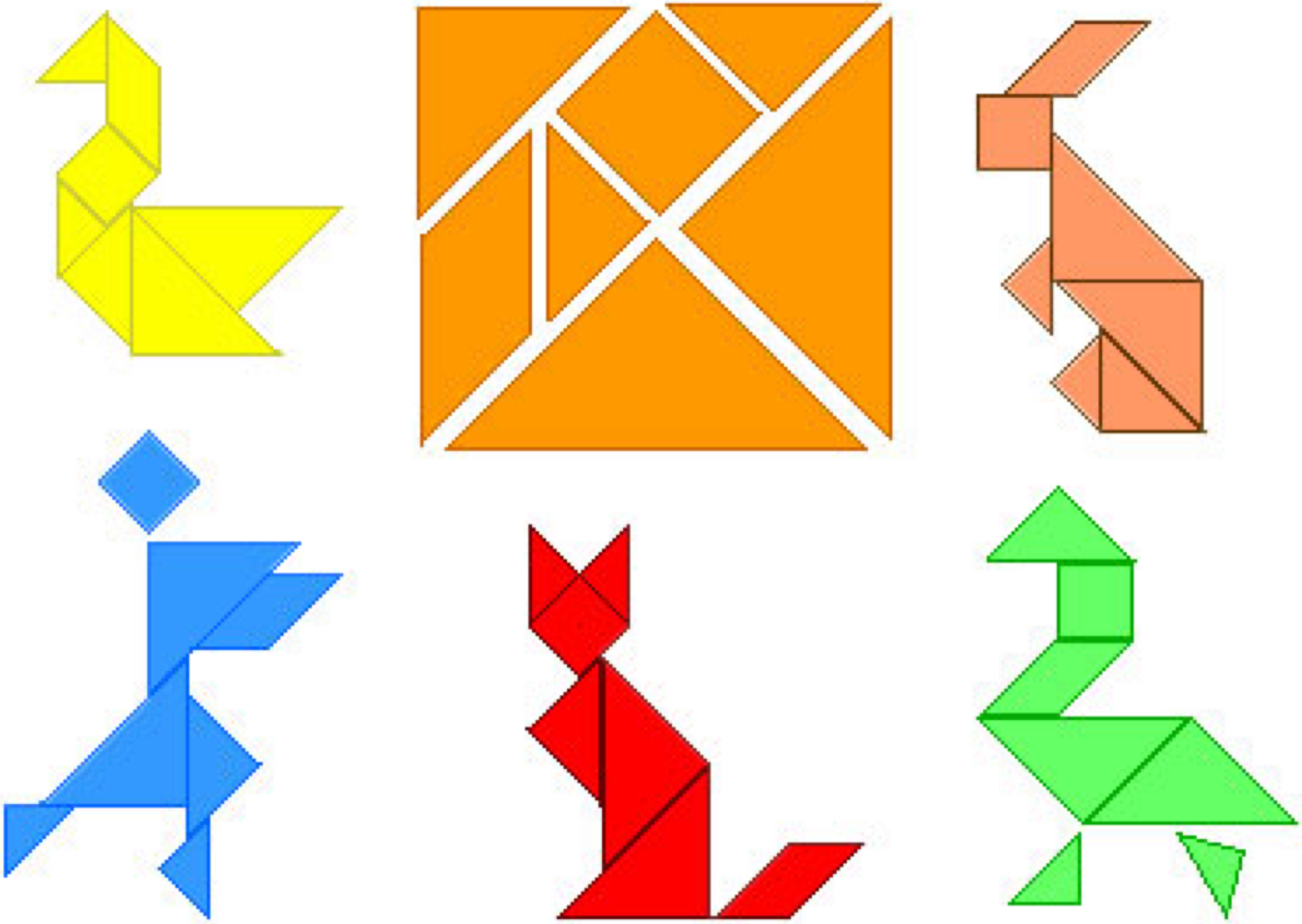
Key concepts from last Thursday

- 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
  - Too many extension points
  - Too generic





(one modularization: tangrams)

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

This week: API design

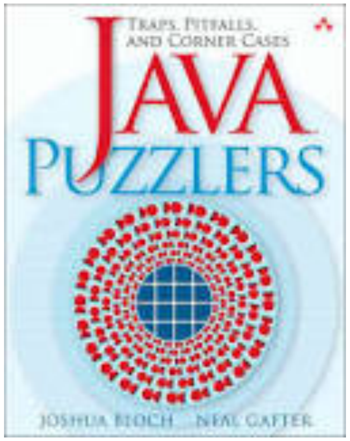
- An API design process
- The key design principle: information hiding
- Concrete advice for user-centered design

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

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);
    }
}
```



From *An Evening Of Puzzlers* by Josh Bloch

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

```java
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

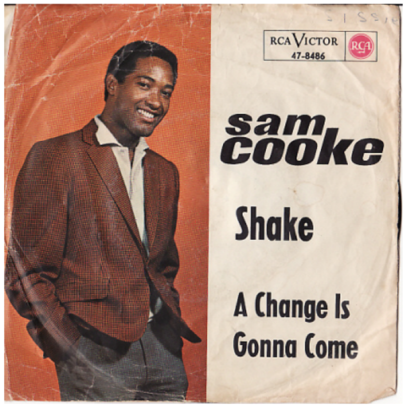
```
// ...or you could fix it this way
public class Change {
    public static void main(String args[]) {
        System.out.println(200 - 110);
    }
}
```

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:

```
0.8999999999999999111821580299874767661094665527343 75
```

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

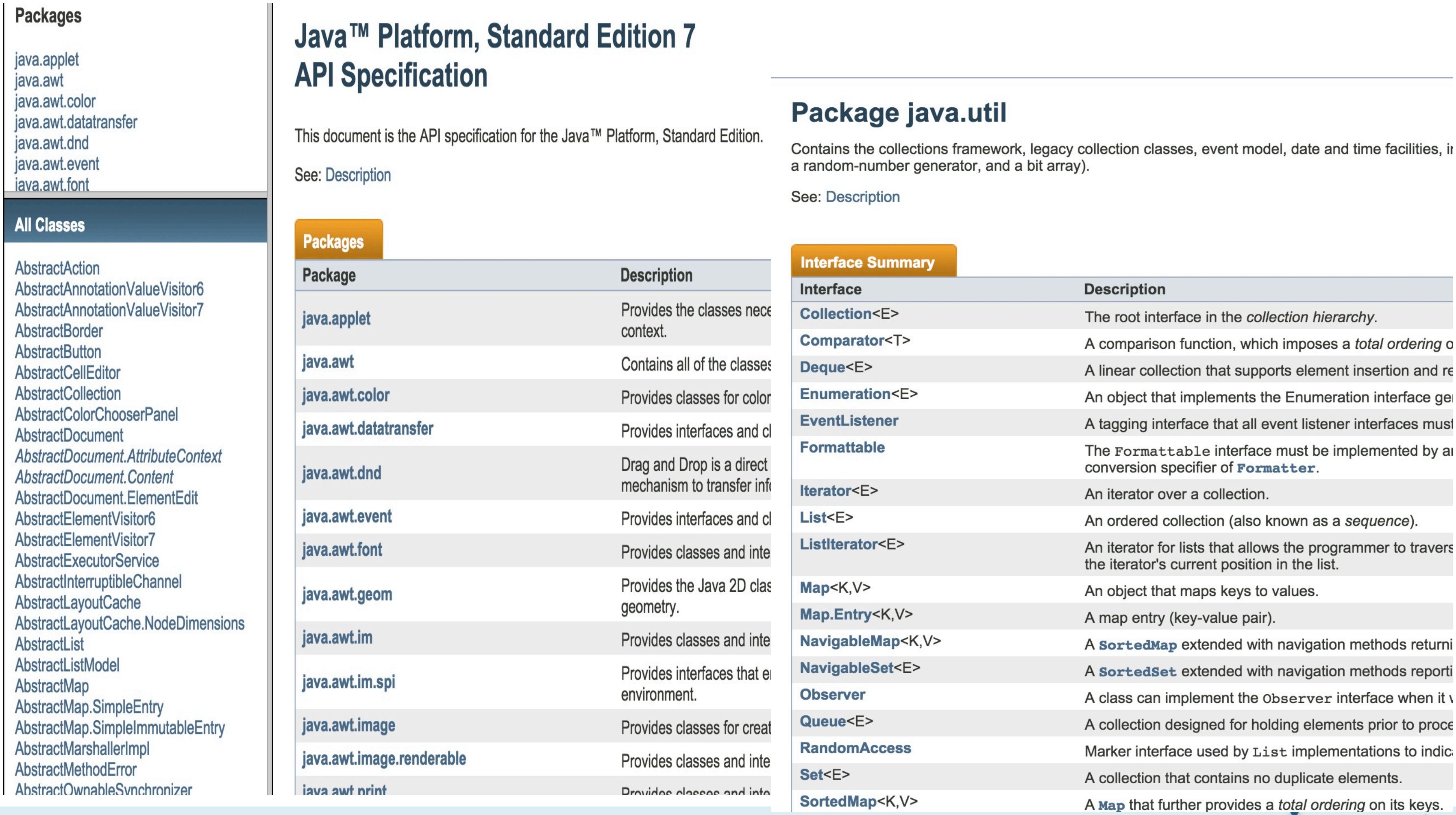
- Use `new BigDecimal(String)`,
  not `new BigDecimal(double)`
- `BigDecimal.valueOf(double)` is better, but not perfect
  - Use it for non-constant values.
- For API designers
  - Make it easy to do the commonly correct thing
  - Make it hard to misuse
  - Make it possible to do exotic things

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 of 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 inf
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).

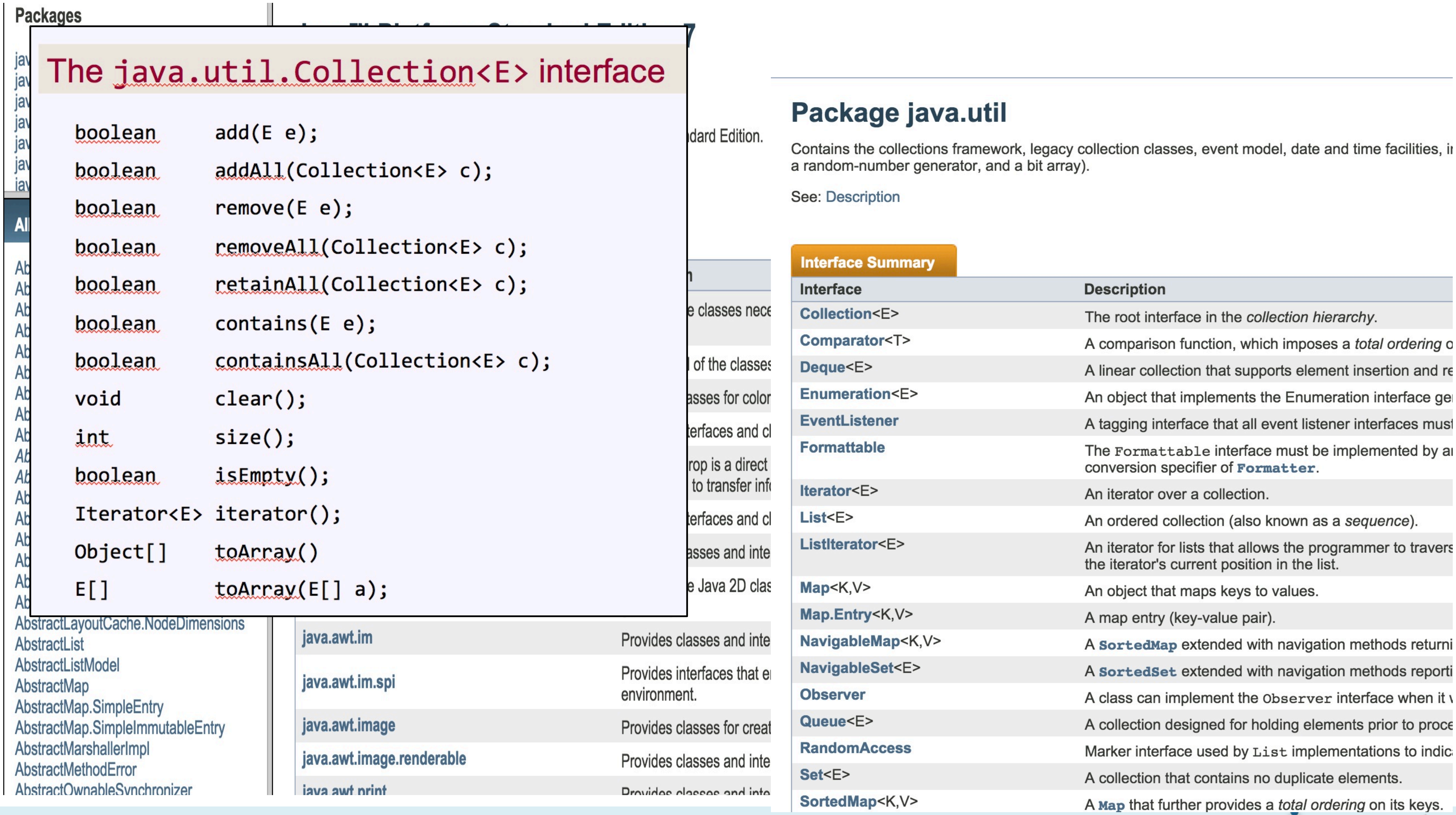
See: Description

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
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



Packages

dard Edition.

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);
```

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
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).

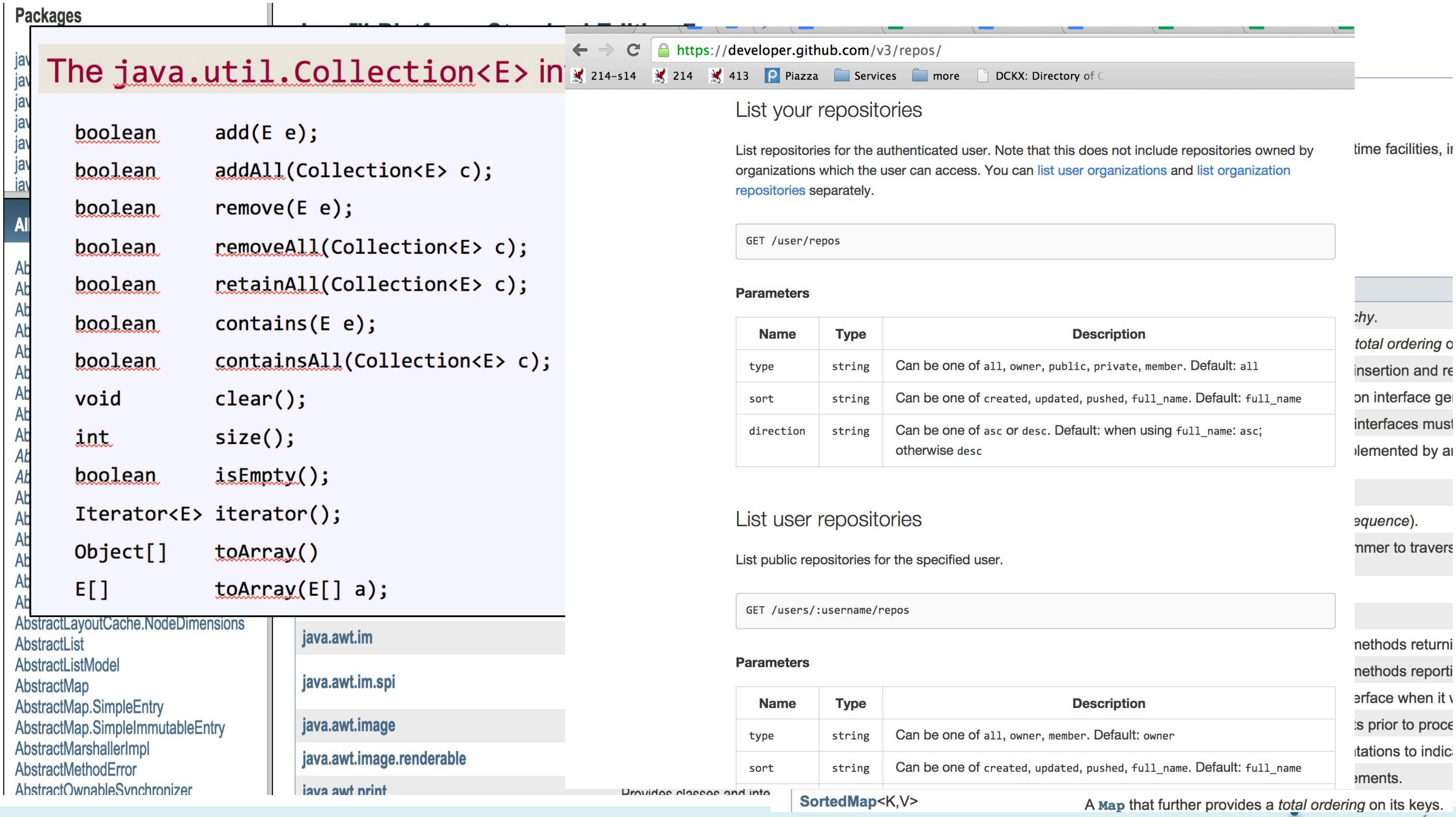
See: Description

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
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



Packages

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

java.awt.im
java.awt.im.spi
java.awt.image
java.awt.image.renderable
java.awt.print

The java.util.Collection<E> in

```
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);
```

https://developer.github.com/v3/repos/

List your repositories

List repositories for the authenticated user. Note that this does not include repositories owned by organizations which the user can access. You can list user organizations and list organization repositories separately.

```
GET /user/repos
```

Parameters

Name	Type	Description
`type`	`string`	Can be one of `all`, `owner`, `public`, `private`, `member`. Default: `all`
`sort`	`string`	Can be one of `created`, `updated`, `pushed`, `full_name`. Default: `full_name`
`direction`	`string`	Can be one of `asc` or `desc`. Default: when using `full_name`: `asc`; otherwise `desc`

List user repositories

List public repositories for the specified user.

```
GET /users/:username/repos
```

Parameters

Name	Type	Description
`type`	`string`	Can be one of `all`, `owner`, `member`. Default: `owner`
`sort`	`string`	Can be one of `created`, `updated`, `pushed`, `full_name`. Default: `full_name`

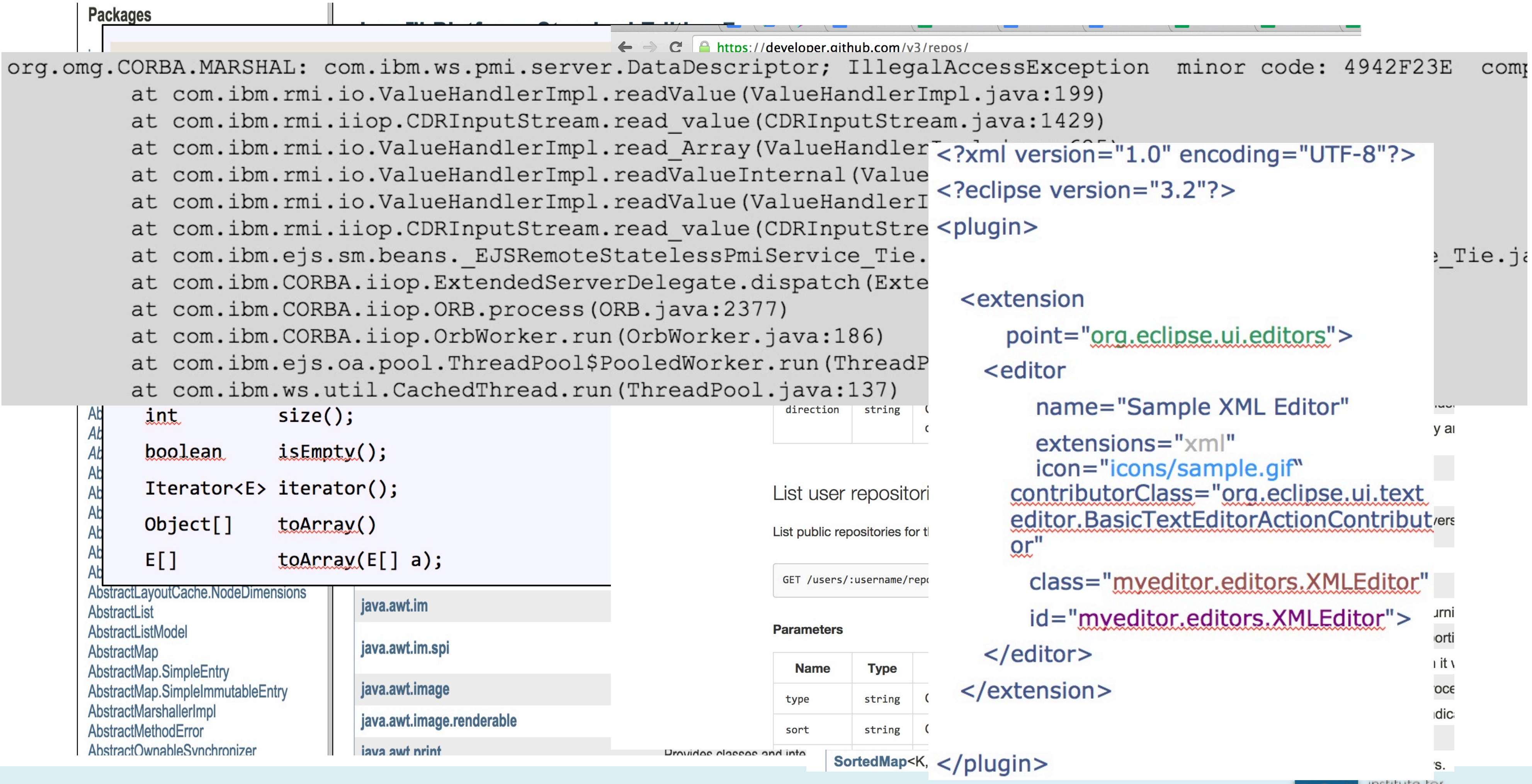
time facilities, i
chy.
total ordering o
insertion and re
on interface ge
interfaces must
lemented by a
equence).
mmer to travers
nethods returni
nethods reporti
erface when it
s prior to proce
tations to indic
ements.

Provides classes and inte

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



```
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(ValueHandler
        at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(Value
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerI
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStre
        at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie.                e_Tie.ja
        at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(Exte
        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(ThreadP
        at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

  <extension
      point="org.eclipse.ui.editors">
    <editor
        name="Sample XML Editor"
        extensions="xml"
        icon="icons/sample.gif"
     contributorClass="org.eclipse.ui.text
     editor.BasicTextEditorActionContribut
     or"
       class="myeditor.editors.XMLEditor"
       id="myeditor.editors.XMLEditor">
    </editor>
  </extension>

</plugin>
```

```
int          size();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
E[]          toArray(E[] a);
```

https://developer.github.com/v3/repos/

direction	string

List user repositori

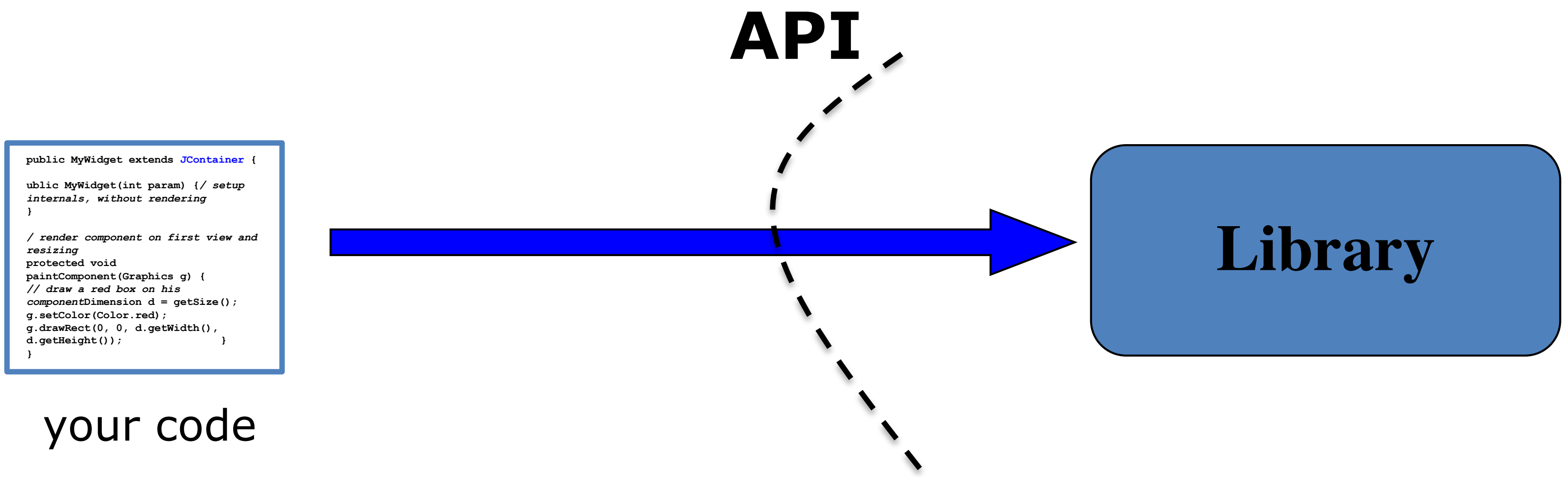
List public repositories for t

```
GET /users/:username/repo
```

Parameters

Name	Type
type	string
sort	string

Libraries and frameworks both define APIs

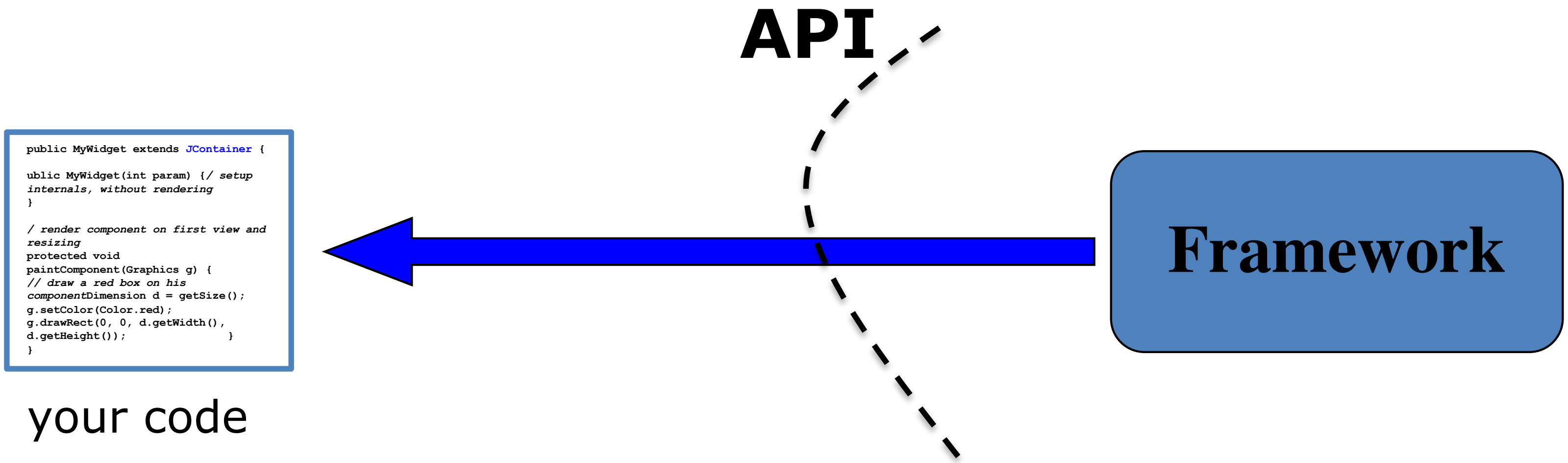


API

```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                  }
}
```

your code

Library



API

```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                  }
}
```

your code

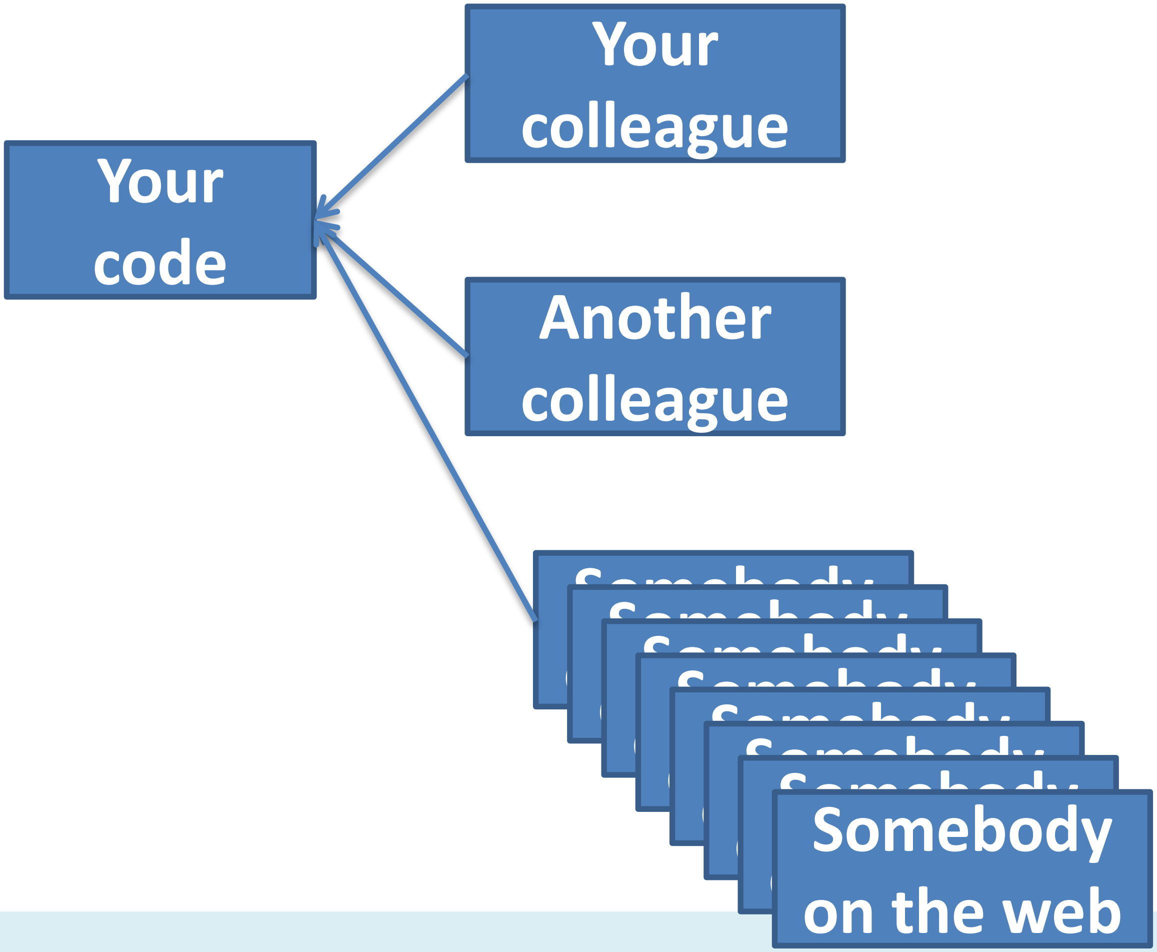
Framework

Motivation to create a public API

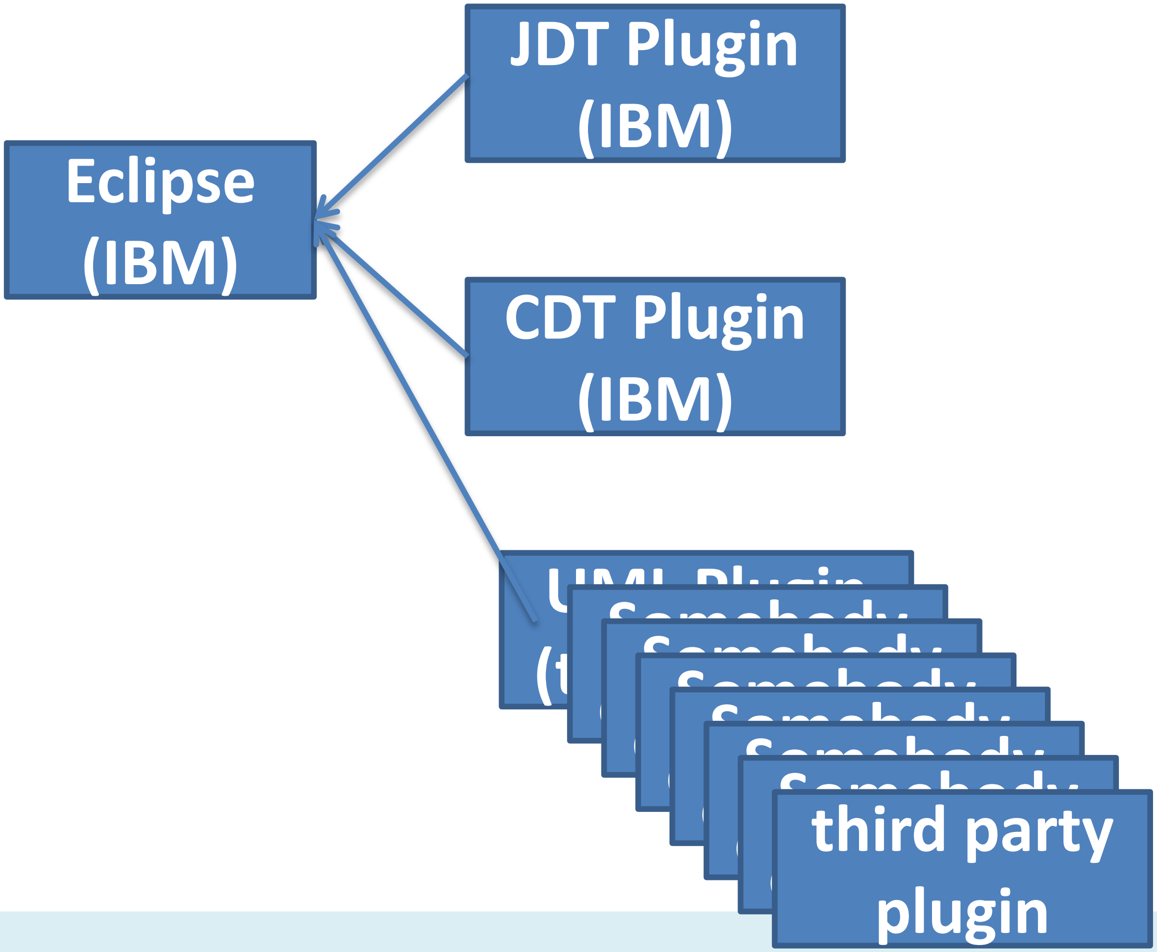
- Good APIs are a great asset
  - Distributed development among many teams
    - Incremental, non-linear software development
    - Facilitates communication
  - Long-term buy-in from clients & customers
    - Users invest heavily: acquiring, writing, learning
    - Cost to **stop** using an API can be prohibitive
    - Successful public APIs capture users

- Poor APIs are a great liability
  - Lost productivity from your software developers
  - Wasted customer support resources
  - Lack of buy-in from clients & customers

Public APIs are forever

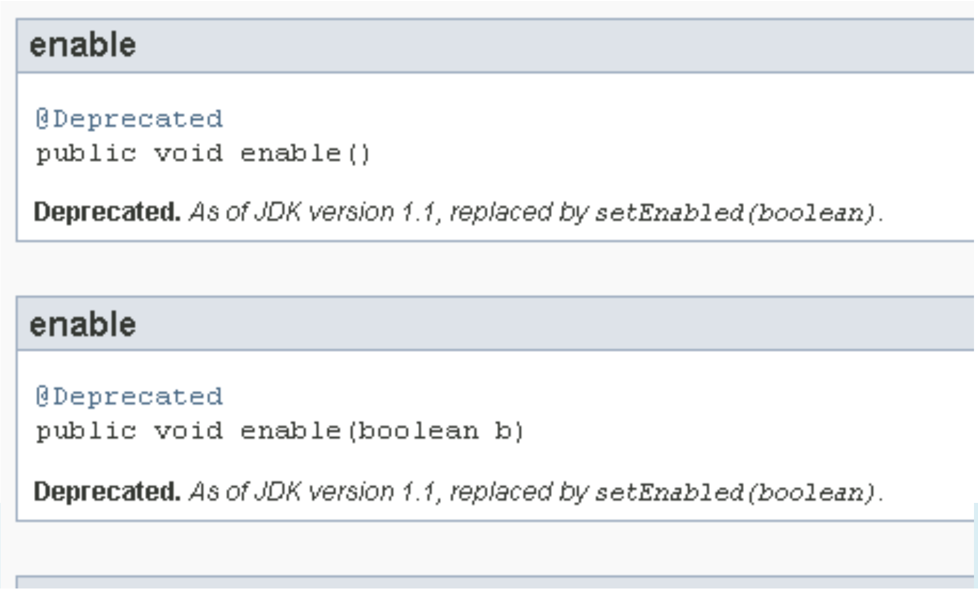


Public APIs are forever



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*
- *Deprecation of APIs as weak workaround



enable

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

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

enable

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

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

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

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."**

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?

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

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
}
```

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*: **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."

Documentation matters

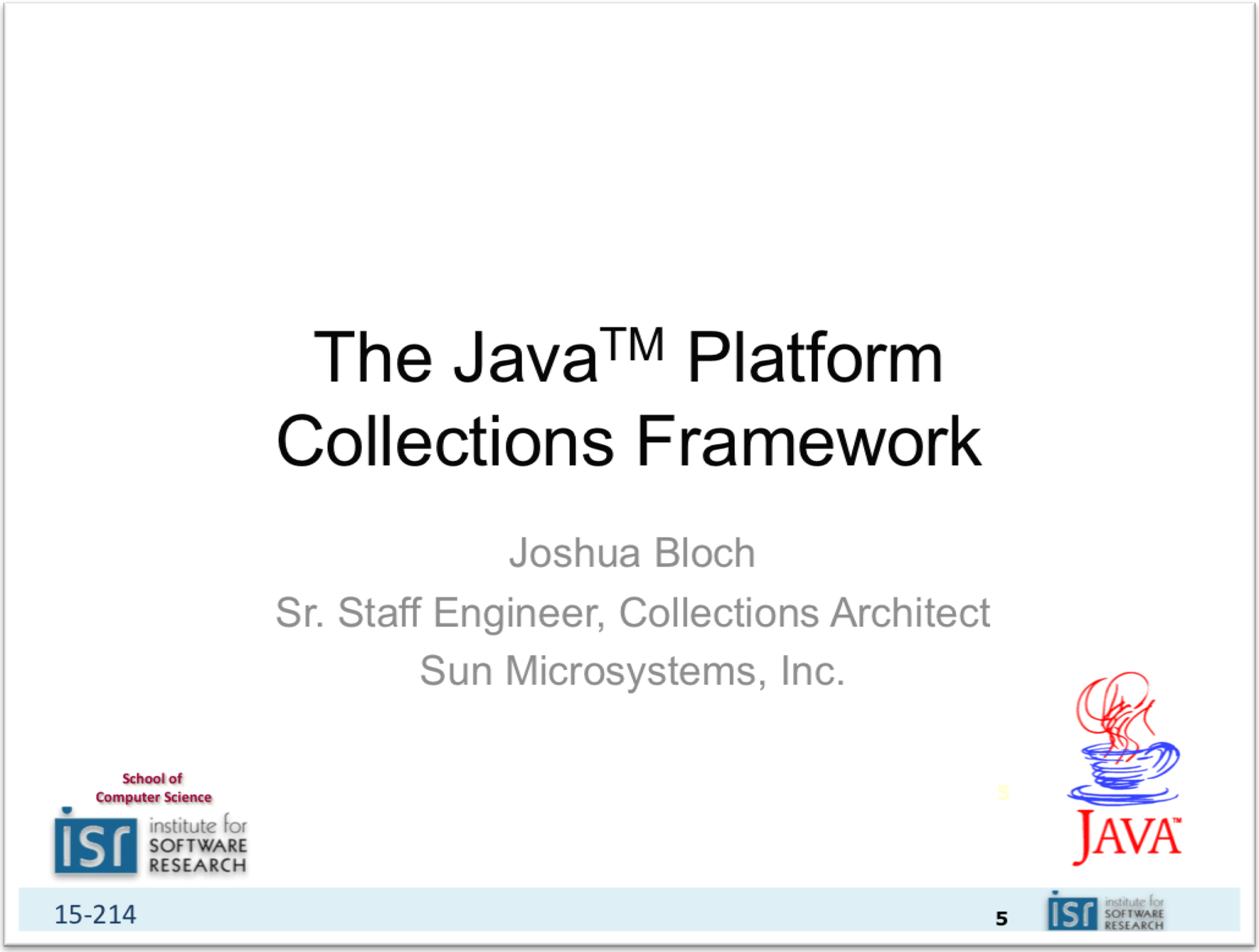
Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.

– D. L. Parnas, *Software Aging. Proceedings of the 16th International Conference on Software Engineering, 1994*

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

Josh Bloch’s experiences designing the Collections framework



The Java™ Platform Collections Framework

Joshua Bloch

Sr. Staff Engineer, Collections Architect

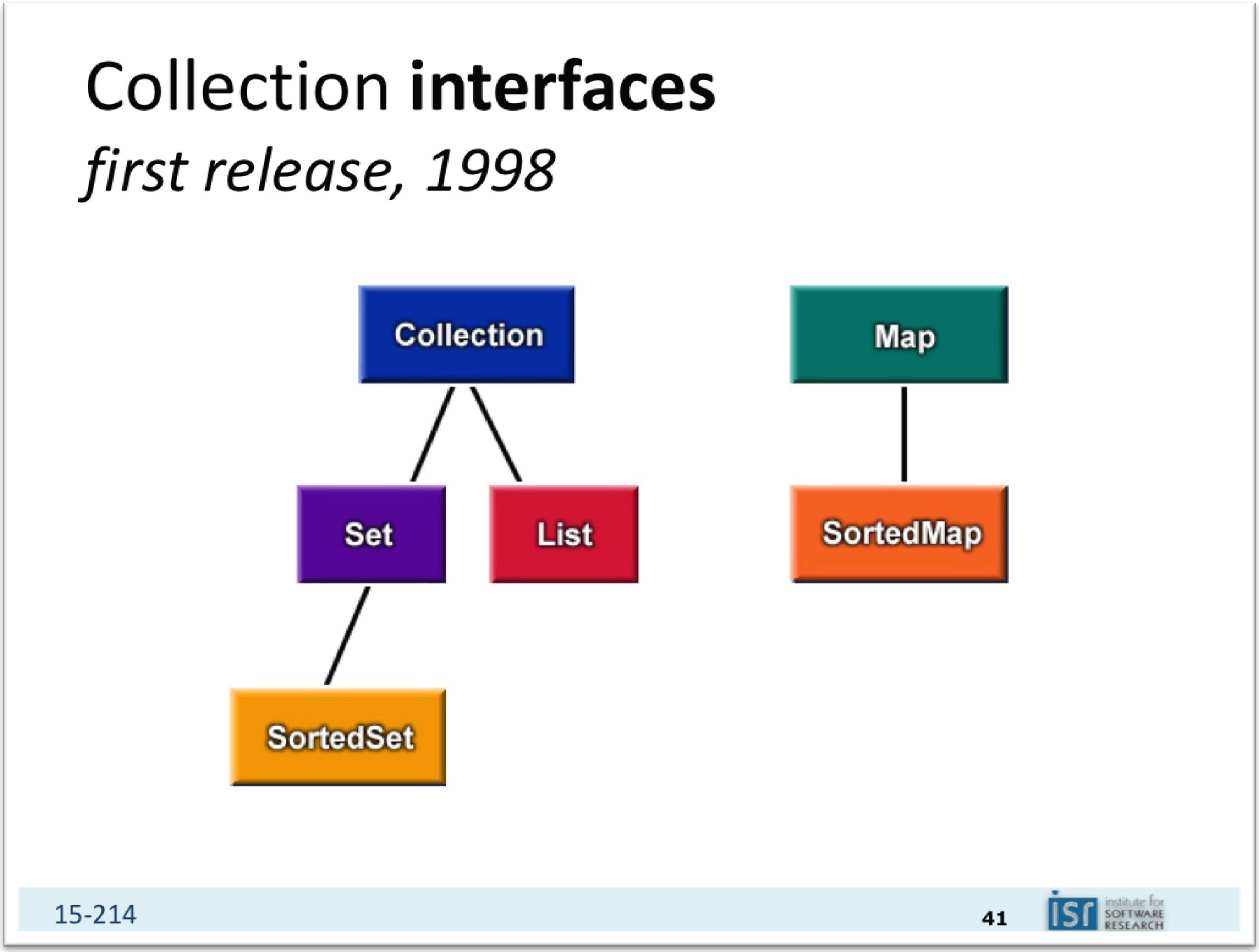
Sun Microsystems, Inc.

School of Computer Science

15-214

5

Josh Bloch’s experiences designing the Collections framework

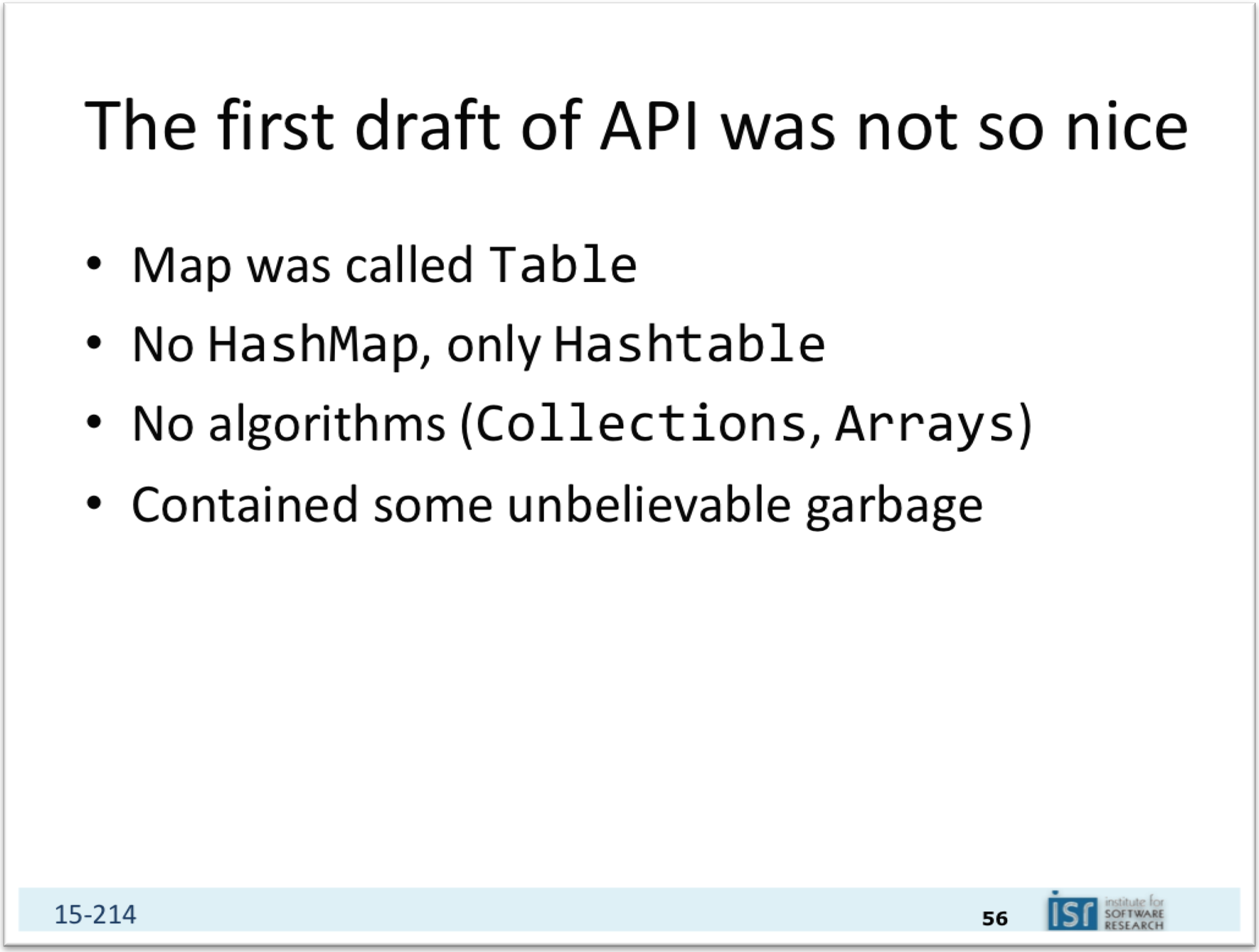


Collection **interfaces**

first release, 1998

15-214 41

Josh Bloch's experiences designing the Collections framework

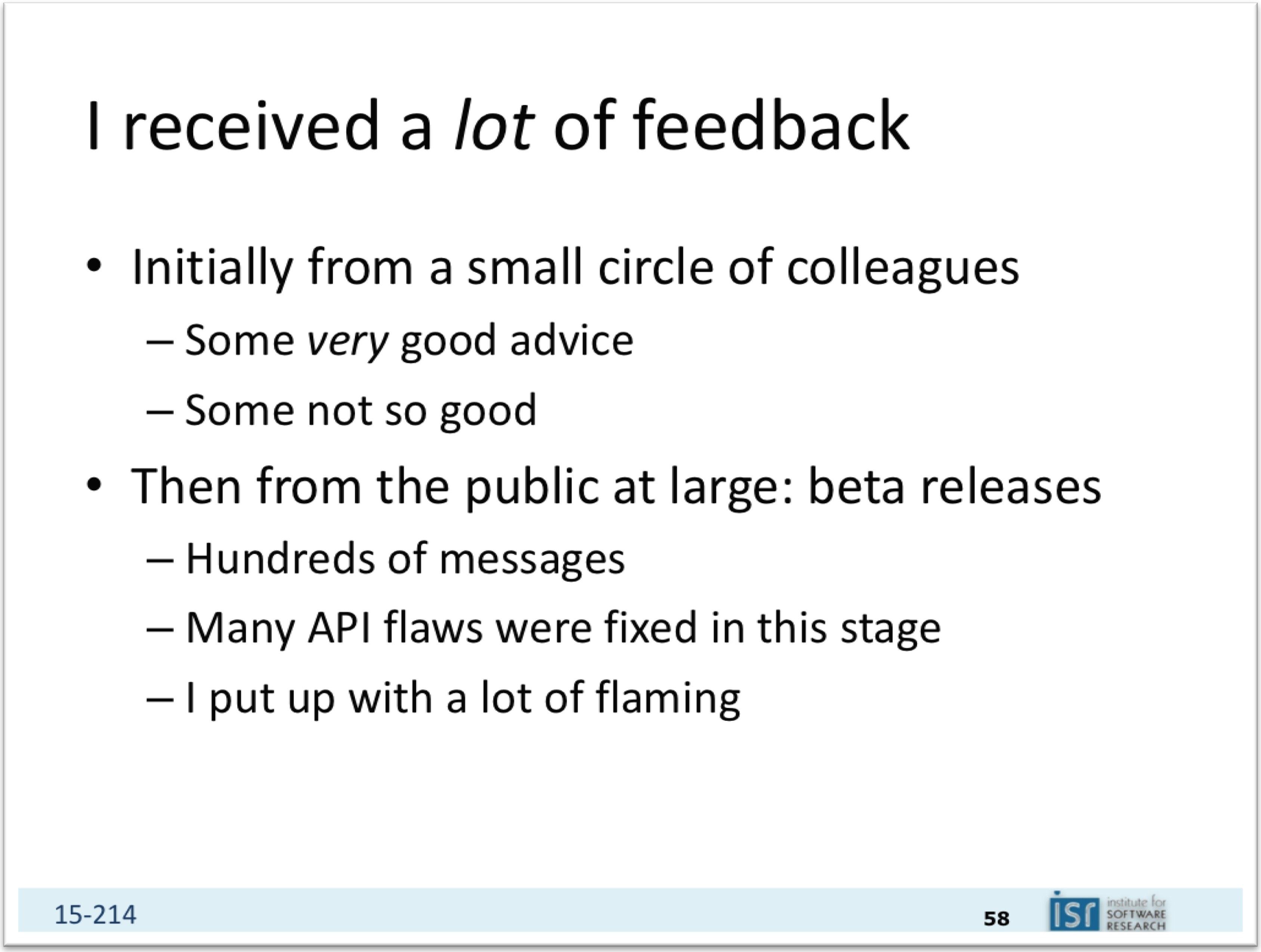


The first draft of API was not so nice

- Map was called `Table`
- No `HashMap`, only `Hashtable`
- No algorithms (`Collections`, `Arrays`)
- Contained some unbelievable garbage

15-214 56

Josh Bloch’s experiences designing the Collections framework

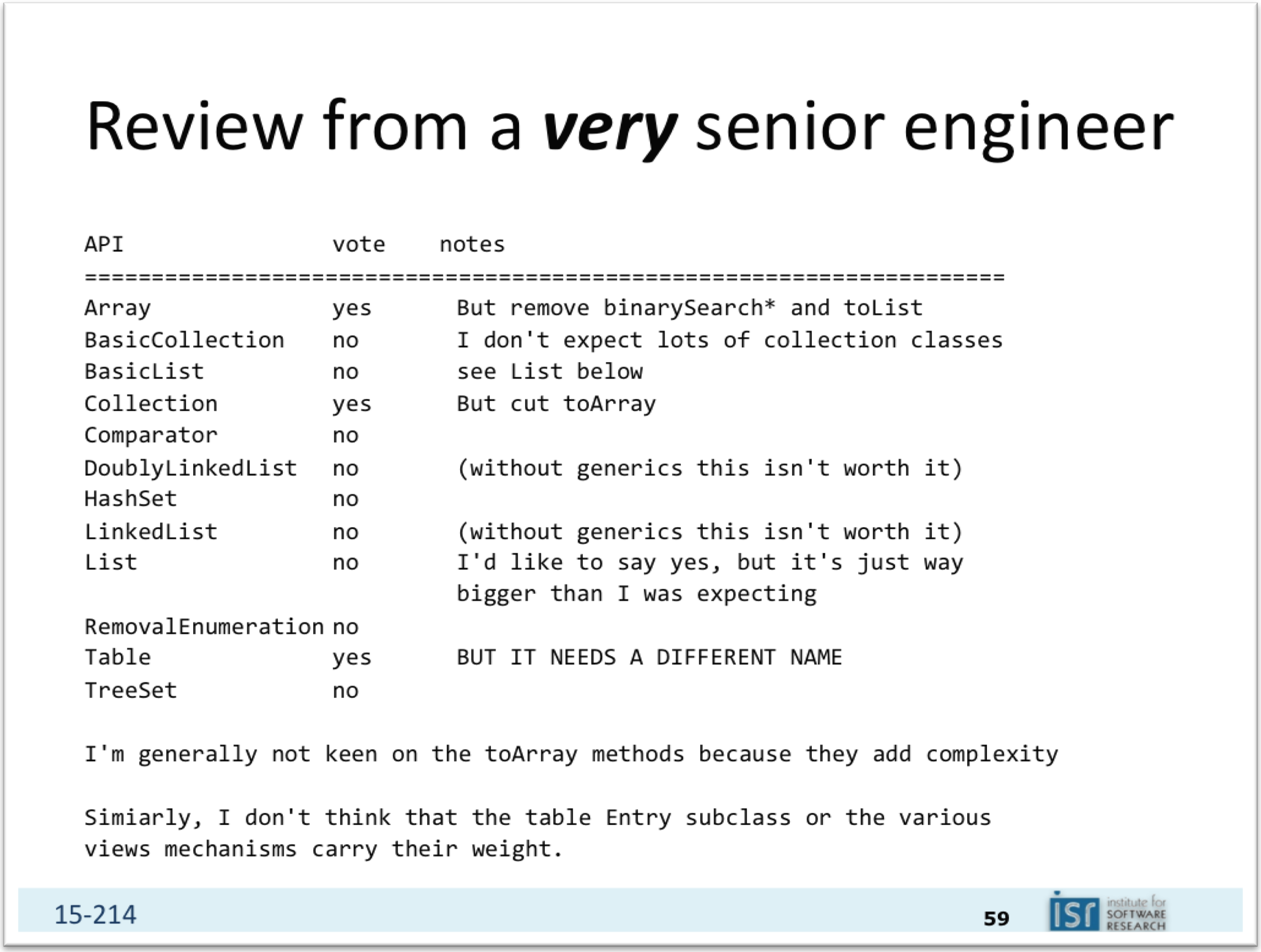


I received a *lot* of feedback

- Initially from a small circle of colleagues
  - Some *very* good advice
  - Some not so good
- Then from the public at large: beta releases
  - Hundreds of messages
  - Many API flaws were fixed in this stage
  - I put up with a lot of flaming

15-214 58

Josh Bloch’s experiences designing the Collections framework



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

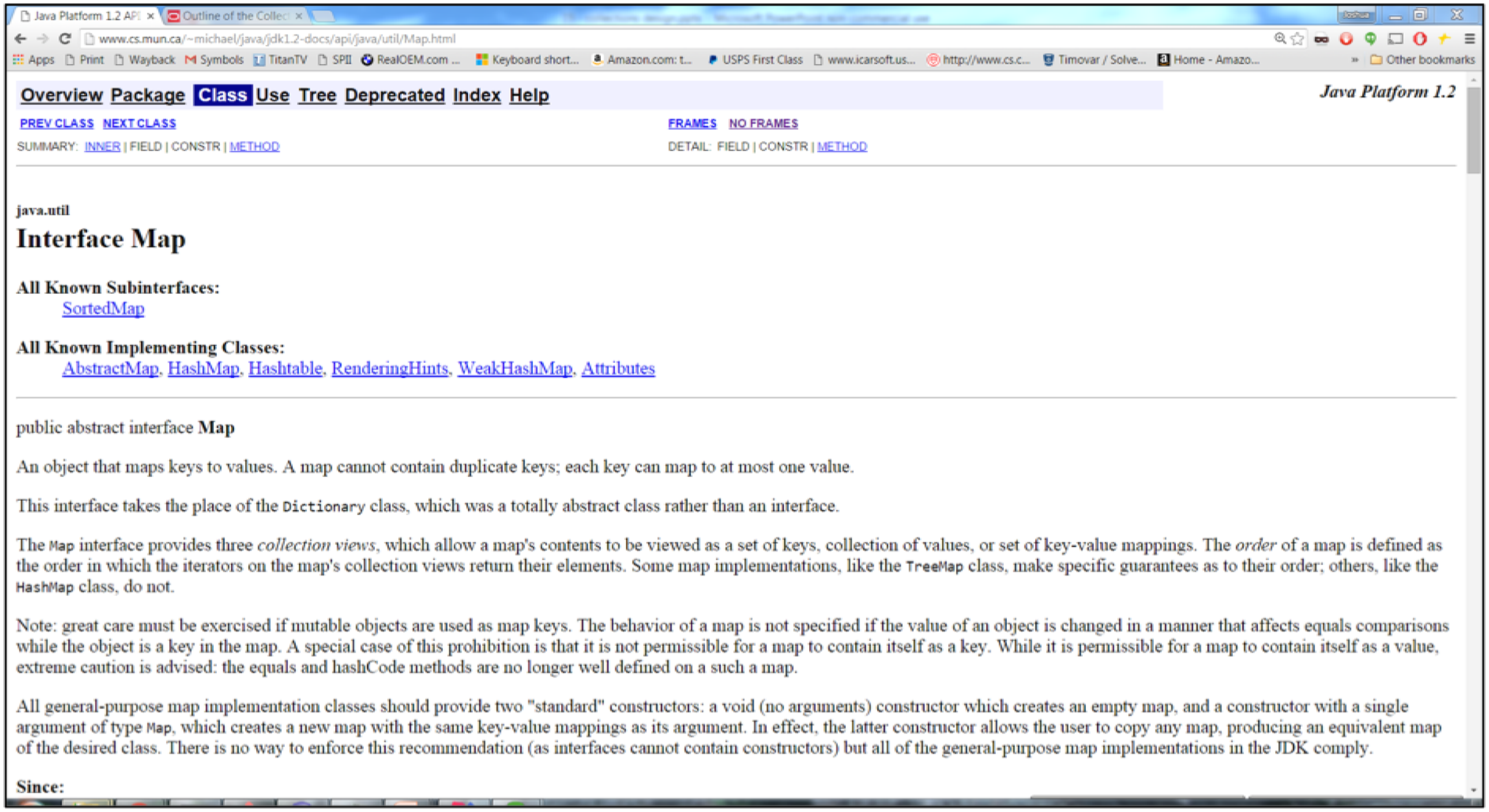
```
API                 vote    notes
=====================================================================
Array               yes      But remove binarySearch* and toList
BasicCollection     no       I don't expect lots of collection classes
BasicList           no       see List below
Collection          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

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

15-214 59

Josh Bloch's experiences designing the Collections framework

Of course you need good JavaDoc

But it is not sufficient for a substantial API



java.util

Interface Map

All Known Subinterfaces:
SortedMap

All Known Implementing Classes:
AbstractMap, HashMap, Hashtable, RenderingHints, WeakHashMap, Attributes

public abstract interface **Map**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on a such a map.

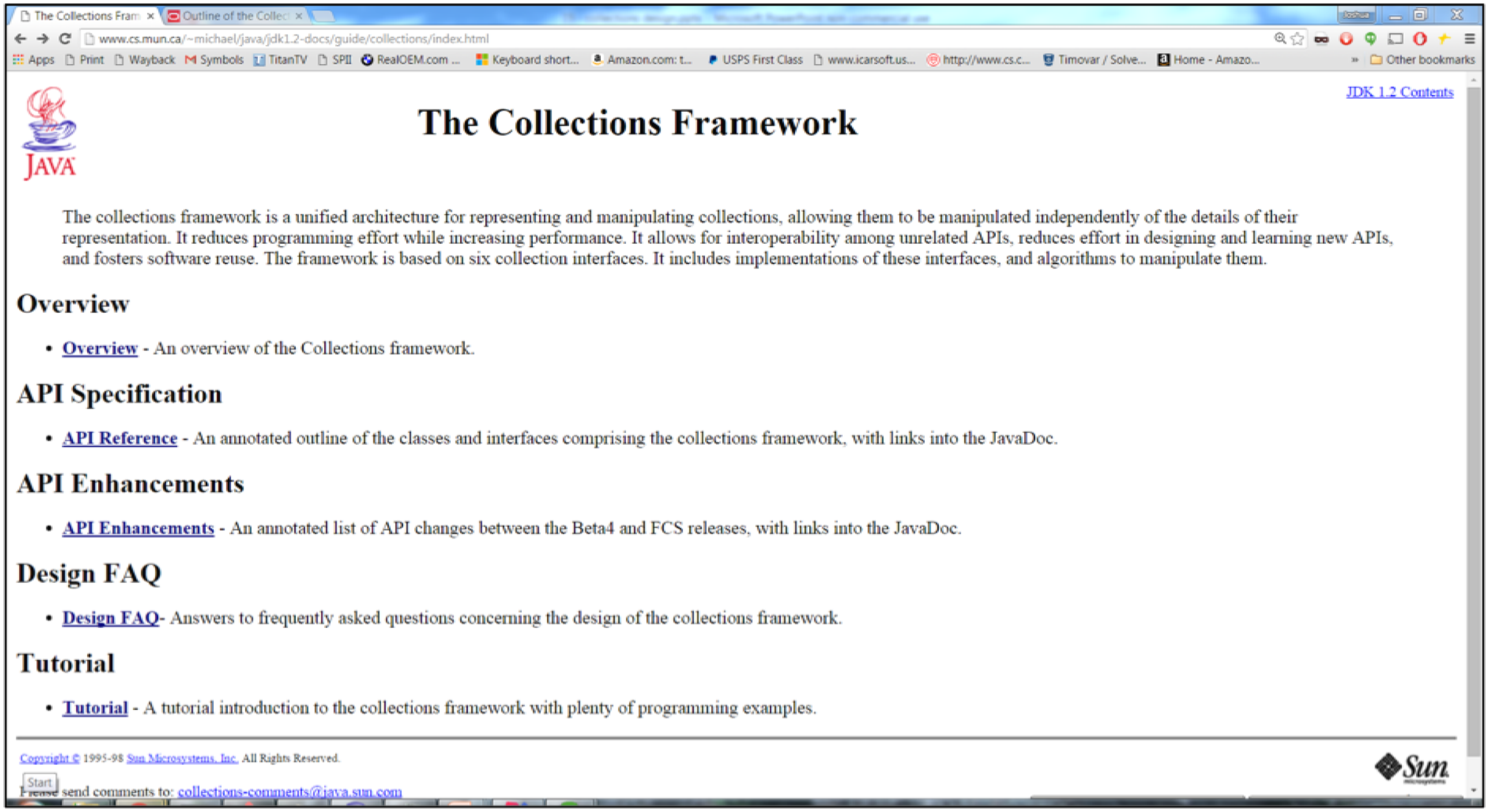
All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

Since:

15-214 48 institute for SOFTWARE RESEARCH

Josh Bloch's experiences designing the Collections framework

A single place to go for documentation



JDK 1.2 Contents

The Collections Framework

The collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse. The framework is based on six collection interfaces. It includes implementations of these interfaces, and algorithms to manipulate them.

Overview

- **Overview** - An overview of the Collections framework.

API Specification

- **API Reference** - An annotated outline of the classes and interfaces comprising the collections framework, with links into the JavaDoc.

API Enhancements

- **API Enhancements** - An annotated list of API changes between the Beta4 and FCS releases, with links into the JavaDoc.

Design FAQ

- **Design FAQ**- Answers to frequently asked questions concerning the design of the collections framework.

Tutorial

- **Tutorial** - A tutorial introduction to the collections framework with plenty of programming examples.

Copyright © 1995-98 Sun Microsystems, Inc. All Rights Reserved.

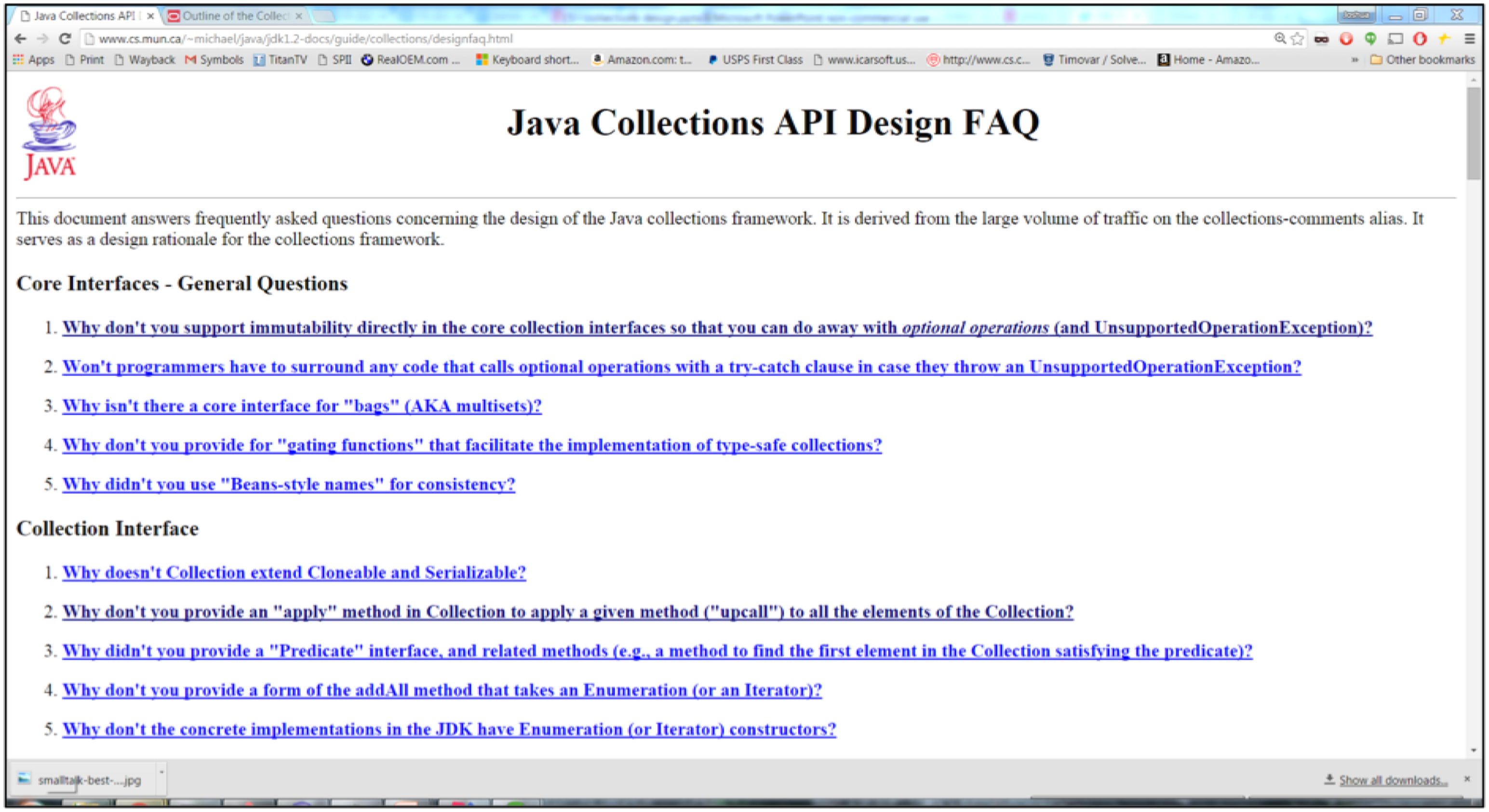
Please send comments to: collections-comments@java.sun.com

15-214 49

Josh Bloch's experiences designing the Collections framework

A design rationale saves you hassle

and provides a testament to history



www.cs.mun.ca/~michael/java/jdk1.2-docs/guide/collections/designfaq.html

Java Collections API Design FAQ

This document answers frequently asked questions concerning the design of the Java collections framework. It is derived from the large volume of traffic on the collections-comments alias. It serves as a design rationale for the collections framework.

Core Interfaces - General Questions

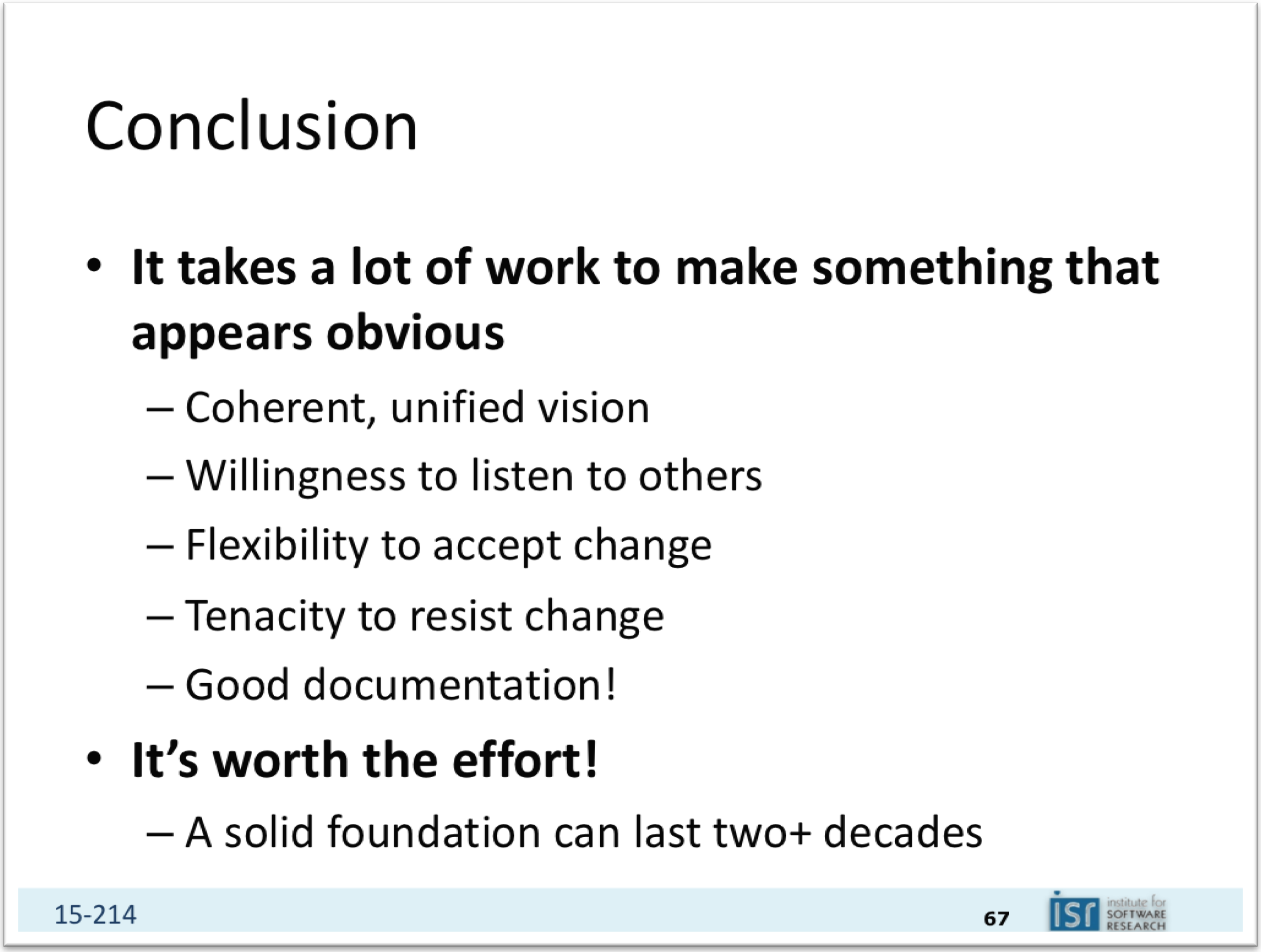
1. Why don't you support immutability directly in the core collection interfaces so that you can do away with *optional operations* (and UnsupportedOperationException)?
2. Won't programmers have to surround any code that calls optional operations with a try-catch clause in case they throw an UnsupportedOperationException?
3. Why isn't there a core interface for "bags" (AKA multisets)?
4. Why don't you provide for "gating functions" that facilitate the implementation of type-safe collections?
5. Why didn't you use "Beans-style names" for consistency?

Collection Interface

1. Why doesn't Collection extend Cloneable and Serializable?
2. Why don't you provide an "apply" method in Collection to apply a given method ("upcall") to all the elements of the Collection?
3. Why didn't you provide a "Predicate" interface, and related methods (e.g., a method to find the first element in the Collection satisfying the predicate)?
4. Why don't you provide a form of the addAll method that takes an Enumeration (or an Iterator)?
5. Why don't the concrete implementations in the JDK have Enumeration (or Iterator) constructors?

15-214 52

Josh Bloch’s experiences designing the Collections framework



Conclusion

- **It takes a lot of work to make something that appears obvious**
  - Coherent, unified vision
  - Willingness to listen to others
  - Flexibility to accept change
  - Tenacity to resist change
  - Good documentation!
- **It’s worth the effort!**
  - A solid foundation can last two+ decades

API design to be continued Thursday