Building a Great API

Eric Stein
Fulminatus Consulting

Slides

- http://www.fulminatus.com/presentations/

Overview

- What is an API?
- Designing
- Implementing
- Evolving

APIs are Everywhere

- Does it say “public” or “protected”?
- Does anyone else rely on it?
- If you write code, you write APIs

Designing an API

Before You Start

- Find three end users
- Choose an API owner
- Choose a specification owner
- Choose a logging owner

Design Process

- Start with use cases
- Write client code and tests first
- Write a short, simple specification
- Write minimal code to support use cases
- Iterate!

Stable

- Locks in users
  - no need to relearn
  - no need to rework
- Less change means fewer bugs
- Write once, support forever

Easy to Read

- Good naming
- Java conventions!
- Use client terminolgy
- Limit method arguments
- Can mom read it?

Easy to Write

- Principle of least astonishment
- Consistent naming
  - limit abbreviations
- Avoid boilerplate code
  - leads to cut-n-paste programming
- Limit method arguments

Powerful Enough

- Limit support to core use cases
  - don‘t overcommit
  - avoid corner cases
  - can always add functionality later
  - more API makes it harder to learn
  - more can go wrong in bigger APIs

Extensible

- Give clients the ability to customize via SPI
- Keep API and SPI in separate classes
  - Allows for evolution later
- Tightly restrict subclassing
  - hard to implement safely
  - strongly consider restricting to SPI

Design for Extension

- Document
  - self-use of overridable methods
  - side effects of overridable methods
- Provide hooks into the class internals
  - prefer protected methods to fields
  - non-hook methods must be final
  - prefer abstract methods to concrete
- Test by writing at least three subclasses
  - preferably written by somebody else

Specification

- Hide implementation details
  - Impossible if documenting for extension
- Spec is a reference, not a novel
  - Duplication inevitable, desirable
- RFC 2119 - specification language
- Clients should never have to look at code

Type Specification

- What do instances represent?
- Construction
- Usage
- Immutability
- Thread safety

Method Specification

- Preconditions
- Postconditions
- Side Effects
- Parameters
  - units, ownership, null handling
- Exceptions
- Thread Safety
- Self Use (if method is extensible)

Implementing an API

Dangers of Allowing Subclassing

- super()
- Constructor, Serializable, Cloneable can‘t call extensible methods
- Serializing code expecting parent class
- Committing to implementation
- Liskov Substitution Principle

Liskov Substitution Principle

```
package java.lang;

public class Rectangle {
    public Rectangle(int x, int y) { .. }
    public void setX(int x) { .. }
    public void setY(int y) { .. }
    public int getArea() { .. }
}

public class Square extends Rectangle { .. }

public int foo(final Rectangle rectangle) {
    rectangle.setX(5); rectangle.setY(12); return rectangle.getArea();
}
foo(new Square(4));
```

Liskov Substitution Principle

```
package java.lang;

public class Square {
    public Square(int side) { .. }
    public void setSide(int side) { .. }
    public int getArea() { .. }
}

public class Rectangle extends Square { .. }

public int foo(final Square square) {
    square.setSide(8); return square.getArea();
}
foo(new Rectangle(5, 5));
```

Properties is a Hashtable?

```
public class Properties
extends Hashtable<Object, Object> {

    public String getProperty(String key) { .. }
    public Object setProperty(String key, String value) { .. }
    public Enumeration<?> propertyNames() { .. }
    public Set<String> stringPropertyNames() { .. }

    /* Inherited from Hashtable */
    public Object get(Object key) { ..}
    public Object put(Object key, Object value) { .. }
    public Enumeration<Object> keys() { ..}
}
```

Properties is not a Hashtable

```
final Properties p = new Properties();
p.put(Integer.valueOf(12), Boolean.TRUE);
p.put("Key", "Value");

System.out.println(p.keys());
// 12, Key

System.out.println(p.propertyNames());
// ClassCastException

System.out.println(p.stringPropertyNames());    // since 1.6
// Key
```

Composition!

```
public final class Properties {
    private final Hashtable<String, String> hash =
        new Hashtable<String, String>();

    public String getProperty(String key) {
        return this.hash.get(key);
    }
    public String setProperty(String key, String value) {
        return this.hash.put(key, value);
    }
    public Enumeration<String> propertyNames() {
        return this.hash.keys();
    }
    public String getProperty(String key, String defaultValue) { .. }
}
```

Immutable Objects

- Can be freely shared
  - so can their internals (Flyweight pattern)
  - no defensive copies
  - thread safe
- Never inconsistent
- Good key for collections
- Easier to read and understand

Disadvantage of Immutability

- Potentially creating many objects
  - Especially in multistep operations
  - Especially if they‘re expensive to create
- VMs good at GCing short-lived objects
- Public mutable peer
- Expose multistep operations on object
  - package-private mutable peer

Building an Immutable Class

- Cannot be extended
  - final class or private constructor
  - don‘t let ‘this‘ escape
- All state set at construction time
  - all fields are final
- Control mutable members
  - defensive copies when returned
  - state not changed after construction
  - are only referenced from the object

Immutable Widget

```
public final class Widget {

    private final Color color;
    private final Dimension size;

    private Widget(Widget.Builder builder) {
        this.color = builder.color;
        this.size = builder.size;
    }

    public Color getColor() { return new Color(this.color.getRGB()); }
    public Dimension getSize() { return new Dimension(this.size); }

    public static class Builder { .. }
}
```

Widget Builder

```
public static final class Builder {

    private Color color;
    private Dimension size = new Dimension(12, 12);

    public Builder(final Color color) {
        this.color = new Color(color.getRGB());
    }
    public Builder size(final Dimension size) {
        this.size = new Dimension(size);
    }
    public Widget build() {
        return new Widget(this);
    }
}
```

Creating a Widget

```
final Widget widget =
    new Widget.Builder(Color.RED)
    .size(new Dimension(12, 24))
    .build();

// OR

final Widget.Builder widgetBuilder =
    new Widget.Builder(Color.BLUE);
...
widgetBuilder.size(new Dimension(55, 18));
widgetBuilder.texture(Texture.COARSE);
...
final Widget widget2 = widgetBuilder.build();
```

Method Signatures

- Get the name right
- Control your parameters
  - no more than four parameters
  - avoid out and in-out parameters
  - avoid multiples of the same type
  - avoid booleans, Strings
- Avoid overloading

Defensive Methods

- Fail quickly and atomically
- Check parameters
  - null, out of range
  - assert, exception
- Defensive copies
- Assume clients will attack your invariants

Exceptions

- Only if something goes wrong
  - not for control flow!
- Strongly prefer unchecked
- Only checked exception if
  - proper use of API
  - can be handled by caller
- Include all failure data in thrown exception
- Always log internal exceptions
  - include stack trace!

Generics

- Good implementation is very powerful
- Don‘t get too crazy
- Hard to mix generics and arrays, varargs
- Don‘t suppress unchecked warnings
- Don‘t return wildcard types

Interfaces

- Use for SPI, but not API
  - clients tend to implement
  - no constructors or statics
  - everything is public
  - not serializable
  - preserves extension for SPI classes
- Abstract classes provide
  - default implementation
  - helper implementation

Logging

- Use a Logging Facade
  - Simple Logging Facade for Java (SLF4J)
  - Jakarta Commons Logging (JCL)
- Trace, Debug are for debugging
- Info, Warn, Error are for the client
  - be consistent!
  - tell the story of the call

Evolving an API

- Maintain backwards compatibility
  - everything matters to someone
  - less important for internal-facing APIs
- Retain ability to evolve in the future

Compatibility

- Behavioral Compatibility
  is the contract still honored?

- Binary Compatibility
  will existing binaries still run?

- Source Compatibility
  will existing source still compile?

Precondition Contracts

```
/**
  * STRONGER
  * @param widget the widget to paint. May not be null.
  * @throws NullPointerException if the widget is null.
  */
public void paint(final Widget widget) { .. }

/**
  *WEAKER
  * @param widget the widget to paint. May be null.
  */
public void paint(final Widget widget) { .. }

// Callers: stronger to weaker is safe
// Implementors: weaker to stronger is safe
```

Postcondition Contracts

```
/**
  * STRONGER
  * @return the weight of the widget. Will always be greater than zero.
  */
public int computeWeight(final Widget widget) { .. }

/**
  * WEAKER
  * @return the weight of the widget, or zero if the widget is null.
  */
public int computeWeight(final Widget widget) { .. }

// Callers: weaker to stronger is safe
// Implementors: stronger to weaker is safe
```

Field Contracts

```
/**
  * STRONGER
  * The very best widget. Never null.
  */
private Widget bestWidget;

/**
  * WEAKER
  * The very best widget, or null if not yet determined.
  */
private Widget bestWidget;

// API
//   setter: stronger to weaker is OK
//   getter: weaker to stronger is OK
```

Other Contract Changes

- Adding or deleting a checked exception
- Deleting a documented unchecked exception
- Changing the order of enum constants
- Adding or removing synchronized

General Binary Compatibility

- Never safe to
  - rename, delete or make less visible
  - change anything‘s kind
  - change return type
  - change between static and non-static
  - move method down type hierarchy
  - change the value of a public constant
  - add the first constructor to a class
  - change existing generic types

Extensible Type Binary Compatibility

- Never safe to
  - add visible fields
  - add visible methods
  - make a field or method abstract
  - make a field or method final

Binary Validation

- japi-checker
- API Compliance Checker
- SigTest
  - last update March 2011
- Japitools
  - old, barely supported

Conclusion

- Designing a good API is Hard
- Many considerations to juggle
- Put the client first

What Makes an API Great?

- Stable
- Easy to work with
- Powerful Enough
- Extensible
- Well documented
- Designed to evolve

Implementing APIs

- Minimize inheritance
  - Never subclass for code reuse
- Maximize immutability
- Write good method signatures
- Write methods defensively
- Throw useful exceptions
- Use generics appropriately
- Use interfaces in SPI, not API
- Write useful log messages

Resources

Effective Java, Second Edition
Joshua Bloch

Practical API Design
Jaroslav Tulach

How to Design a Good API and Why it Matters
Joshua Bloch
http://www.infoq.com/presentations/effective-api-design

Eclipse API Central
http://wiki.eclipse.org/API_Central

NetBeans API Wiki
http://wiki.netbeans.org/API_Design

Questions

- Now?
- Later?
  - eric@fulminatus.com
- Slides
  - http://www.fulminatus.com/presentations/