API design

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
Introduction

Application Programming Interface
The set of functionality provided by a service to its clients

We will focus on internal programming-language based APIs in this lecture

- `java.util.stream package`
- jQuery API
- Contrast with external-business APIs
  - External-business APIs Used by external clients outside the organization which designed the API to access the functionality and data provided by the service
    - Facebook Graph API
    - Google’s APIs
- The two are not mutually-exclusive
External-business APIs

Companies and teams provide APIs to allow clients to access their functionality and data

- An API might provide (limited) access to an internal database
- Give access to computational power
- Define protocols for event subscription
- Allowing developers to build applications on top of their APIs
Internal APIs

The language induced by the protocols and signatures of a software module

- A method’s API is its signature and promises
  - I.e., its input, output, and side-effects
- A class’s API is its **public and protected** methods
  - Why protected? Because they are usable outside the package
- A library’s API its set of public-facing classes and functions
  - In Java/C# functions do not exist outside of a class, but this is not true for all languages
  - In other languages, a file is usually referred to as a module, e.g., Python, C++, Haskell, Javascript
Names of API elements tend to follow a structured pattern

- Names need to convey as much information as possible
- **Consistency** is more important than **readability**
  - If a different term is applied, it is assumed the difference is significant
  - Of course, there’s usually no reason you can’t have both
- Popular elements get shorter names
  - Every added term is an additional **qualification**
  - Shorter to read and write
  - Signifies importance
Methods. Usually verbs
- open, run, add, reverse
- But for data objects, sometimes the named of the field is used instead of \texttt{getX / setX}

```java
// Getters and setters
\texttt{int \; getX();}
\texttt{void \; setX(\texttt{int} \; x);}  
// Without the verbose prefix
\texttt{int \; x();}
\texttt{void \; x(\texttt{int} \; newX);}  
```

- Boolean methods are named as questions
  - isVisible(), contains(E e)
- Single (public) method classes can use \texttt{apply}
  - downloader.download(url) becomes downloader.apply(url)
  - More declarative
Classes and types are virtually always nouns
- When emphasis is on data, named after a business domain object
  - File, Figure, List, etc.
- When emphasis is on functionality, will be a verb-noun
  - Writer,Iterable, etc.
  - able is usually used for mixin-like interfaces
    - A class can be Iterable, Writeable, Obeservable, etc.
    - Often combined with the façade pattern
Subclasses names

Subclasses prefix the superclass name with qualifying information

```java
class Figure
class ResizableFigure extends Figure
class ColoredFigure extends Figure
class MonochromeColoredFigure extends ColoredFigure
```
Interface vs Class names

Different idioms in different languages

- **Java**
  - General name for interface
  - Specific name for class

```
interface List
class ArrayList implements List
```

- Pre-Java 8 is was common to supply an abstract class with defaults to reduce code duplication between implementations

```
interface List
abstract class AbstractList implements List
class ArrayList extends AbstractList
```

- If a private implementation is hidden behind a public interface, it’s common to suffix the class with “Impl”

```
interface FileFactory
class FileFactoryImpl implements FileFactory
```
Different idioms in different languages

- **C#**
  - Interface is prefixed with "I"
  - Common implementation without "I"

```csharp
interface IList
// Actually an ArrayList
class List : IList
```

- Less explicit, but easier for new users
- Easy to differentiate between class and interface
Avoid useless **noise words** suffixes and prefixes

- **Method name clutter**: calculate, compute, get/set, method names repeating the class name
  - Although the difference between get and compute *could* signify the difference between a cheap and expansive computation
- **Class name clutter**: Data, Manager, Algorithm
Visibility: Methods

In ascending order of visibility:

1. **private**: helper implementation functions
   - If you can, make these methods `static` (PLoP)

2. **default**: Used for exposing methods for testing
   - Useful for providing alternative constructors
   - Avoid in other cases if you can

3. **protected**: Visible to subclasses and other classes in the package
   - Exposes functionality to subclasses
     - But we saw better alternatives (composition, template pattern)
   - Used in template pattern (in which case, they will be `abstract`)
   - Avoid (ab)using package visibility
   - Part of the class’s API, and therefore should be documented
     - And tested if not abstract

4. **public**: The class’s outward facing API
   - Should not be `static`
     - Can’t use dependency injection
     - Can’t override interface methods

Always default to the most private option possible for **public** classes
Visibility: Classes

- Only two modifiers possible for classes: **default** and **public**
- **default** should be, well, the default
  - Criminally underused
  - A public class is available **everywhere**
    - Namespace pollution
    - Confuses users of your API
    - And, of course, breaks encapsulation
- You should only make a class public if you **need** it to be used outside of your package
Method visibility in package-private classes

- Practically, all methods of a package-private class have either `private` or `default` visibility.
- But we could use `public` and `protected` keyword to aid documentation.
- The same rules apply to package-private classes:
  1. `default` is for method visible for testing.
  2. `protected` is for subclasses.
  3. `public` is for clients of the class.
Encapsulation

- Encapsulation is particularly important when developing API for outside clients
  - That is, clients outside your core group
- Remember, exposition breeds coupling
  - While a coupled class is more powerful, it limits both you and your clients
  - Flexibility is good for all actors involved
- Expose interfaces, not classes
  - Avoid constructors, use static factory methods
  - Expose the smallest interface you can
API Breadth

How many “convenience” methods should we expose?

1. **Thin API.** Nothing! Expose just the bare necessities
2. **Fat API.** Everything! Provide every possible functionality
3. **Humane API.** Middle ground: expose the most commonly used functions
Bare bones list interface:

```java
interface List<E> {
    T get(int i);
    int size();
    void set(E e);
    void remove(int i);
}
```

Enough to implement all other functionality above it
Humane API

- Other languages have much richer standard library functionality
  - **Ruby.** Over 110
  - **Scala.** over 160 methods
  - **C#.** over 170 methods
- Spectrum, not a dichotomy
  - Java’s List has almost 30
  - And 30+ more from Stream
API breadth: Pros and Cons

- **Thin API pros:**
  - Easier to develop (less up-front cost)
  - Easier to maintain
  - YAGNI

- **Humane API pros:**
  - Easier to use
  - Less work for clients
  - And less boilerplate!
  - Generally speaking, an API will have more clients than developers (otherwise, why write it?)
  - Clients don’t have to reinvent the wheel every time
    - Uniform code across different projects
    - Pretty much every stand-alone Java library has to define or import Pair, since it isn’t in the standard library
  - Code is more declarative
    - size() != 0 vs. !isEmpty() vs. nonEmpty()
Boilerplate

Boilerplate code

Long code sections included in many places with little or no alteration

- Can be caused by the language
  - For example, `#IFNDEF`
- Or of the API
  - **Python** `max(list)`
  - **Scala/C#** `list.max()`
  - **Haskell** `max list`
  - **Java** `list.mapToDouble(Double::value).max().getAsDouble()`
Boilerplate: more examples

Can be large:

```java
// Could have been avoided if InputStream just had a
// readString() method
public String toString(InputStream is) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    byte[] buffer = new byte[1024];
    int length;
    while ((length = inputStream.read(buffer)) != -1) {
        baos.write(buffer, 0, length);
    }
    return baos.toString("UTF-8");
}
```

Or small:

```java
// why not have a toList function?!
stream.collection(Collectors.toList());
```
Boilerplate summary

Boilerplate is bad and you should avoid it

- Boilerplate code is noise that makes the code harder to read
- Code is harder to use without guidance
- Can be a cause of bugs
  - For example, **==** vs **.equals** in Java
- Rule of thumb guidance: what’s common should be easy, what’s uncommon should be possible
- A lot of boilerplate is usually a sign your abstraction level is too low

Breadth
YAGNI stands for You Aren’t Gonna Need It

- Don’t implement features you don’t need
- Avoid going out of your way to make your code more generic than it needs to be *write now*
- “But wait, isn’t our first assumption that software changes?!”
YAGNI stands for **You Aren’t Gonna Need It**

- Don’t implement features you don’t need
- Avoid going out of your way to make your code more generic than it needs to be *write now*
- “But wait, isn’t our first assumption that software changes?!”
  - YAGNI doesn’t claim that software won’t change, but rather that *you* can’t determine how
  - Building things you don’t need costs you resources you don’t have
  - More generic code is almost always more complicated code
Common scenario: you need to parse data coming in from a provider
- The provider uses the same API for their clients
- The data is serialized in proprietary XML format
- But you don’t want to lock yourself to a single vendor...
- Solution: **canonize** the vendor XML using, e.g., XSL
Old format

```xml
<root>
    <team>
        <name>...</name>
        <alias>...</alias>
        <players>
            ...
        </players>
    </team>
</root>
```

Canonical format

```xml
<my-root>
    <!-- alias isn't used -->
    <team name=...>
        <player>...</player>
    </player>
</team>
</my-root>
```
YAGNI results

- You spend a lot of time building an extra translation layer
- Every time you want to add a new feature, you have to go through an extra step
  - Twice as much work
  - Twice as many bugs
- It’s not certain you saved any work
  - Changing vendors still requires writing a lot of new canonizing code
- And in the end, you don’t even change vendors!
“But if we did, wouldn’t we have saved time?”
“But if we did, wouldn’t we have saved time?”

- You could have treated the original format as canon, and converted future versions to it.
- The time could have been better spent on things you need now, rather than things you might need later.
- It’s possible that with new information and experience under your belt, you would have chosen a better solution.
YAGNI (cont.)

YAGNI is **not** an excuse to write bad code!

- “Yagni requires (and enables) malleable code.” (MF)
- Your code *will* change, and bad design *will* hinder you productiveness
- You can write good code without having to guess future features will be required
- Focus on tests, DRY, high cohesion, and low coupling
  - But, don’t abstract what you don’t need if it overcomplicates your code
- When the time comes and you have to add new features, you will be ready
Checked exceptions are an example of a good idea with bad execution

- You rarely want to catch exceptions
  - Instead, you would rather fail early and get the stack-trace
- Dealing with them is incredibly verbose
  - A single line becomes five
  - They don’t play well with lambdas
- Worse of all, they have the tendency to leak implementation details

```java
// Leaking SQL implementation
public List<User> getUsers() throws SQLException;
```
Checked exceptions: the good parts

But checked exceptions can be useful when used internally

- Making sure all edge cases are taken care of
- Separating pure and non-pure code, error-prone from total functions, etc.

```java
// So much can go wrong here, each prompting its own
// handling logic...
public void registerUser(String userId, String password);
// With checked exceptions the compiler helps verify
// we don't forget anything
public void registerUser(String userId, String password) throws UserAlreadyExistsException,
InvalidUserNameException,
InvalidPasswordException,
IOException;
```

We’ll talk about exceptions more when we discuss functional programming
In general, it’s best to **avoid** exception handling in your API at all

- Not always possible to know how clients want to handle exceptions
- Instead, let clients decide how to deal with exceptions
- Increase flexibility and cohesion, reduces client coupling, increase exception handling explicitness at client
- But either way, it’s best to **fail early, fail often**
  - Garbage-in-garbage-out (GIGO) is a **terrible** policy
API Discoverability

(Don’t confuse with API discovery)

The discoverability of an API is a measure of how easy it is to find and use all of its features.

- Is your API easy to pick up?
- Do users intuitively know where to find features and configuration options?
- Does your API guide users?
- Can users figure out your API without constantly consulting arcane documentation and the internet?
Discoverability: how to achieve

How can we increase the discoverability of our API?

- Narrow public interfaces
  - Remember the **interface segregation principle**
- Consistent naming
- Preferring types to primitives
  - Later in the course
- Fluent APIs
  - Chaining method calls, e.g., as in Stream
  - Later in the course
- Static factory methods instead of constructors
  - Can be named, raveling intent
  - **Lots of other benefits**