



Java is a trademark of Sun Microsystems, Inc.



JavaOneSM

Introduction to Google Guice: Programming is fun again!

Jesse Wilson
Google, Inc.
<http://code.google.com/p/google-guice/>

What you'll get out of this talk

Dependency Injection and Guice

- > Objects come to you
 - Instead of ‘new’ and factories
- > Reusable modules
- > First-class scopes
- > Easier tests
 - and more confidence
- > Less boilerplate

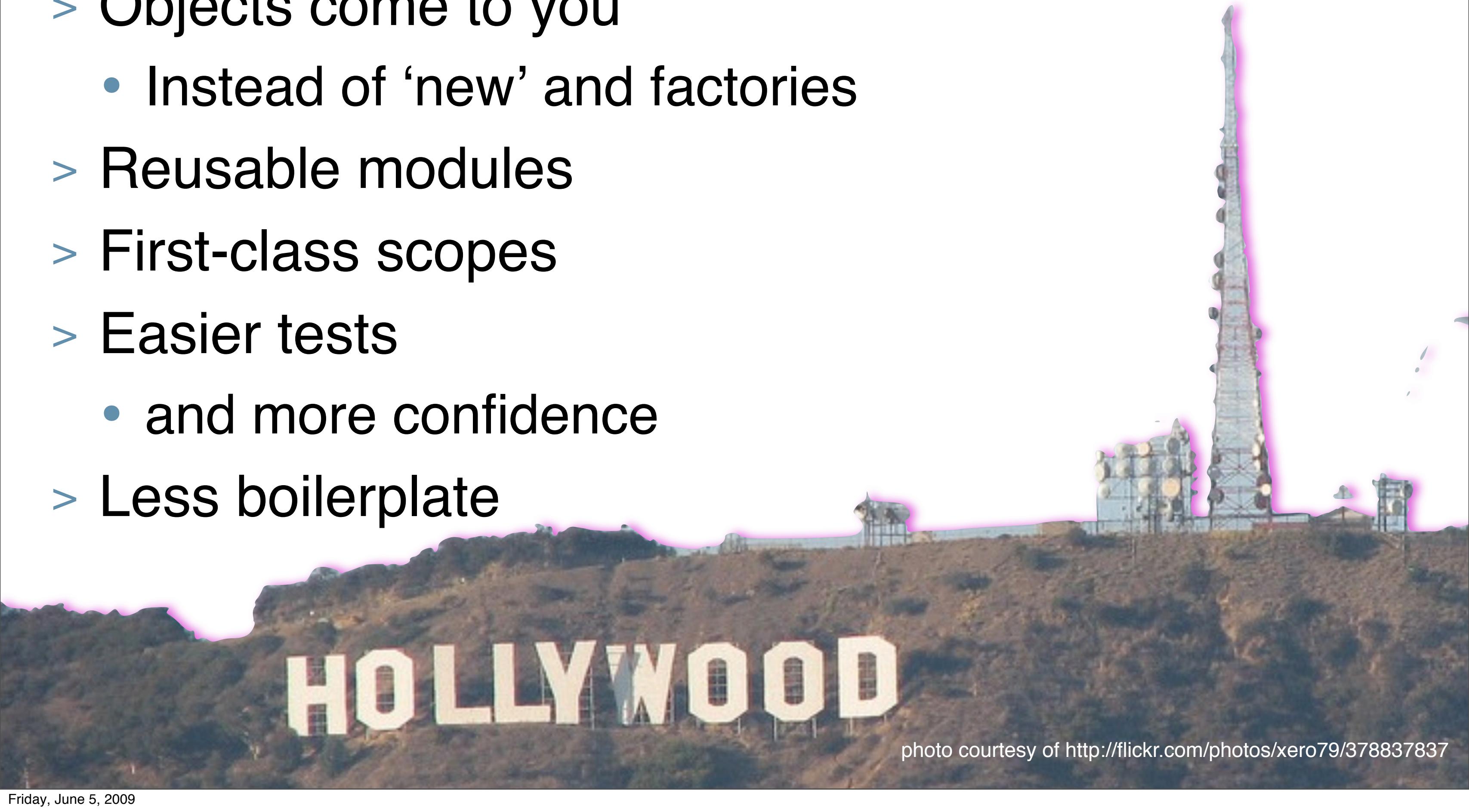
A composite image featuring the iconic Hollywood sign in Los Angeles, California, and the Eiffel Tower in Paris, France. The two landmarks are positioned side-by-side against a clear blue sky. A large, stylized pink cloud or thought bubble originates from the bottom left and sweeps across the slide, partially obscuring the text and the base of the Hollywood sign.

photo courtesy of <http://flickr.com/photos/xero79/378837837>

Overview

- > **Motivation**
 - Ugh, factories aren't fun
- > **Using Guice**
 - `@Inject` is the new `new`
- > **Leveraging Guice**
 - A tour of extensions and advanced features

Example tweet tweet

- > Setting the stage
- > Constructors
- > Factories
- > Dependency Injection
 - by hand
 - with Guice



jessewilson

Follow



photo courtesy of
<http://flickr.com/photos/jessicafm/62271212>

Code you might write

A tweets client

```
public void postButtonClicked() {  
    String text = textField.getText();  
    if (text.length() > 140) {  
        Shortener shortener = new TinyUrlShortener();  
        text = shortener.shorten(text);  
    }  
    if (text.length() <= 140) {  
        Tweeter tweeter = new SmsTweeter();  
        tweeter.send(text);  
        textField.clear();  
    }  
}
```



Calling Dependencies' Constructors Directly



photo courtesy of http://flickr.com/photos/salford_ian/3134250986

Getting dependencies via their constructors ...calling **new** directly doesn't afford testing

```
public void postButtonClicked() {  
    String text = textField.getText();  
    if (text.length() > 140) {  
        Shortener shortener = new TinyUrlShortener();  
        text = shortener.shorten(text);  
    }  
    if (text.length() <= 140) {  
        Tweeter tweeter = new SmsTweeter();  
        tweeter.send(text);  
        textField.clear();  
    }  
}
```

We post to tinyurl.com and send an SMS for each test! This is neither fast nor reliable.

Getting dependencies via their constructors ...calling `new` directly doesn't afford testing

```
public void postButtonClicked() {  
    String text = textField.getText();  
    if (text.length() > 140) {  
        Shortener shortener = new TinyUrlShortener();  
        text = shortener.shorten(text);  
    }  
    if (text.length() <= 140) {  
        Tweeter tweeter = new SmsTweeter();  
        tweeter.send(text);  
        textField.clear();  
    }  
}
```

We post to tinyurl.com and send an SMS for each test! This is neither fast nor reliable.

Getting Dependencies from Factories



photo courtesy of http://flickr.com/photos/abulic_monkey/130899453

Getting dependencies from factories

```
public void postButtonClicked() {  
    String text = textField.getText();  
    if (text.length() > 140) {  
        Shortener shortener = ShortenerFactory.get();  
        text = shortener.shorten(text);  
    }  
    if (text.length() <= 140) {  
        Tweeter tweeter = TweeterFactory.get();  
        tweeter.send(text);  
        textField.clear();  
    }  
}
```



Implementing the factory all of this boilerplate slows you down. Ugh!

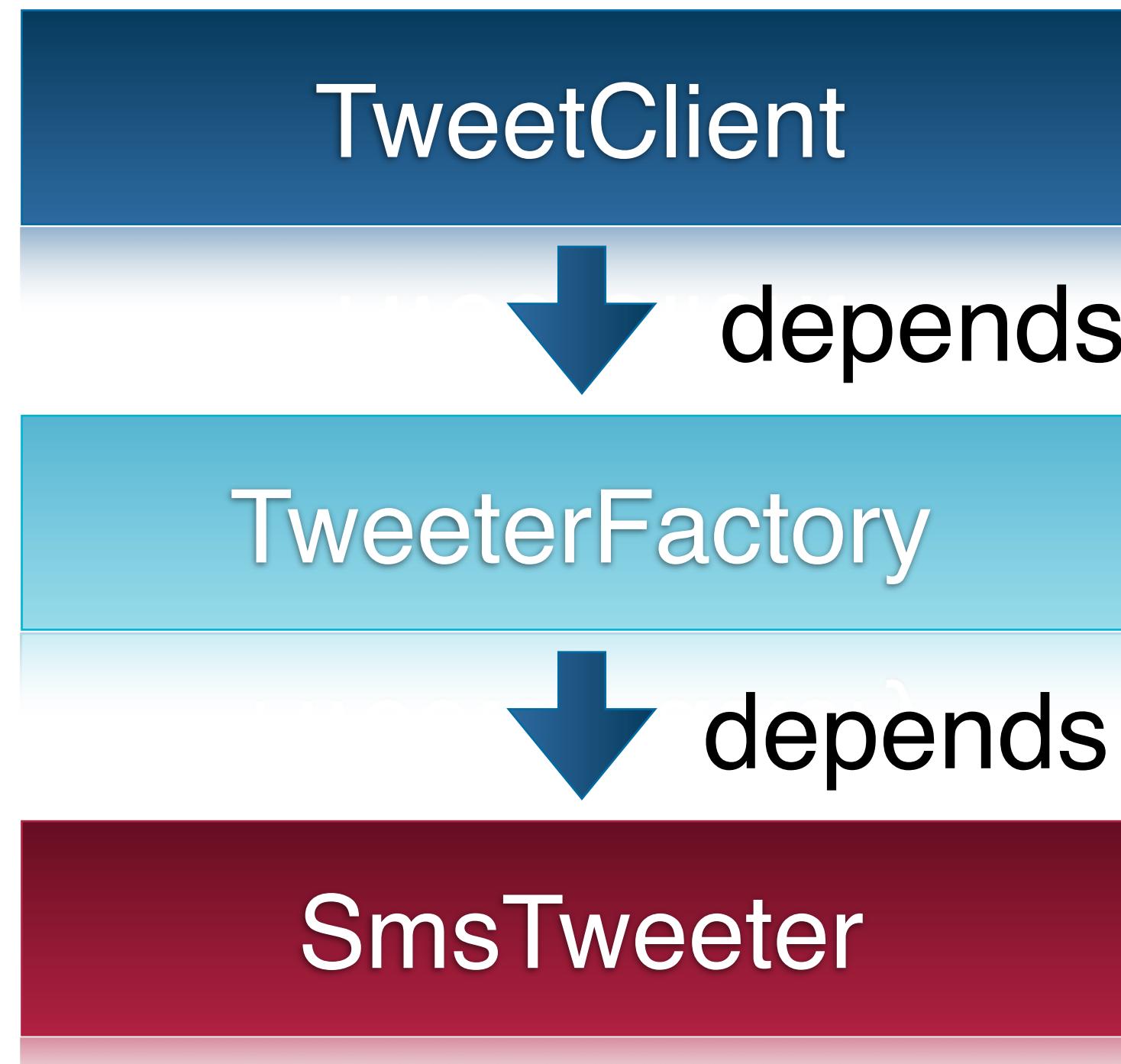
We still have to
write a factory
for the URL
shortener.

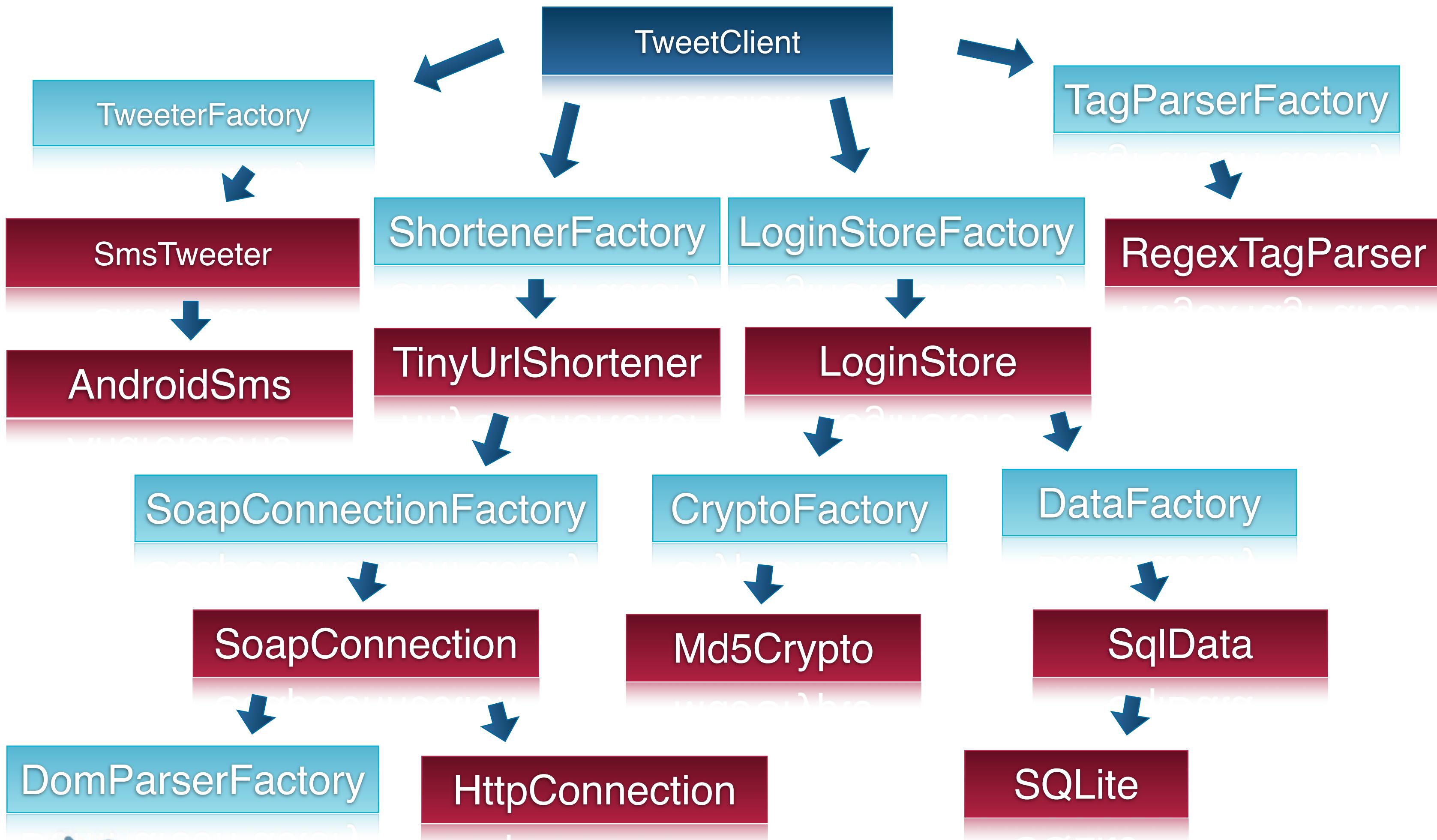
Implementing the factory all of this boilerplate slows you down. Ugh!

```
public class TweeterFactory {  
  
    private static Tweeter testValue;  
    public static Tweeter get() {  
        if (testValue != null) {  
            return testValue;  
        }  
        return new SmsTweeter();  
    }  
  
    public static void setForTesting(Tweeter tweeter) {  
        testValue = tweeter;  
    }  
}
```

We still have to
write a factory
for the URL
shortener.

Factory dependency graph the static dependency causes monolithic compiles...





Testing with a factory it's testable

```
public void testSendTweet() {  
    MockTweeter tweeter = new MockTweeter();  
TweeterFactory.setForTesting(tweeter);  
  
    TweetClient tweetClient = new TweetClient();  
    tweetClient.getEditor().setText("Hello!");  
    tweetClient.postButtonClicked();  
    assertEquals("Hello!", tweeter.getSent());  
  
}
```



Testing with a factory

it's testable ...but don't forget to clean up

```
public void testSendTweet() {  
    MockTweeter tweeter = new MockTweeter();  
TweeterFactory.setForTesting(tweeter);  
  
    TweetClient tweetClient = new TweetClient();  
    tweetClient.getEditor().setText("Hello!");  
    tweetClient.postButtonClicked();  
    assertEquals("Hello!", tweeter.getSent());  
  
TweeterFactory.setForTesting(null);  
}
```



Testing with a factory

it's testable ...but don't forget to clean up ...properly!

```
public void testSendTweet() {  
    MockTweeter tweeter = new MockTweeter();  
TweeterFactory.setForTesting(tweeter);  
    try {  
        TweetClient tweetClient = new TweetClient();  
        tweetClient.getEditor().setText("Hello!");  
        tweetClient.postButtonClicked();  
        assertEquals("Hello!", tweeter.getSent());  
    } finally {  
        TweeterFactory.setForTesting(null);  
    }  
}
```



Dependency Injection (DI) by hand



Dependency injection by hand objects come to you

```
public class TweetClient {  
  
    private final Shortener shortener;  
    private final Tweeter tweeter;  
  
    public TweetClient(Shortener shortener, Tweeter tweeter) {  
        this.shortener = shortener;  
        this.tweeter = tweeter;  
    }  
  
    public void postButtonClicked() {  
        ...  
        if (text.length() <= 140) {  
            tweeter.send(text);  
            textField.clear();  
        }  
    }  
}
```

Dependency
Injection:
rather than
looking it up, get
it passed in.

Testing with dependency injection no cleanup required

```
public void testSendTweet() {  
    MockShortener shortener = new MockShortener();  
    MockTweeter tweeter = new MockTweeter();  
    TweetClient tweetClient  
        = new TweetClient(shortener, tweeter);  
    tweetClient.getEditor().setText("Hello!");  
    tweetClient.postButtonClicked();  
    assertEquals("Hello!", tweeter.getSent());  
}
```

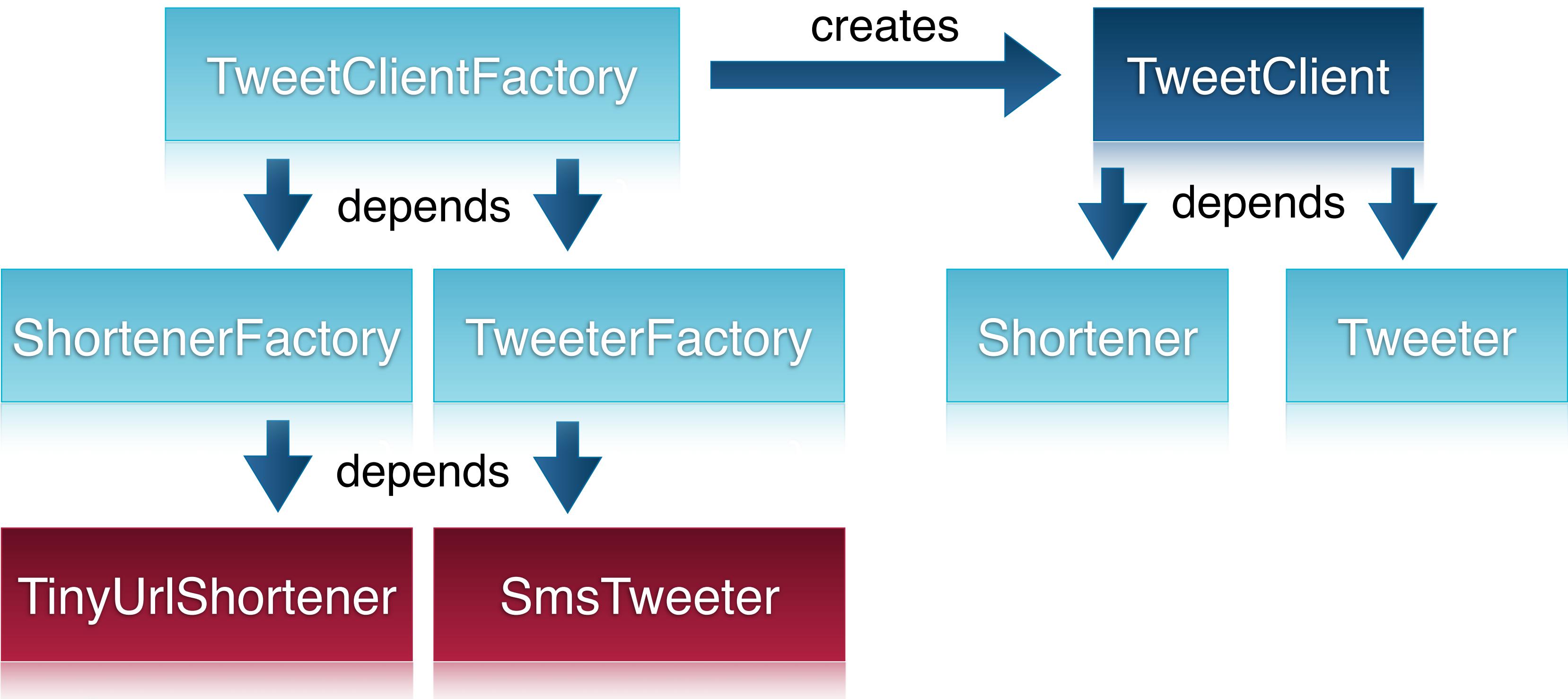
Where does the dependency go? ugh, you still have to write boilerplate code to build stuff

```
public class TweetClientFactory {  
    private static TweetClient testValue;  
  
    public static TweetClient get() {  
        if (testValue != null) {  
            return testValue;  
        }  
    }
```

DI motto:
Push dependencies
from the core to
the edges

```
    Shortener shortener = ShortenerFactory.get();  
    Tweeter tweeter = TweeterFactory.get();  
    return new TweetClient(shortener, tweeter);  
}
```

Where does the dependency go? your application code sheds its heavyweight dependencies

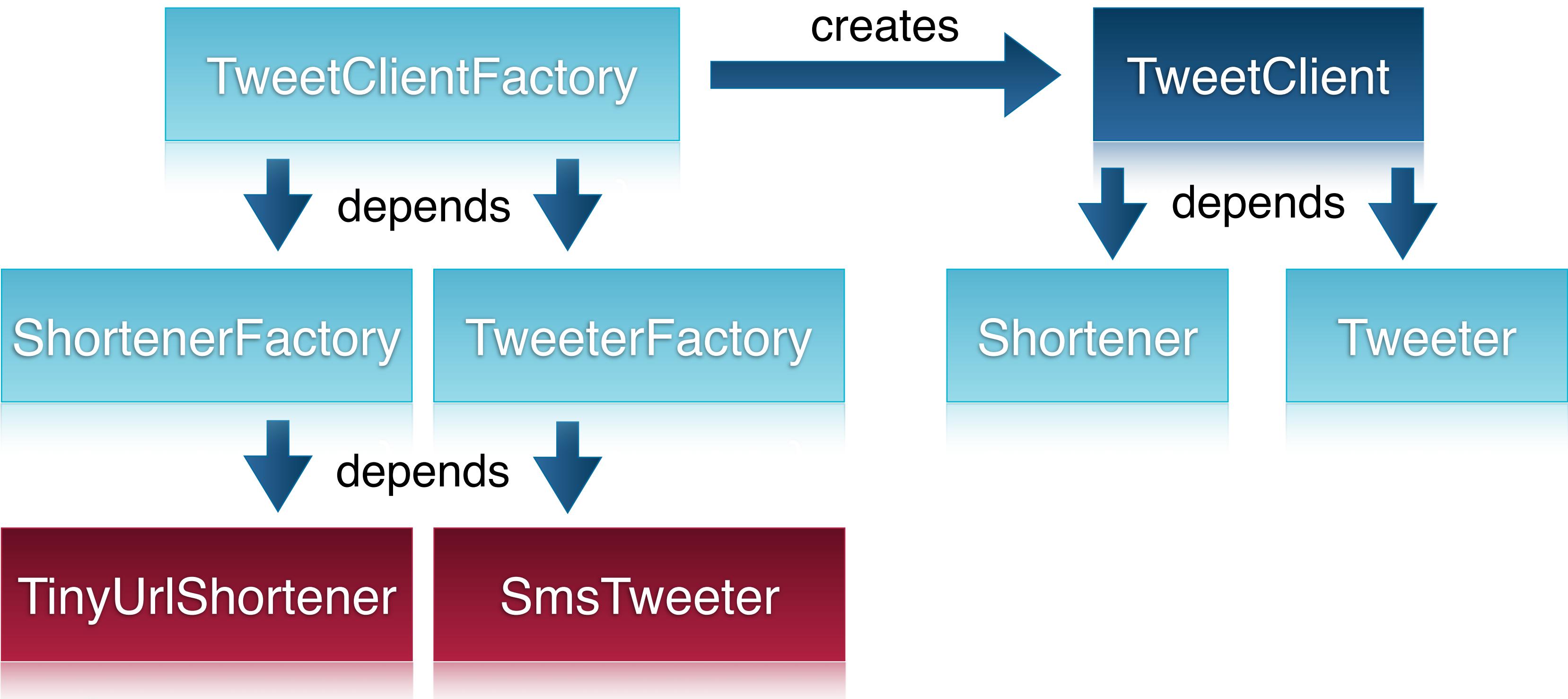


Dependency Injection with Guice

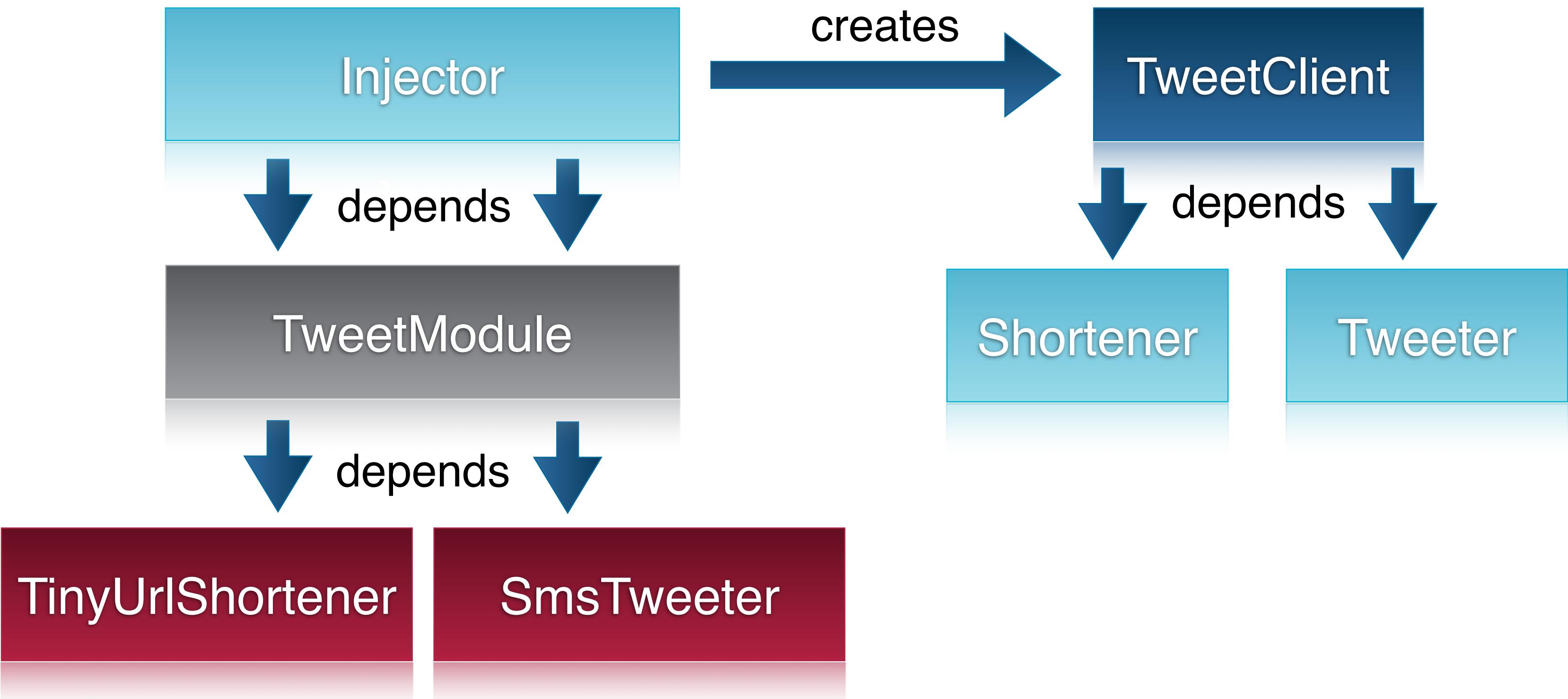


photo courtesy of <http://flickr.com/photos/randysonofrobert/347327376>

Dependency injection with Guice



Dependency injection with Guice



Configuring the injector using modules

```
import com.google.inject.AbstractModule;

public class TweetModule extends AbstractModule {
    protected void configure() {
    }

}
```

Configuring the injector using modules

```
import com.google.inject.AbstractModule;

public class TweetModule extends AbstractModule {
    protected void configure() {
        bind(Tweeter.class)
            .to(SmsTweeter.class);

        bind(Shortener.class)
            .to(TinyUrlShortener.class);
    }
}
```

Telling Guice to use your constructor annotate a constructor with @Inject

```
import com.google.inject.Inject;

public class TweetClient {
    private final Shortener shortener;
    private final Tweeter tweeter;

    public TweetClient(Shortener shortener, Tweeter tweeter) {
        this.shortener = shortener;
        this.tweeter = tweeter;
    }
}
```

Telling Guice to use your constructor annotate a constructor with @Inject

```
import com.google.inject.Inject;

public class TweetClient {
```

```
    private final Shortener shortener;
    private final Tweeter tweeter;
```

@Inject

```
public TweetClient(Shortener shortener, Tweeter tweeter) {
    this.shortener = shortener;
    this.tweeter = tweeter;
}
```

Bootstrapping Guice

```
public static void main(String[] args) {  
    Injector injector = Guice.createInjector(new TweetModule());  
  
    TweetClient tweetClient  
        = injector.getInstance(TweetClient.class);  
  
    tweetClient.show();  
}
```

Using Guice

- > Why?
- > Bindings
- > Scopes
- > Injections



Why use Guice?

- > Writing boilerplate slows you down
- > More up front type checking
- > **It makes it easier to write better code**

- > Plus...
 - Scopes
 - AOP
 - Tight integration with web, data access APIs, etc.

Guice in a nutshell

- > Classes have dependencies
 - these are passed in automatically
 - identified by annotations
- > Modules define how dependencies are resolved

Bindings



photo courtesy of <http://flickr.com/photos/uwehermann/3417729678>

Bindings

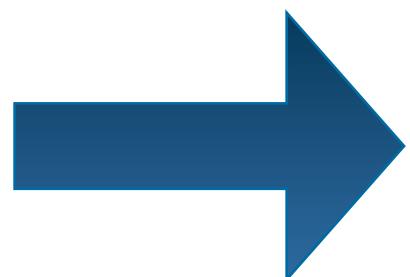
map types to their implementations

```
public class TweetModule extends AbstractModule {  
    protected void configure() {  
        bind(TweetClient.class);  
        bind(Tweeter.class)  
            .to(SmsTweeter.class);  
        bind(String.class).annotatedWith(Username.class)  
            .toInstance("jesse");  
    }  
  
    @Provides Shortener provideShortener() {  
        return new TinyUrlShortener();  
    }  
}
```



Constructor Bindings to resolve a type, call its constructor

TweetClient



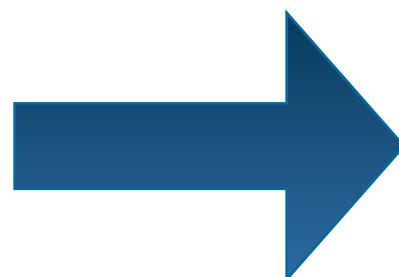
new TweetClient(...)

Constructor Bindings

to resolve a type, call its constructor

```
public class TweetModule extends AbstractModule {  
    protected void configure() {  
        bind(TweetClient.class);  
    }  
}
```

TweetClient

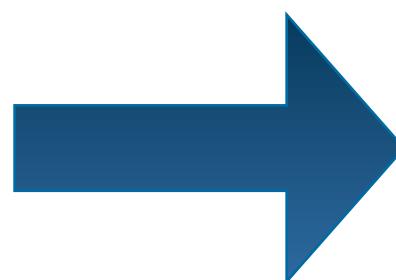


new TweetClient(...)

- > Requires @Inject on the constructor
 - Dependencies are passed in as parameters

Provider methods
to resolve a type, call this method

Shortener

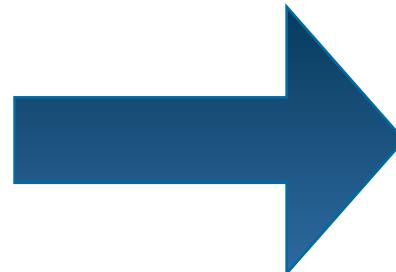


provideShortener(...)

Provider methods to resolve a type, call this method

```
public class TweetModule extends AbstractModule {  
    protected void configure() {...}  
  
    @Provides Shortener provideShortener() {  
        return new TinyUrlShortener();  
    }  
}
```

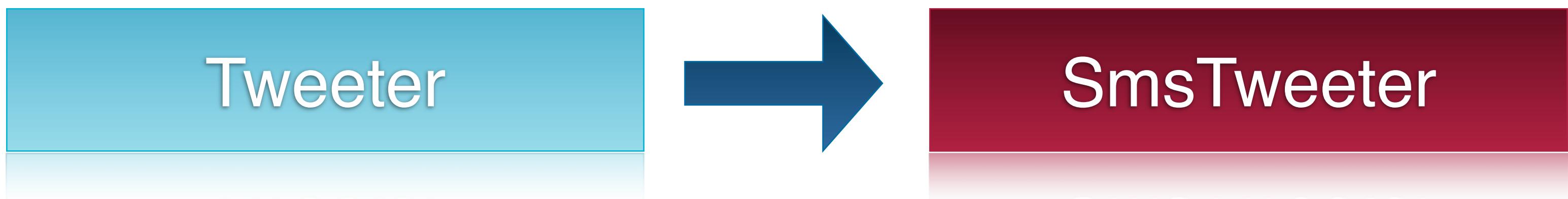
Shortener



provideShortener(...)

- > Annotate a module method with `@Provides`
 - The return type is bound
 - Dependencies are passed in as parameters

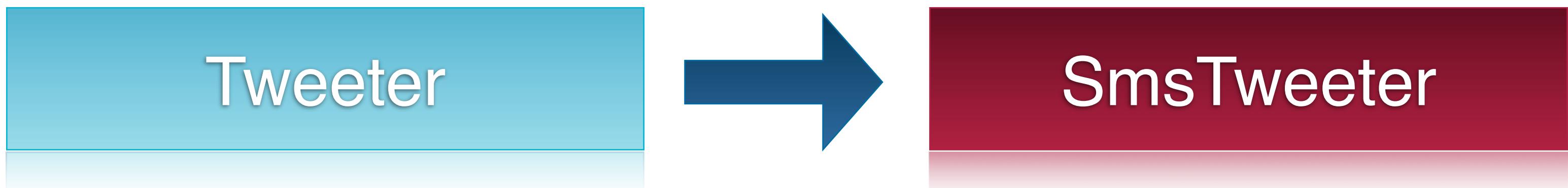
Linked Bindings to resolve a type, use another binding



Linked Bindings

to resolve a type, use another binding

```
public class TweetModule extends AbstractModule {  
    protected void configure() {  
        bind(Tweeter.class).to(SmsTweeter.class);  
    }  
}
```



- > Requires a binding for the target type
 - If none exists, one will be created automatically

Linked Bindings

to resolve a type, use another binding

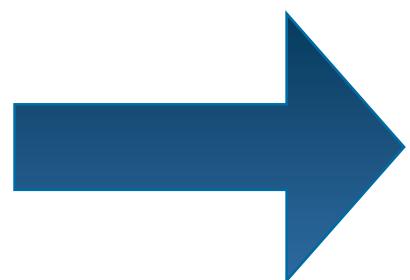
```
public class TweetModule extends AbstractModule {  
    protected void configure() {  
        bind(Tweeter.class).to(SmsTweeter.class);  
    }  
}
```



- > Requires a binding for the target type
 - If none exists, one may be created automatically...

Binding Annotations uniquely identify a binding

@Username String

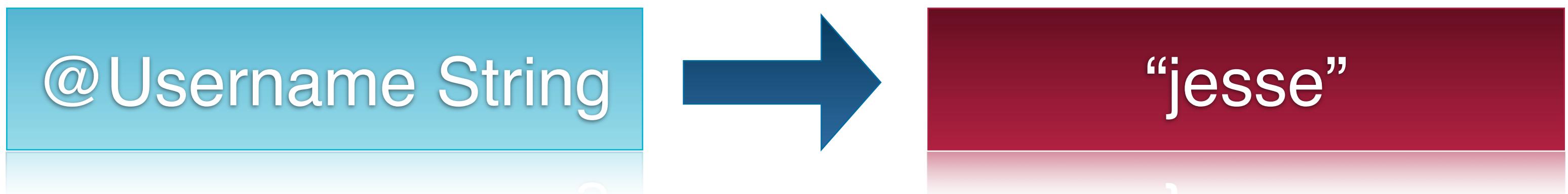


“jesse”

Binding Annotations

uniquely identify a binding

```
protected void configure() {  
    bind(String.class).annotatedWith(Username.class)  
        .toInstance("jesse");  
}
```



```
@Inject  
public SmsTweeter(@Username String username) {  
    this.username = username;  
}
```



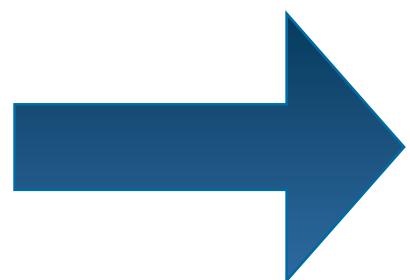
Defining your own binding annotations

```
@BindingAnnotation  
@Retention(RUNTIME)  
@Target({FIELD, PARAMETER, METHOD})  
public @interface Username {}
```

- > This boilerplate defines a binding annotation
- > Everything is compile-time checked
 - IDE autocomplete, import, find usages
 - Avoids clumsy string matching

Instance Bindings always use the same value

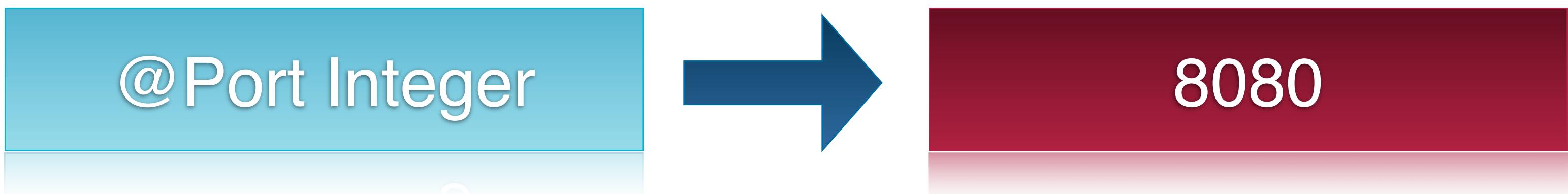
@Port Integer



8080

Instance Bindings always use the same value

```
protected void configure() {  
    bind(Integer.class).annotatedWith(Port.class)  
        .toInstance(8080);  
}
```



- > Best suited for value objects such as a database name, or webserver port

Using Scopes



photo courtesy of <http://flickr.com/photos/houseofsims/3139640931>

Scopes

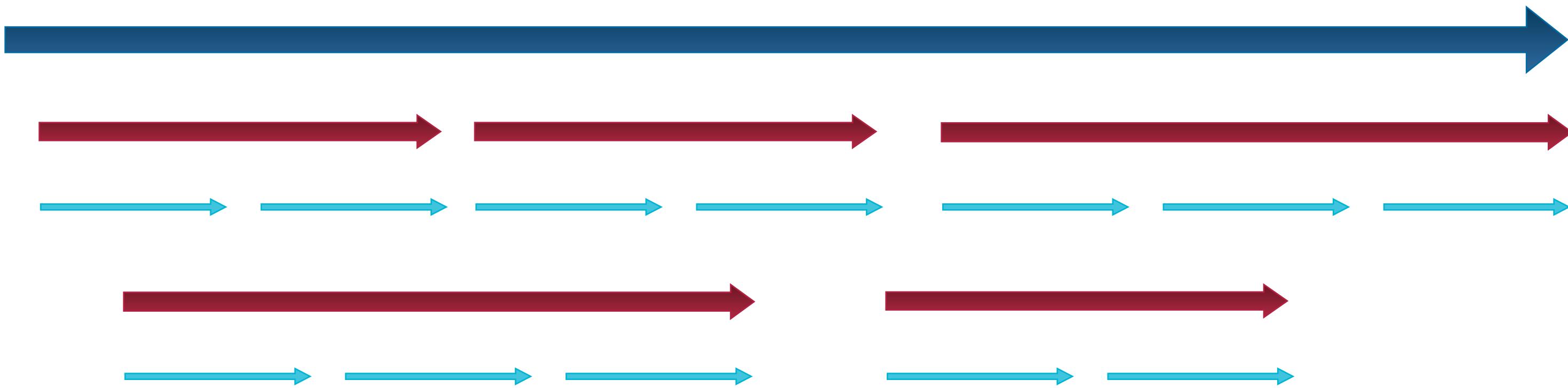
manage how many

- > Scopes manage how instances are reused
 - because they're **stateful**
 - or **expensive** to construct or lookup
 - or expensive to maintain

Scopes

manage how many

- > Scopes manage how instances are reused
 - because they're **stateful**
 - or **expensive** to construct or lookup
 - or expensive to maintain



Common scopes

- > **Unscoped:** one per use
 - create it, use it, and toss it!
 - often the best choice
- > **@Singleton:** one per application
 - for heavyweight resources
 - and application state
- > **@RequestScoped:** one per web or RPC request
- > **@SessionScoped:** one per HTTP session

Everything is unscoped by default in Guice.

Applying scopes

the best way is to annotate a class with its scope

```
public class TweetClient {  
    ...  
  
    @Inject  
    public TweetClient(Shortener shortener, Tweeter tweeter) {  
        this.shortener = shortener;  
        this.tweeter = tweeter;  
    }  
}
```

Applying scopes

the best way is to annotate a class with its scope

@Singleton

```
public class TweetClient {
```

```
...
```

@Inject

```
public TweetClient(Shortener shortener, Tweeter tweeter) {  
    this.shortener = shortener;  
    this.tweeter = tweeter;  
}
```

Applying scopes

you can specify scopes in a module

```
public class TweetModule extends AbstractModule {  
    protected void configure() {  
        bind(ConnectionPool.class)  
            .to(ExecutorServicePool.class)  
            .in(Singleton.class);  
    }  
  
    @Provides @Singleton Shortener provideShortener() {  
        return new TinyUrlShortener();  
    }  
}
```

Defining Injections



photo courtesy of <http://flickr.com/photos/gaetanlee/631004864>

Constructor injection to supply dependencies when creating an object

```
public class TweetClient {  
    private final Shortener shortener;  
    private final Tweeter tweeter;
```

@Inject

```
public TweetClient(Shortener shortener, Tweeter tweeter) {  
    this.shortener = shortener;  
    this.tweeter = tweeter;  
}
```

Constructor injection to supply dependencies when creating an object

```
public class TweetClient {  
    private final Shortener shortener;  
    private final Tweeter tweeter;
```



Immutable

@Inject

```
public TweetClient(Shortener shortener, Tweeter tweeter) {  
    this.shortener = shortener;  
    this.tweeter = tweeter;  
}
```

Method injection

sets dependencies into a new or existing instance

```
public class TweetClient {  
    private Shortener shortener;  
    private Tweeter tweeter;  
  
    @Inject void setShortener(Shortener shortener) {  
        this.shortener = shortener;  
    }  
  
    @Inject void setTweeter(Tweeter tweeter) {  
        this.tweeter = tweeter;  
    }  
}
```

- > Plays nice with inheritance

Field injection

sets dependencies into a new or existing instance

```
public class TweetClient {  
  
    @Inject Shortener shortener;  
    @Inject Tweeter tweeter;  
  
    public TweetClient() {}
```

- > Concise, but difficult to test

Injecting Providers



photo courtesy of <http://flickr.com/photos/toasty/535851893>

The Provider interface

```
public interface Provider<T> {  
    T get();  
}
```

Injecting a Provider

```
public class TweetClient {  
    @Inject Provider<Shortener> shortenerProvider;  
    @Inject Tweeter tweeter;  
  
    public void postButtonClicked() {  
        String text = textField.getText();  
        if (text.length() > 140) {  
            Shortener shortener = shortenerProvider.get();  
            text = shortener.shorten(text);  
        }  
        ...  
    }  
}
```

Why inject Providers?

- > **to load lazily**
- > **to get multiple instances**
 - for example, if you need multiple DatabaseConnections
- > **to mix scopes**
 - for example, to access request-scoped objects from a singleton-scoped object

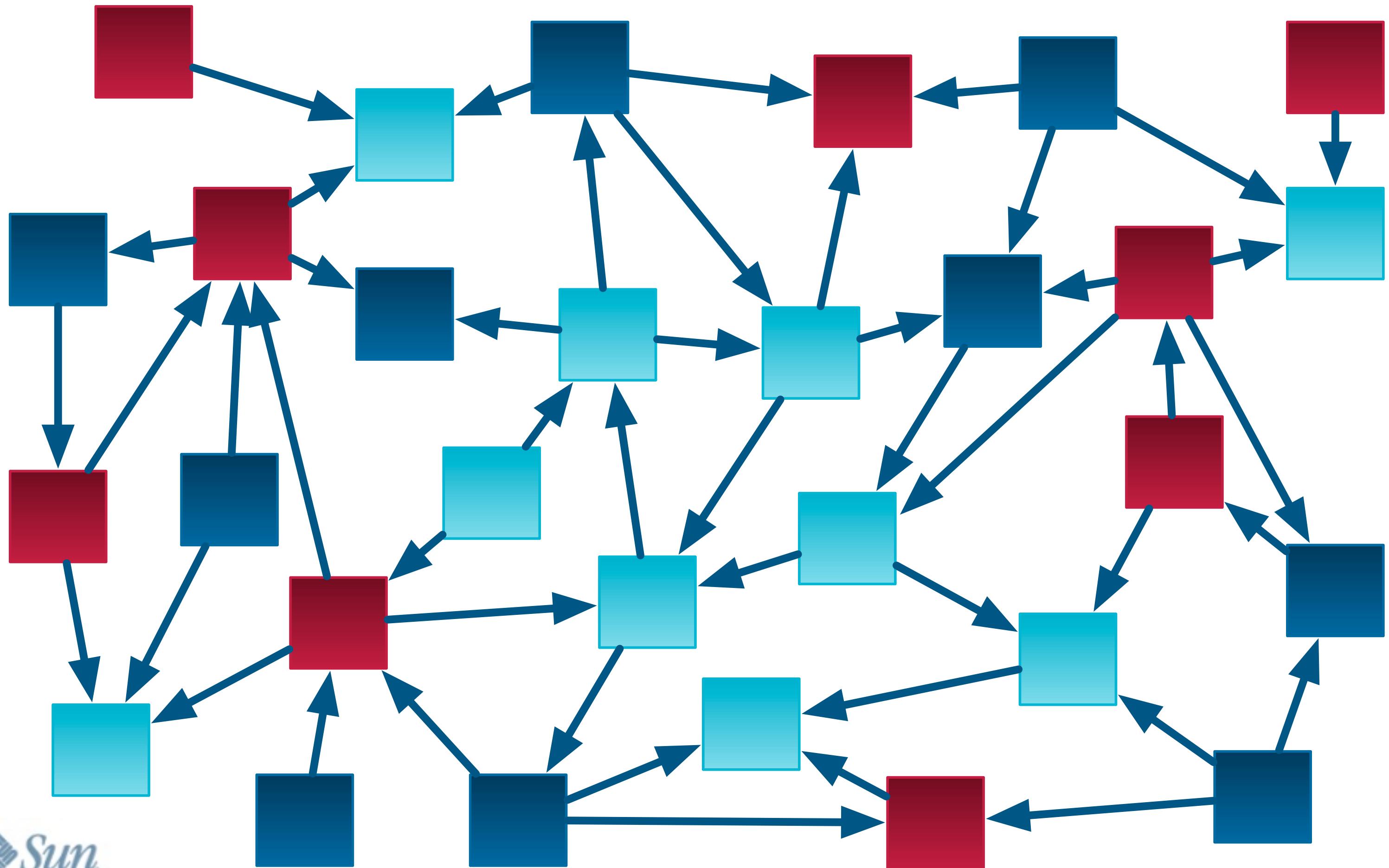
Leveraging Guice



- > Modularity
- > Type Literals
- > AJAX via GWT
- > Servlets
- > AOP
- > Introspection SPI

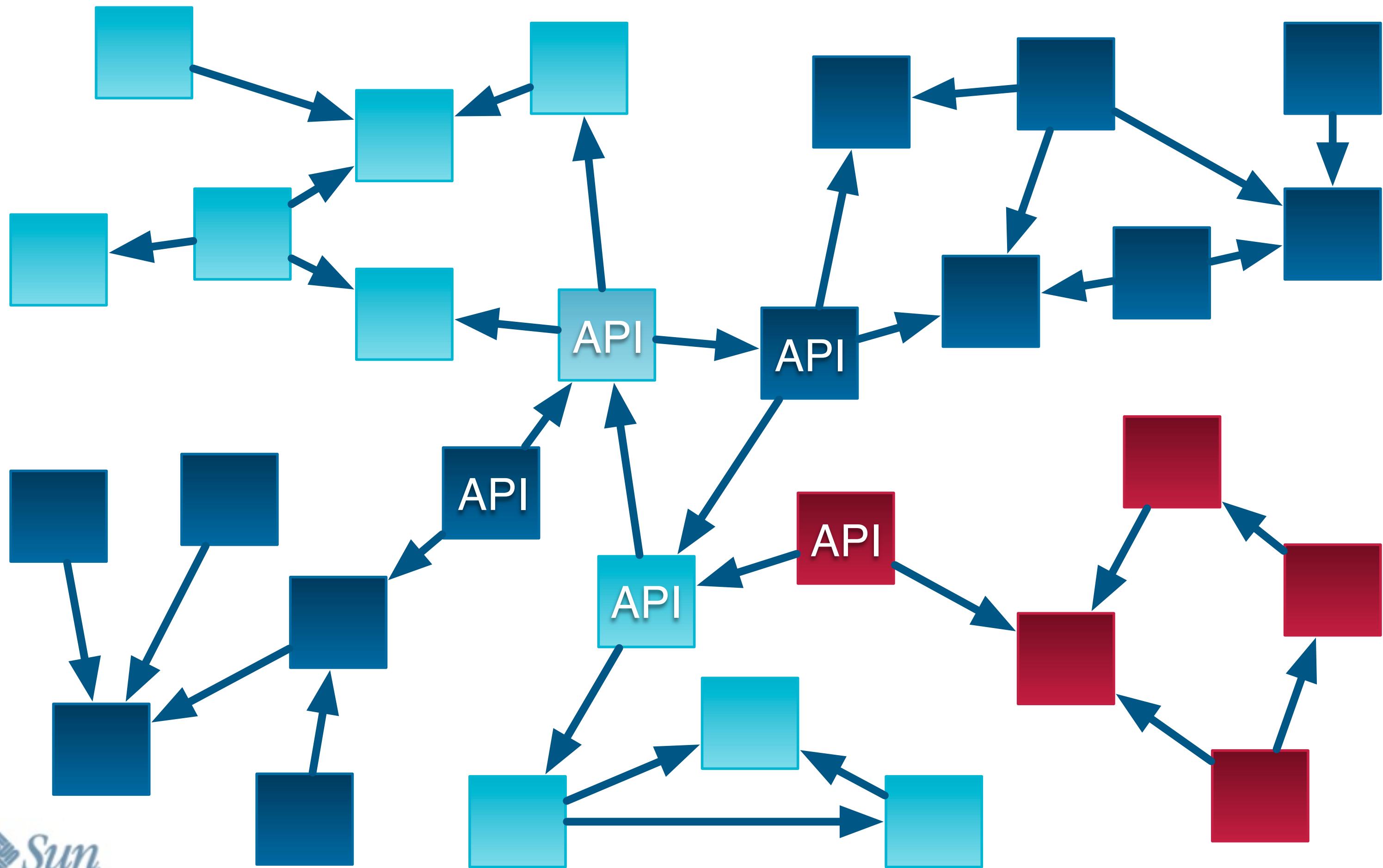
Modularity

Separate implementation from interface



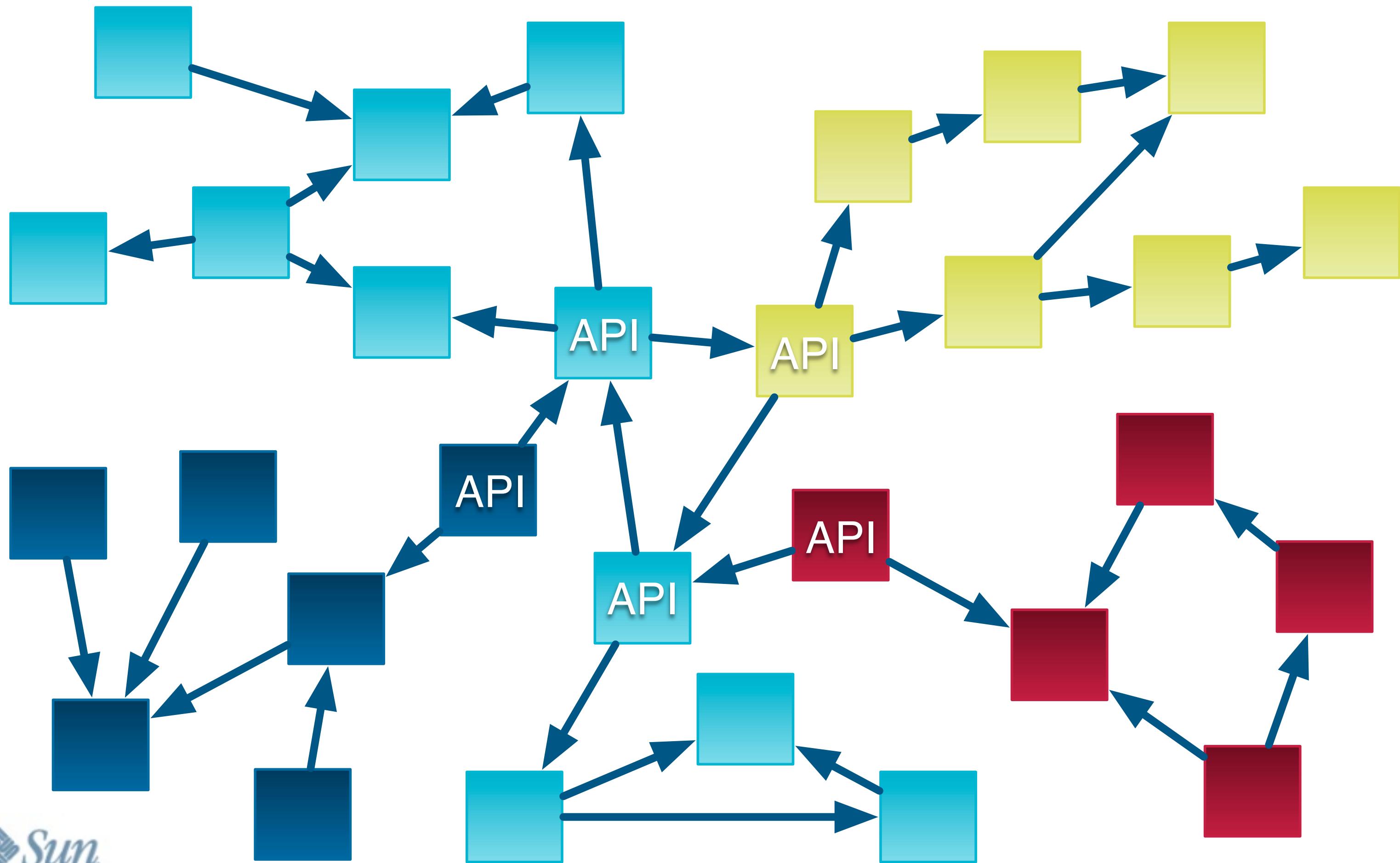
Modularity

Separate implementation from interface



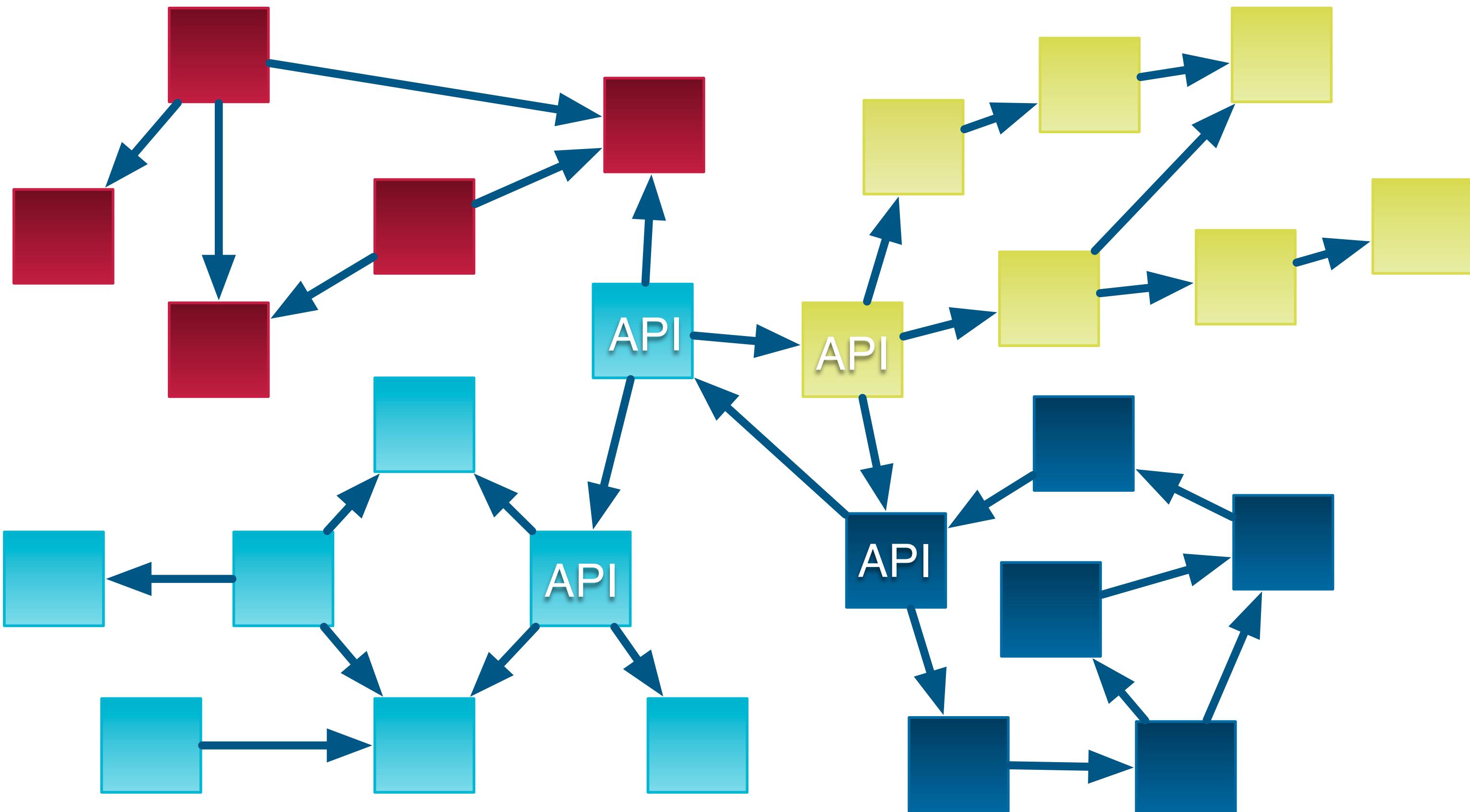
Modularity

Separate implementation from interface



Modularity

Separate implementation from interface



Type Literals

- > Like the language it's built on, Guice loves types
 - `Set<User>` is distinct from `Set<Tweet>`

```
public void configure() {  
    bind(new TypeLiteral<Set<User>>() {})  
        .toProvider(FollowersProvider.class);  
}
```



```
@Inject  
public SocialView(  
    Set<User> followers,  
    Set<Tweet> tweets) {
```

...

}

AJAX

Today's web apps are complex

- > **Testable**
 - across browsers
 - without browsers

- > **Modular**
 - reuse code across applications
 - and across platforms

AJAX

Today's web apps are complex



- > **Testable**
 - across browsers
 - without browsers

- > **Modular**
 - reuse code across applications
 - and across platforms

AJAX via GWT and GIN



- > Zero runtime cost: GIN generates JavaScript from modules
- > API compatibility with Guice
 - Great for GWT-RPC
 - Test without a browser

```
public class MyWidgetClientModule extends AbstractGinModule {  
    protected void configure() {  
        bind(MyWidgetMainPanel.class).in(Singleton.class);  
        bind(MyRemoteService.class)  
            .toProvider(MyRemoteServiceProvider.class);  
    }  
}
```



Servlets

- > Configure your servlets programmatically
- > Use @RequestScoped and @SessionScoped to manage application state safely and easily



jetty://



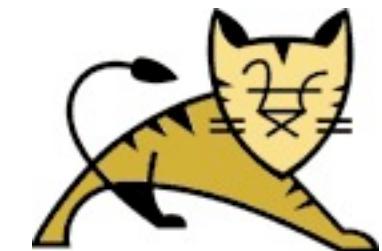
Servlets

- > Configure your servlets programmatically
- > Use @RequestScoped and @SessionScoped to manage application state safely and easily

```
public class TweetSearchServletModule extends ServletModule {  
    protected void configureServlets() {  
        serve("/search").with(TweetSearchServlet.class);  
    }  
}
```



jetty://



AOP aspect oriented programming

- > You can apply method interceptors to injected objects
 - This is fantastic for cross-cutting concerns like transactions, security, and performance

AOP

aspect oriented programming



- > You can apply method interceptors to injected objects
 - This is fantastic for cross-cutting concerns like transactions, security, and performance

```
public class DatabaseTweetStorage implements TweetStorage {  
  
    @Transactional  
    public void saveTweet(String message) {  
        ...  
    }  
}
```



<http://www.wideplay.com>

Introspection SPI service provider interface

- > Module and injector internals are available via a mirror SPI
 - Inspect, analyze and rewrite bindings

Introspection SPI service provider interface

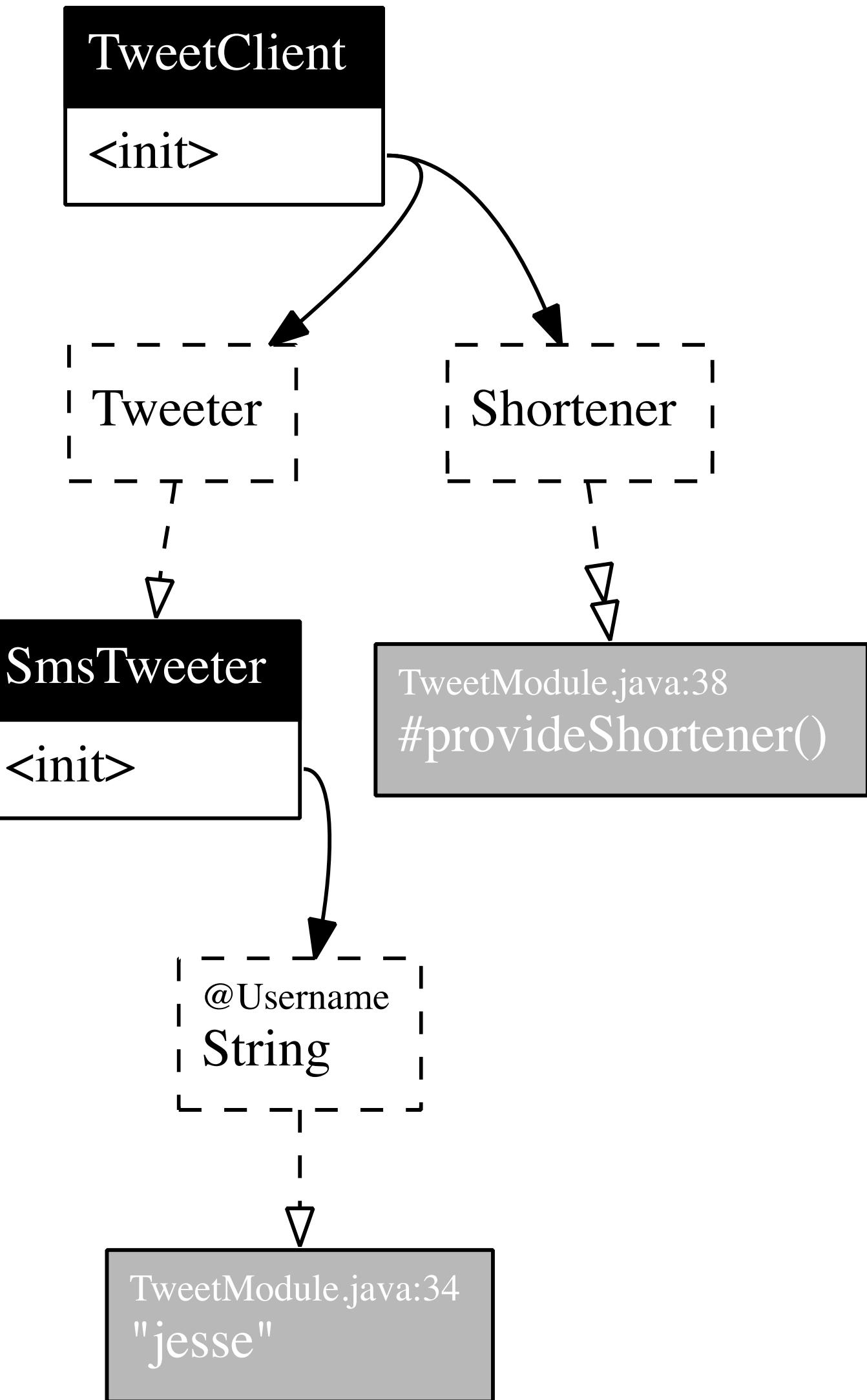
Elements	Bindings	Model
<ul style="list-style-type: none">• Binding• InjectionRequest• StaticInjectionRequest• MembersInjectorLookup• ProviderLookup• Message• PrivateElements• TypeConverterBinding• TypeListenerBinding• ScopeBinding• InterceptorBinding	<ul style="list-style-type: none">• ConstructorBinding• ConvertedConstantBinding• InstanceBinding• LinkedKeyBinding• ProviderInstanceBinding• ProviderKeyBinding• ProviderBinding• UntargettedBinding• ExposedBinding	<ul style="list-style-type: none">• InjectionPoint• Dependency• HasDependencies• ProviderWithDependencies• TypeConverter• TypeEncounter• TypeListener• InjectionListener
		Visitors <ul style="list-style-type: none">• ElementVisitor• BindingScopingVisitor• BindingTargetVisitor

Introspection SPI service provider interface

- > Guice uses it internally for...
 - `createInjector()`
 - module overrides
 - graphing

Introspection SPI service provider interface

- > Guice uses it internally for...
 - `createInjector()`
 - module overrides
 - graphing



Wrapping up...

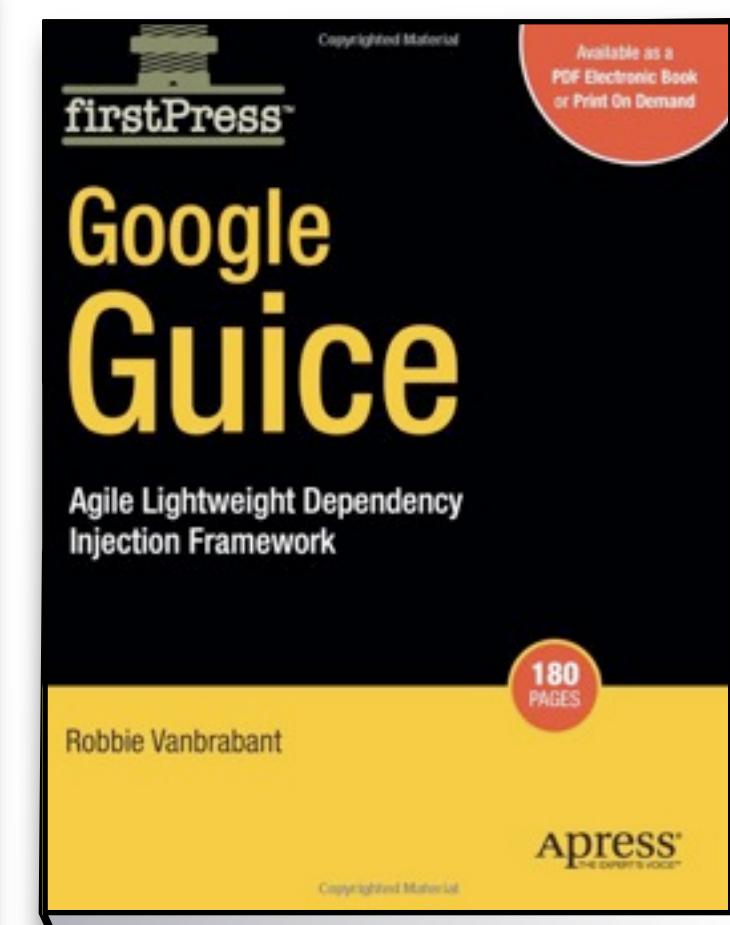
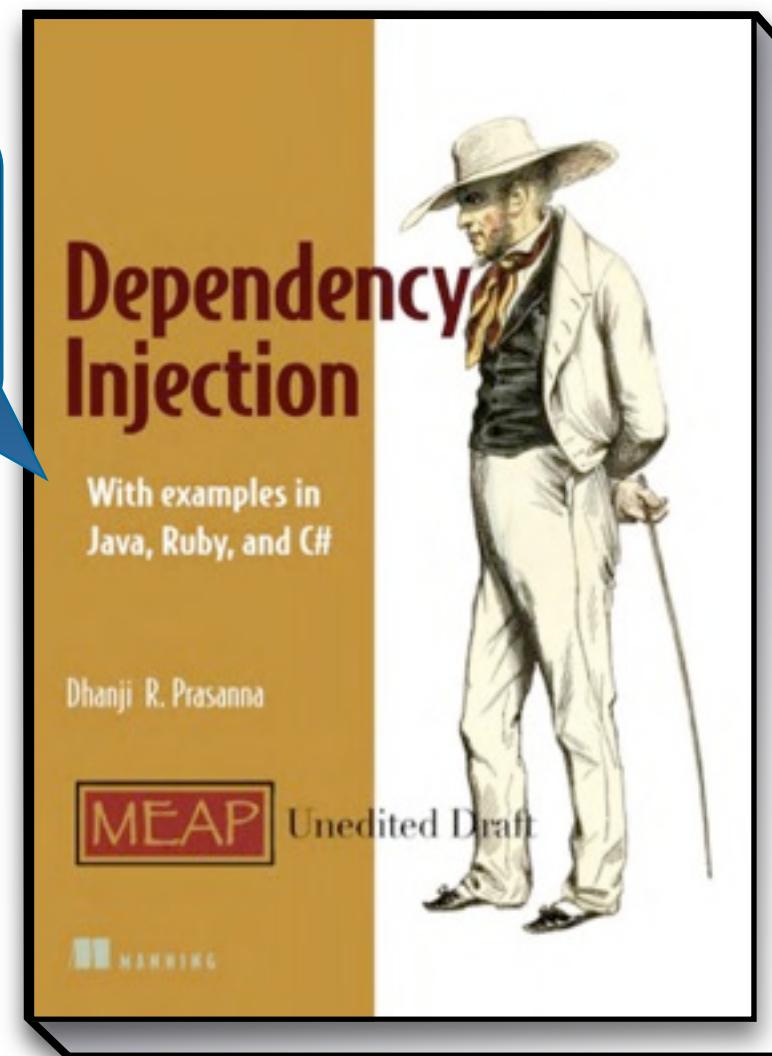
- > Dependency injection leads to testable and reusable code
- > Guice makes dependency injection easy
 - plus it enables scopes
 - and it integrates neatly with the other APIs you use
- > It works on both Java™ SE and Java™ EE
 - Plus Android, App Engine and GWT (via GIN)

For more information...

- > Usage, extensions, and best practices

<http://code.google.com/p/google-guice/>

depinj40
manning.com/prasanna



JavaOne09
apress.com/

Questions?

More from Google at JavaOne...

- > Building Enterprise Java™ Technology-Based Web Apps with Google Open-Source Technology
 - Session TS-4062, coming up at 16:10



JavaOneSM

Thank You

Jesse Wilson

<http://code.google.com/p/google-guice/>

