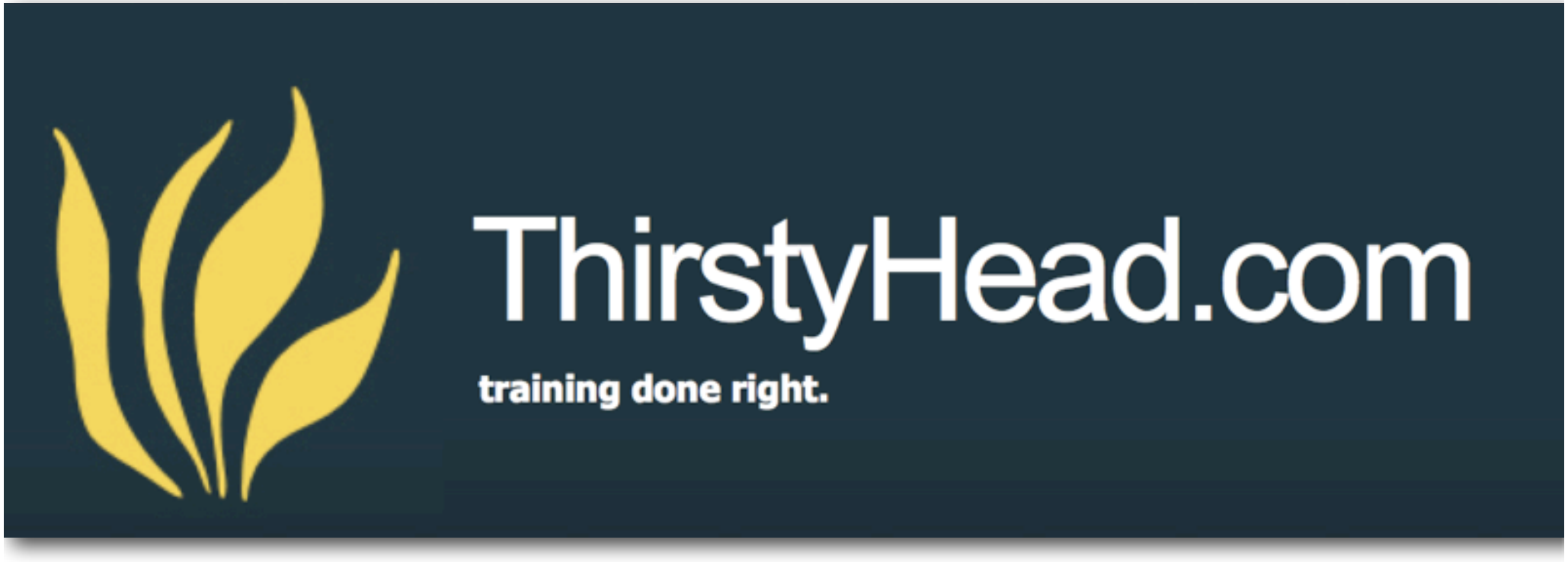
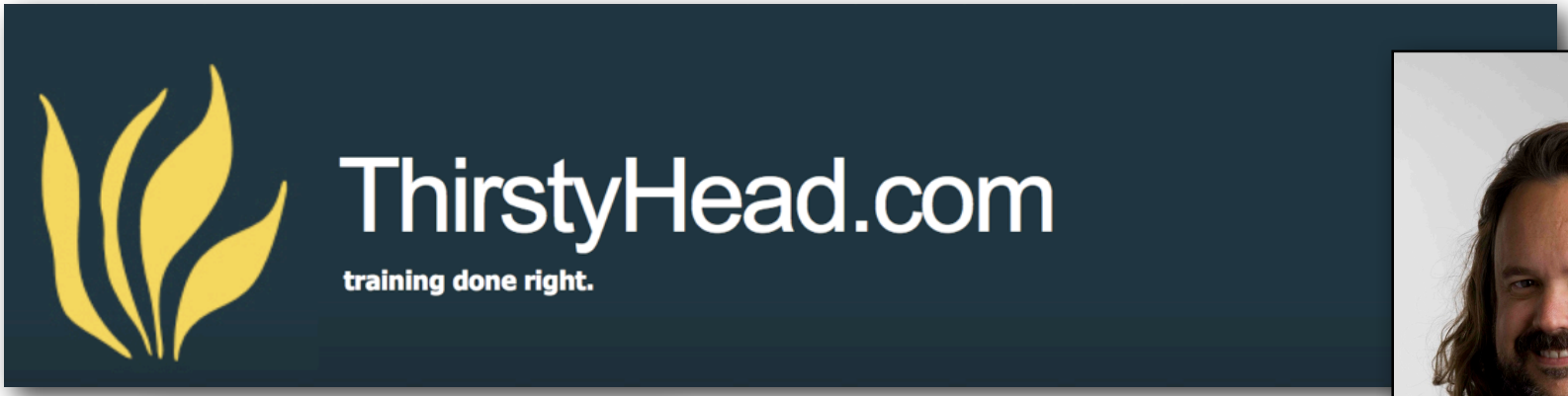
Compile-time and Runtime Metaprogramming with Groovy

Scott Davis, ThirstyHead.com


ThirstyHead.com
training done right.

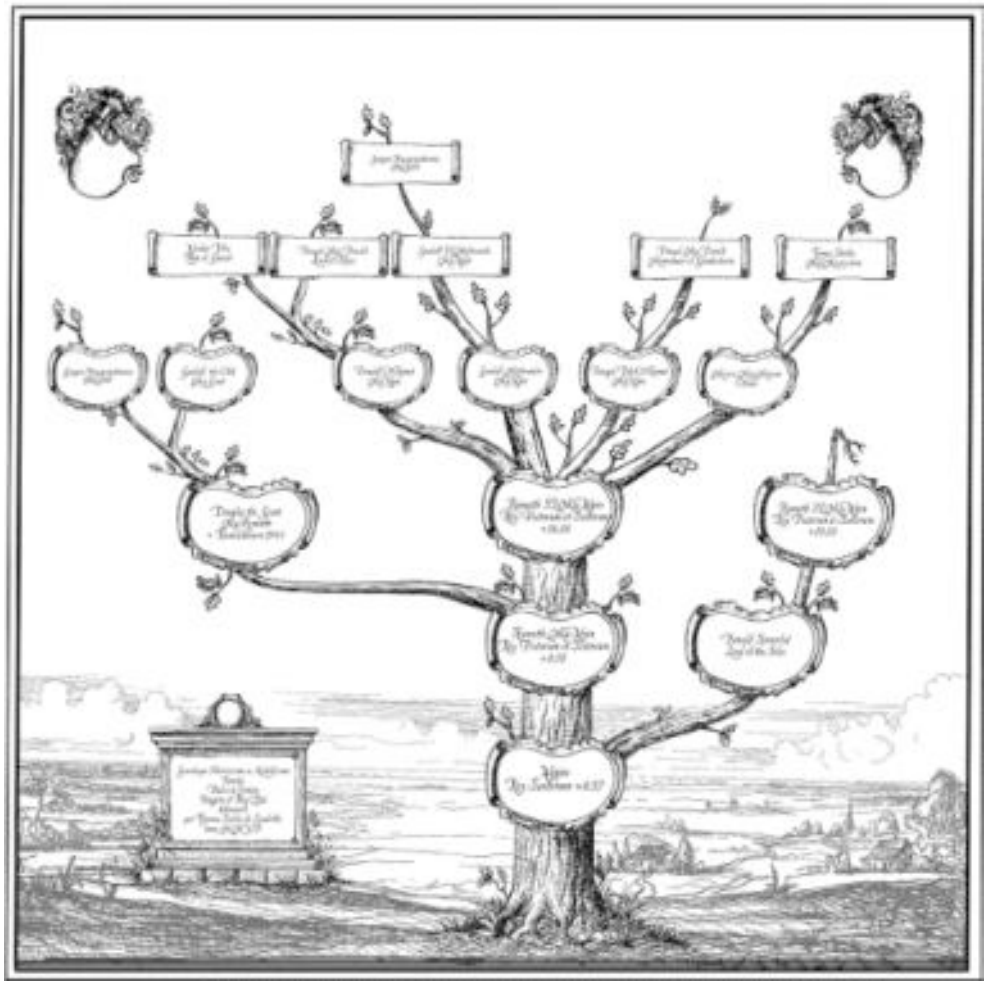

ThirstyHead.com
training done right.


Scott Davis
@scottdavis99


The Pragmatic Programmers
GIS For Web Developers
Adding Where to Your Web Applications
The Pragmatic Programmers
Groovy Recipes
Greasing the Wheels of Java
Scott Davis
Jason Rudolph
Getting Started with Grails Second Edition
A modern web framework for the Java platform
ENTERPRISE SOFTWARE DEVELOPMENT SERIES
InfoQ


HTML

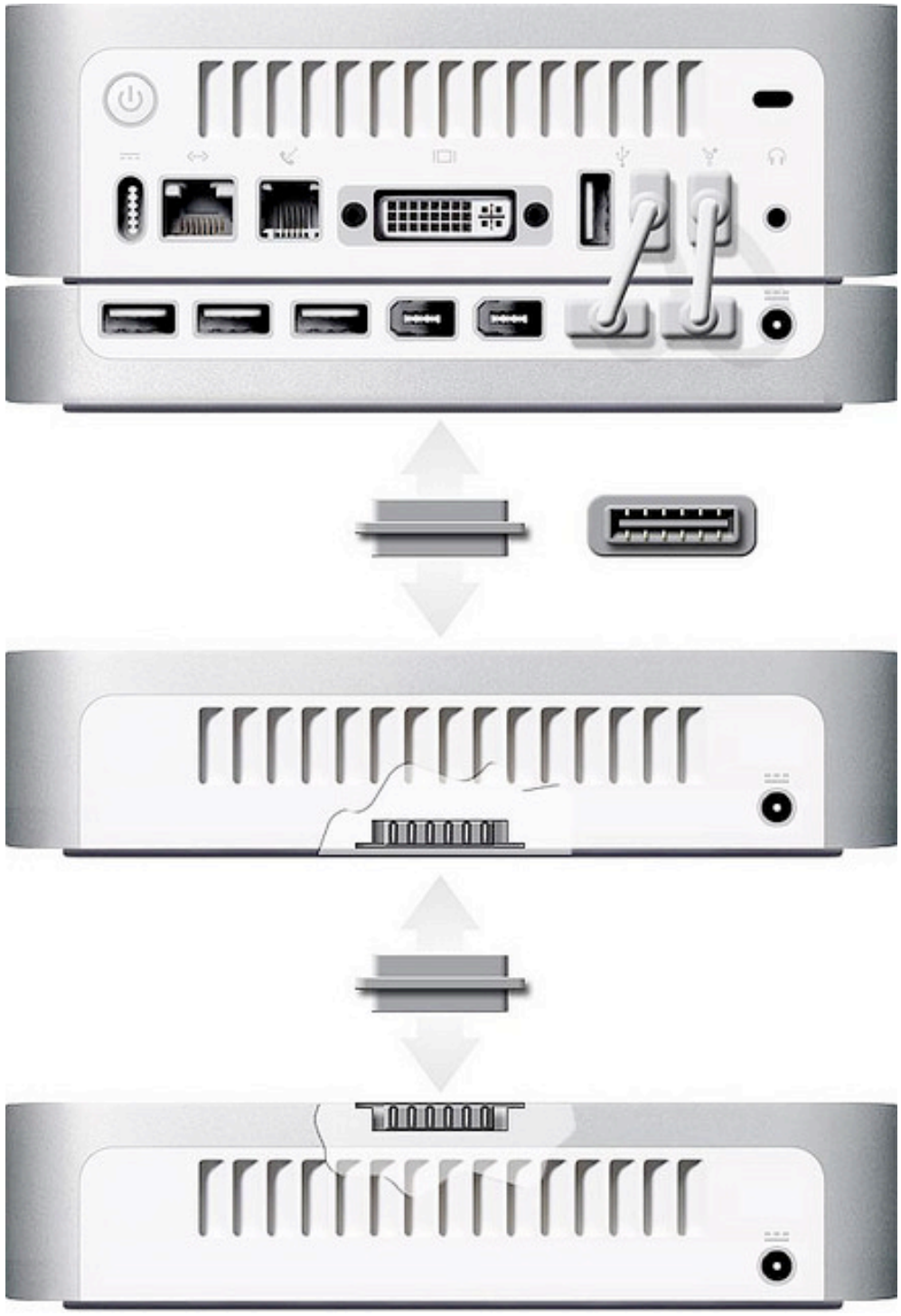

Java™



inheritance


Groovy

metaprogramming



Runtime

Categories
ExpandoMetaClass
MethodMissing

Compile-time

AST Transformations

Categories

(Runtime)

Objective-C

From Wikipedia, the free encyclopedia

Objective-C is a reflective, object-oriented programming language that adds Smalltalk-style messaging to the C programming language.

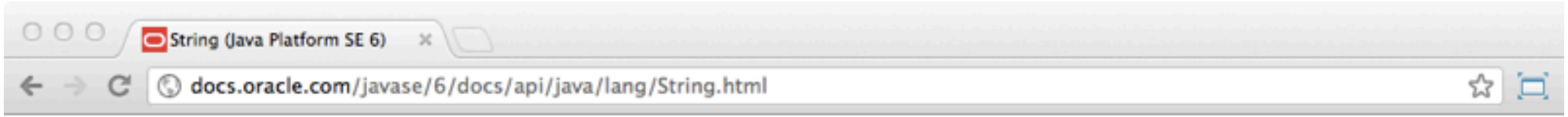
Today, it is used primarily on Apple's Mac OS X and iOS: two environments based on the OpenStep standard, though not compliant with it.[2] Objective-C is the primary language used for Apple's Cocoa API, and it was originally the main language on NeXT's NeXTSTEP OS. Generic Objective-C programs that do not use these libraries can also be compiled for any system supported by gcc or Clang.

Categories [edit]

During the design of Objective-C, one of the main concerns was the maintainability of large code bases. Experience from the structured programming world had shown that one of the main ways to improve code was to break it down into smaller pieces. Objective-C borrowed and extended the concept of *categories* from Smalltalk implementations to help with this process.[7]

A category collects method implementations into separate files. The programmer can place groups of related methods into a category to make them more readable. For instance, one could create a "SpellChecking" category in the String object, collecting all of the methods related to spell checking into a single place.

Furthermore, the methods within a category are added to a class at runtime. Thus, categories permit the programmer to add methods to an existing class without the need to recompile that class or even have access to its source code. For example, if a system does not contain a spell checker in its String implementation, it could be added without modifying the String source code.



java.lang

Class String

java.lang.Object
  └ **java.lang.String**

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

NOTE: Evolutionary dead-end in Java!

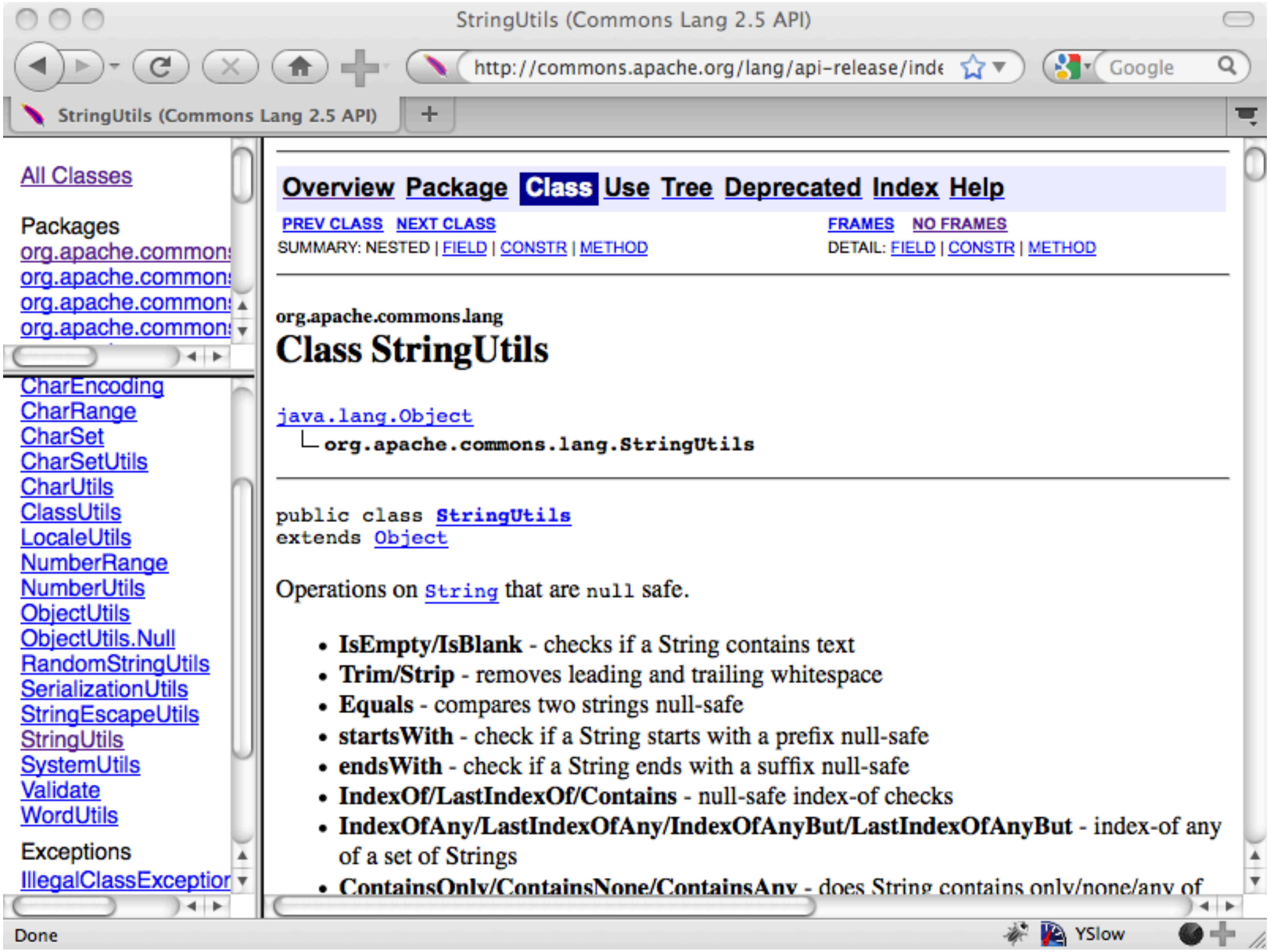
The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:


Lang – Home
http://commons.apache.org/lang/
Google
Lang – Home
Apache Commons
http://commons.apache.org/
commons
lang
Last Published: 01 August 2010 | Version: 3.0-SNAPSHOT
ApacheCon | Apache | Commons
ApacheCon
North America 2010
1-5 November
Atlanta, GA
Lang
Overview
Download
Users guide
Release History
Javadoc (2.5 release)
Development
Building
Mailing Lists
Issue Tracking
Proposal
Developer guide
Source Repository
Javadoc (SVN latest)
Project Documentation
Project Information
About
Commons Lang
The standard Java libraries fail to provide enough methods for manipulation of its core classes. Apache Commons Lang provides these extra methods.
Lang provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode, toString and equals.
Documentation
A getting started user guide is available together with various project reports.
The JavaDoc API documents are available online:
The beta release 3.0-beta
The current stable release 2.5
The previous version 2.4
Older releases - see the Release History page
The subversion repository can be browsed.
Done
YSlow

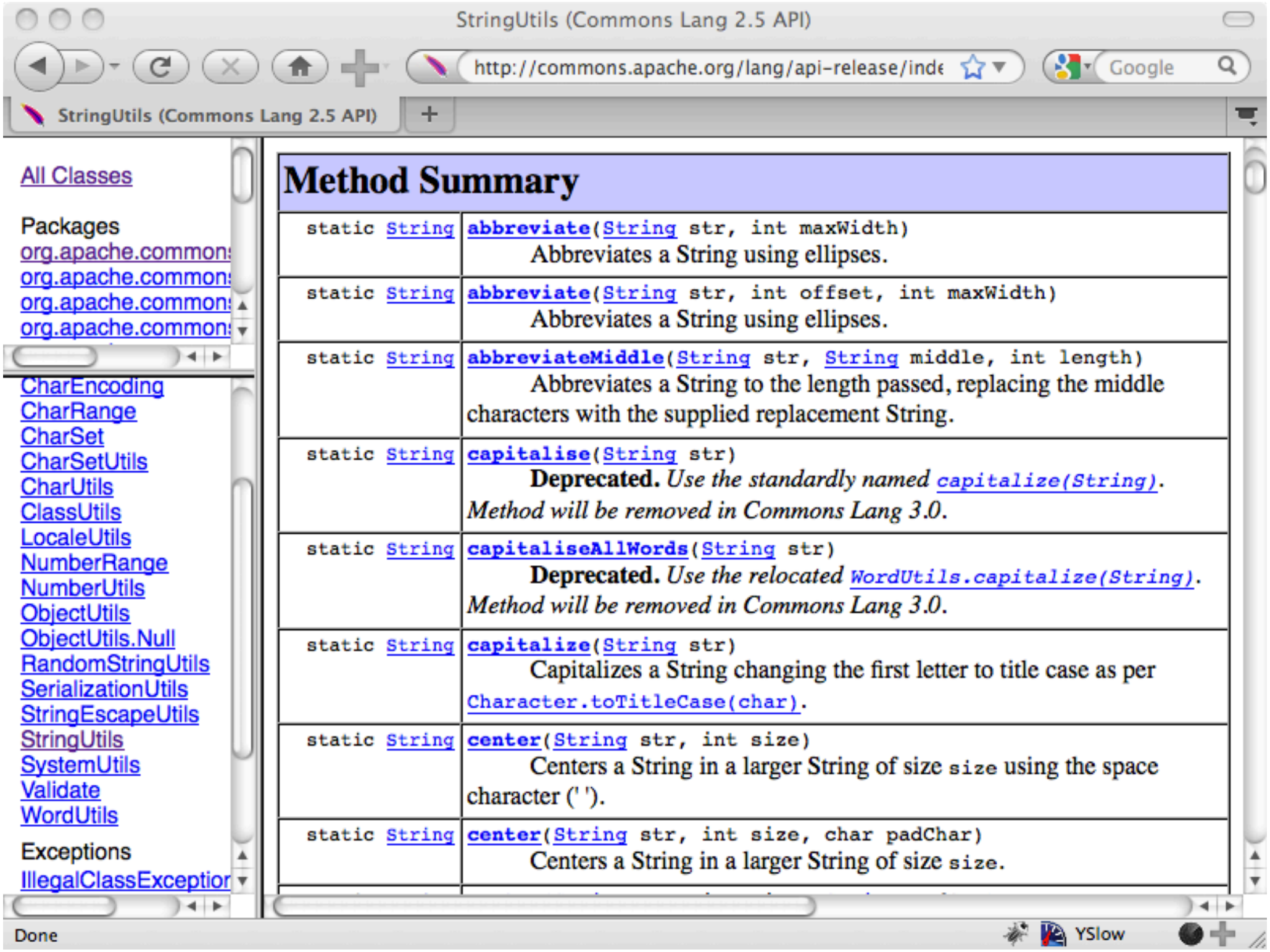

StringUtils (Commons Lang 2.5 API)
http://commons.apache.org/lang/api-release/inde
Google
StringUtils (Commons Lang 2.5 API)
All Classes
Packages
org.apache.commons
org.apache.commons
org.apache.commons
org.apache.commons
CharEncoding
CharRange
CharSet
CharSetUtils
CharUtils
ClassUtils
LocaleUtils
NumberRange
NumberUtils
ObjectUtils
ObjectUtils.Null
RandomStringUtils
SerializationUtils
StringEscapeUtils
StringUtils
SystemUtils
Validate
WordUtils
Exceptions
IllegalClassExceptior
Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
org.apache.commons.lang
Class StringUtils
java.lang.Object
org.apache.commons.lang.StringUtils
public class StringUtils
extends Object
Operations on String that are null safe.
IsEmpty/IsBlank - checks if a String contains text
Trim/Strip - removes leading and trailing whitespace
Equals - compares two strings null-safe
startsWith - check if a String starts with a prefix null-safe
endsWith - check if a String ends with a suffix null-safe
IndexOf/LastIndexOf/Contains - null-safe index-of checks
IndexOfAny/LastIndexOfAny/IndexOfAnyBut/LastIndexOfAnyBut - index-of any of a set of Strings
ContainsOnly/ContainsNone/ContainsAny - does String contains only/none/any of
Done
YSlow

Category metaprogramming works with any library (Groovy, Java, or otherwise) as long as:

1. The method is static

2. The first argument of the method is the same data-type as the "delegate" class

```
java.lang.String (delegate)

StringUtils:
static String abbreviate(String str, int maxWidth)
```


StringUtils (Commons Lang 2.5 API)
http://commons.apache.org/lang/api-release/inde
Google
StringUtils (Commons Lang 2.5 API)
All Classes
Packages
org.apache.commons
org.apache.commons
org.apache.commons
org.apache.commons
CharEncoding
CharRange
CharSet
CharSetUtils
CharUtils
ClassUtils
LocaleUtils
NumberRange
NumberUtils
ObjectUtils
ObjectUtils.Null
RandomStringUtils
SerializationUtils
StringEscapeUtils
StringUtils
SystemUtils
Validate
WordUtils
Exceptions
IllegalClassExceptior
Method Summary
static String abbreviate(String str, int maxWidth)
Abbreviates a String using ellipses.
static String abbreviate(String str, int offset, int maxWidth)
Abbreviates a String using ellipses.
static String abbreviateMiddle(String str, String middle, int length)
Abbreviates a String to the length passed, replacing the middle characters with the supplied replacement String.
static String capitalise(String str)
Deprecated. Use the standardly named capitalize(String). Method will be removed in Commons Lang 3.0.
static String capitaliseAllWords(String str)
Deprecated. Use the relocated WordUtils.capitalize(String). Method will be removed in Commons Lang 3.0.
static String capitalize(String str)
Capitalizes a String changing the first letter to title case as per Character.toTitleCase(char).
static String center(String str, int size)
Centers a String in a larger String of size size using the space character (' ').
static String center(String str, int size, char padChar)
Centers a String in a larger String of size size.
Done
YSlow

Java:

```
import org.apache.commons.lang.*;

public class TestJava{
  public static void main(String[] args){
    String sentence = "My name is Inigo Montoya.";
    System.out.println(sentence);
    System.out.println(StringUtils.abbreviate(sentence, 15));
  }
}
```

```
$ ./runJava.sh

My name is Inigo Montoya.
My name is I...
```

Groovy:

```
import org.apache.commons.lang.*

use(StringUtils){
  def sentence = "My name is Inigo Montoya."
  println sentence

  //NOTE: look at the cool new method
  //      on java.lang.String!!!
  println sentence.abbreviate(15)
}
```

```
$ ./runGroovy.sh

My name is Inigo Montoya.
My name is I...
```

ExpandoMetaClass

(Runtime)

ExpandoMetaClass

Using ExpandoMetaClass to add behaviour

Groovy 1.1 includes a special MetaClass called an ExpandoMetaClass that allows you to dynamically add methods, constructors, properties and static methods using a neat closure syntax.

How does it work? Well every java.lang.Class is supplied with a special "metaClass" property that when used will give you a reference to an ExpandoMetaClass instance.

Groovy (class-level metaprogramming):

```
def message = "I love groovy"

String.metaClass.shout = {
  return delegate.toUpperCase()
}

println message.shout()
```

```
$ groovy shout

I LOVE GROOVY
```

Groovy (instance-level metaprogramming):

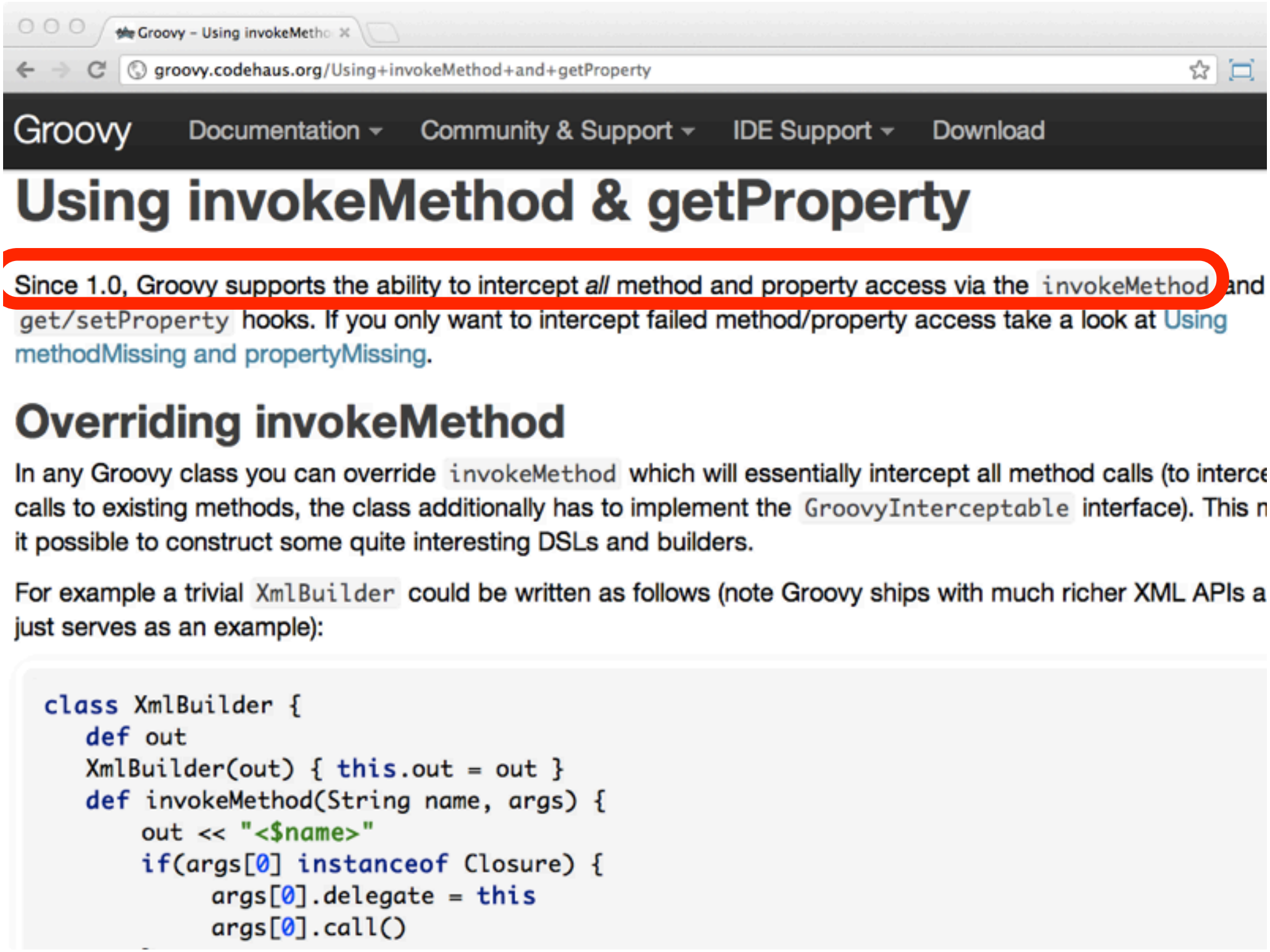
```
def message = "I love groovy"

message.metaClass.shout = {
  return delegate.toUpperCase()
}

println message.shout()

"java is ok, too".shout()
```

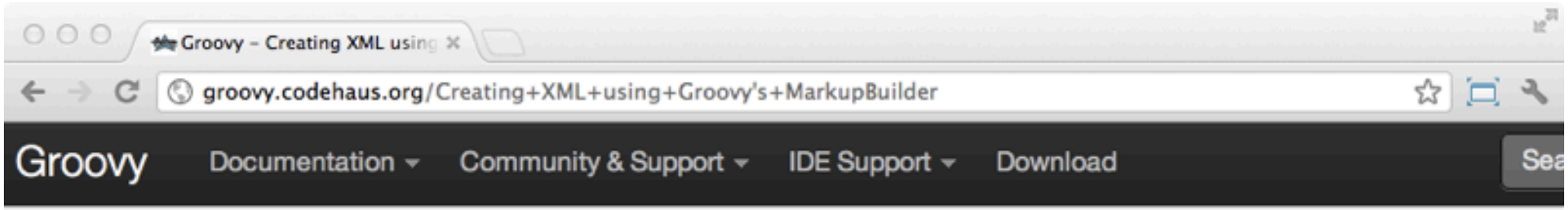
```
$ groovy shout

I LOVE GROOVY
Caught: groovy.lang.MissingMethodException: No
signature of method: java.lang.String.shout() is
applicable for argument types: () values: []
```

MethodMissing

(Runtime)

invokeMethod()


Groovy – Using invokeMetho
groovy.codehaus.org/Using+invokeMethod+and+getProperty
Groovy
Documentation
Community & Support
IDE Support
Download
Using invokeMethod & getProperty
Since 1.0, Groovy supports the ability to intercept all method and property access via the invokeMethod and get/setProperty hooks. If you only want to intercept failed method/property access take a look at Using methodMissing and propertyMissing.
Overriding invokeMethod
In any Groovy class you can override invokeMethod which will essentially intercept all method calls (to interce
calls to existing methods, the class additionally has to implement the GroovyInterceptable interface). This m
it possible to construct some quite interesting DSLs and builders.
For example a trivial XmlBuilder could be written as follows (note Groovy ships with much richer XML APIs a
just serves as an example):
class XmlBuilder {
def out
XmlBuilder(out) { this.out = out }
def invokeMethod(String name, args) {
out << "<$name>"
if(args[0] instanceof Closure) {
args[0].delegate = this
args[0].call()



Simple Example

Here is an example of using Groovy's `MarkupBuilder` to create a new XML file:

```
// require(groupId:'xmlunit', artifactId:'xmlunit', version:'1.0')
import groovy.xml.MarkupBuilder
import org.custommonkey.xmlunit.*

def writer = new StringWriter()
def xml = new MarkupBuilder(writer)
xml.records() {
  car(name:'HSV Maloo', make:'Holden', year:2006) {
    country('Australia')
    record(type:'speed', 'Production Pickup Truck with speed of 271kph')
  }
  car(name:'P50', make:'Peel', year:1962) {
    country('Isle of Man')
    record(type:'size', 'Smallest Street-Legal Car at 99cm wide and 59 kg in weight')
  }
  car(name:'Royale', make:'Bugatti', year:1931) {
    country('France')
    record(type:'price', 'Most Valuable Car at $15 million')
  }
}

XMLUnit.setIgnoreWhitespace(true)
def xmlDiff = new Diff(writer.toString(), XmlExamples.CAR_RECORDS)
```

We have used XMLUnit to compare the XML we created with our sample XML. To do this, make sure the sample XML is available, i.e. that the following class is added to your CLASSPATH:

XmlExamples.groovy

```
class XmlExamples {
  static def CAR_RECORDS = '''
    <records>
      <car name='HSV Maloo' make='Holden' year='2006'>
        <country>Australia</country>
        <record type='speed'>Production Pickup Truck with speed of 271kph</record>
      </car>
      <car name='P50' make='Peel' year='1962'>
        <country>Isle of Man</country>
        <record type='size'>Smallest Street-Legal Car at 99cm wide and 59 kg in
weight</record>
      </car>
      <car name='Royale' make='Bugatti' year='1931'>
        <country>France</country>
        <record type='price'>Most Valuable Car at $15 million</record>
      </car>
    </records>
  '''
}
```

Wednesday, April 27, 2011

Groovy Goodness: Build JSON with JsonBuilder and Pretty Print JSON Text

Groovy 1.8 adds JSON support. We can build a JSON data structure with the `JsonBuilder` class. This class functions just like other builder classes. We define a hiearchy with values and this is converted to JSON output when we view the String value. We notice the syntax is the same as for a `MarkupBuilder`.

```
00. import groovy.json.*
01.
02. def json = new JsonBuilder()
03.
04. json.message {
05.     header {
06.         from('mrhaki')  // parenthesis are optional
07.         to 'Groovy Users', 'Java Users'
08.     }
09.     body "Check out Groovy's gr8 JSON support."
10. }
11.
12. assert json.toString() == '{"message":{"header":
    ➲ {"from":"mrhaki","to":["Groovy Users","Java
       Users"]},"body":"Check out Groovy\'s gr8 JSON
       support."}}'
13.
14. // We can even pretty print the JSON output
15. def prettyJson = JsonOutput.prettyPrint(json.toString())
16. assert prettyJson == '''{
17.     "message": {
18.         "header": {
19.             "from": "mrhaki",
20.             "to": [
```

```
class Person{
  String name Map relationships = [:]
  Object invokeMethod(String what, Object who){
    if(relationships.containsKey(what)){
      who.each{thisPerson ->
        relationships.get(what).add(thisPerson)
      }
    } else{
      relationships.put(what,who as List)
    }
  }
}

def scott = new Person(name:"Scott")
scott.married "Kim"
scott.knows "Neal"
scott.workedWith "Brian"
scott.knows "Ted", "Ben", "David"
println scott.relationships
```

```
["married" :["Kim" ],
"knows":["Neal", "Venkat", "Ted", "Ben", "David"],
"workedWith":["Brian", "Jared"]]
```

methodMissing()

Using methodMissing & propertyMissing

Since 1.5, Groovy supports the concept of "methodMissing". This differs from invokeMethod in that it is only invoked in the case of failed method dispatch.

There are a couple of important aspects to this behaviour:

1. Since method/propertyMissing only occur in the case of failed dispatch, they are expensive to execute
2. Since method/propertyMissing aren't intercepting EVERY method call like invokeMethod they can be more efficient with a few meta-programming tricks

Using methodMissing with dynamic method registration

Typically when using methodMissing the code will react in some way that makes it possible for the next time the same method is called, that it goes through the regular Groovy method dispatch logic.

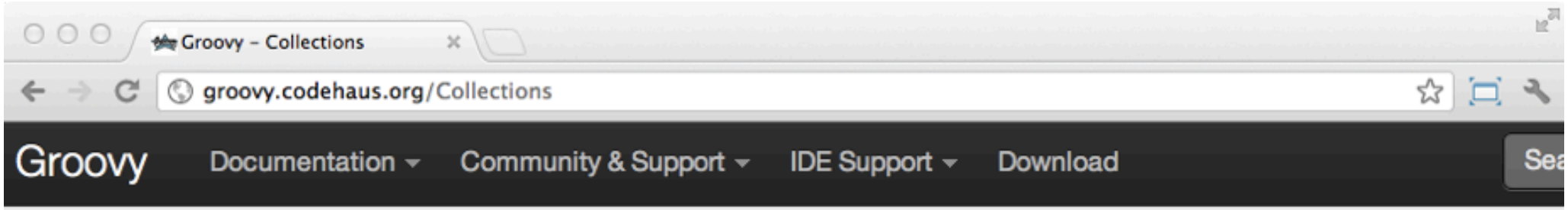
For example consider dynamic finders in GORM. These are implemented in terms of methodMissing. How does it work? The code resembles something like this:

```
class GORM {

   def dynamicMethods = [...] // an array of dynamic methods that use regex
   def methodMissing(String name, args) {
       def method = dynamicMethods.find { it.match(name) }
       if(method) {
          GORM.metaClass."$name" = { Object[] varArgs ->
             method.invoke(delegate, name, varArgs)
```

```
class Ipod {
  ...

  String spacify(String camelCase){
    String title = ""
    camelCase.each{
      char c = (char) it
      title += Character.isUpperCase(c) ? " ${c}" : "${c}"
    }
    title.trim()
  }

  def methodMissing(String methodCall, Object arg){
    if(methodCall.startsWith("play")){
      // "playTwoOfUs" --> "TwoOfUs"
      return play(spacify(methodCall - "play"))
    }
  }
}
```

```
ipod = new Ipod()
ipod << new Song(title:"Two Of Us", duration:217)
ipod << new Song(title:"Dig A Pony", duration:235)

println ipod.play("Dig A Pony")
println ipod.playTwoOfUs()
println ipod.playSomeOtherSong()
```

Expando



Dynamic objects (Expandos)

The Expando is not a collection in the strictest sense, but in some ways it is similar to a Map, or objects in JavaScript that do not have to have their properties defined in advance. It allows you to create dynamic objects by making use of Groovy's closure mechanisms. An Expando is different from a map in that you can provide synthetic methods that you can call on the object.

```
def player = new Expando()
player.name = "Dierk"
player.greeting = { "Hello, my name is $name" }

println player.greeting()
player.name = "Jochen"
println player.greeting()
```

The player.greeting assignment passes in a closure to execute when greeting() is called on the Expando. Notice that the closure has access to the properties assigned to the Expando, even though these values may change over time, using Groovy's GString "$variableOrProperty" notation.

```
def e = new Expando()
e.latitude = 70
e.longitude = 30
println e


e.areWeLost = {->
  return (e.longitude != 30) || (e.latitude != 70)
}

e.areWeLost()
//===> false
```

AST Transformations

(Compile-time)

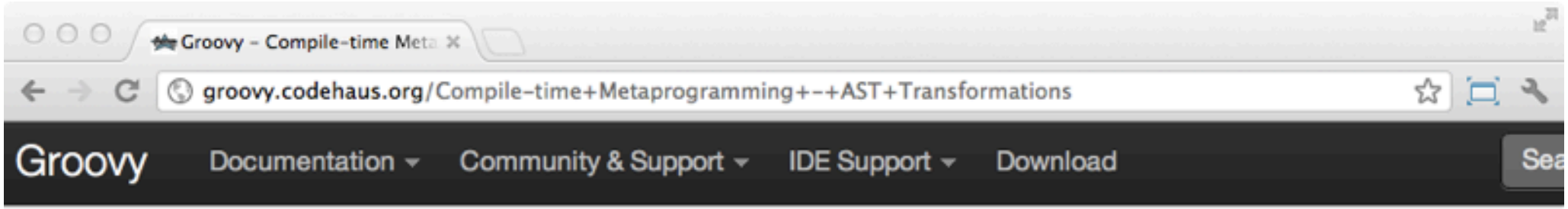
Compile-time Metaprogramming - AST Transformations

AST Transformations

Although at times, it may sound like a good idea to extend the syntax of Groovy to implement new features (like this is the case for instance for multiple assignments), most of the time, we can't just add a new keyword to the grammar, or create some new syntax construct to represent a new concept. However, with the idea of AST (Abstract Syntax Tree) Transformations, we are able to tackle new and innovative ideas without necessary grammar changes.

When the Groovy compiler compiles Groovy scripts and classes, at some point in the process, the source code will end up being represented in memory in the form of a Concrete Syntax Tree, then transformed into an Abstract Syntax Tree. The purpose of AST Transformations is to let developers hook into the compilation process to be able to modify the AST before it is turned into bytecode that will be run by the JVM.

AST Transformations provides Groovy with improved compile-time metaprogramming capabilities allowing powerful flexibility at the language level, without a runtime performance penalty.



One hook for accessing this capability is via annotations (for local AST transformations). In your Groovy code you will make use of one of more annotations. Behind the scenes, an AST processor relevant to the annotation you are using is inserted into the compiler phases at the appropriate point. You can explore some of the more popular Annotations below:

- Bindable and Vetoable transformation
- Building AST Guide
- Category and Mixin transformations
- Compiler Phase Guide
- Delegate transformation
- Immutable AST Macro
- Immutable transformation
- Lazy transformation
- Newify transformation
- PackageScope transformation
- Singleton transformation

Grape also provides its own transformation with @Grab.

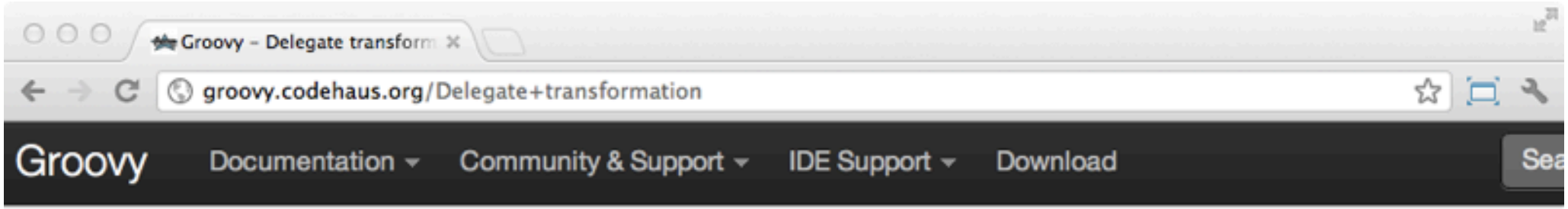
Implementing your own AST Transformations

There are two kinds of AST Transformations, local and global transformations:

- implementing a Global AST Transformations
- implementing a local AST transformation

When writing an AST Transformation, you may find the following guides helpful:

@Delegate



Java doesn't provide any built-in delegation mechanism, and so far Groovy didn't either. But with the @Delegate transformation, a class field or property can be annotated and become an object to which method calls are delegated. In the following example, an Event class has a date delegate, and the compiler will delegate all of Date's methods invoked on the Event class to the Date delegate. As shown in the latest assert, the Event class has got a before(Date) method, and all of Date's methods.

```
import java.text.SimpleDateFormat

class Event {
    @Delegate Date when
    String title, url
}

def df = new SimpleDateFormat("yyyy/MM/dd")

def gr8conf = new Event(title: "GR8 Conference",
                        url: "http://www.gr8conf.org",
                       when: df.parse("2009/05/18"))
def javaOne = new Event(title: "JavaOne",
                        url: "http://java.sun.com/javaone/",
                       when: df.parse("2009/06/02"))

assert gr8conf.before(javaOne.when)
```

NOTE: All of the Date field's methods are "pushed up" to the Event "parent" class

The Groovy compiler adds all of Date's methods to the Event class, and those methods simply delegate the call to the Date field. If the delegate is not a final class, it is even possible to make the Event class a subclass of Date simply by

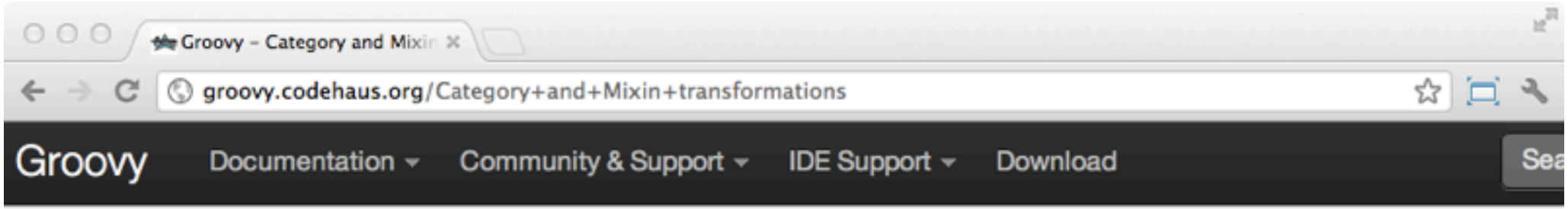
Suppose we wanted to create a new type of String: AllCapsString

We cannot extend java.lang.String because it is final.

But we can use the @Delegate AST transformation to delegate all "String" calls to AllCapsString...

```
class AllCapsString{
  @Delegate final String body

  AllCapsString(String body){
    this.body = body.toUpperCase()
  }

  String toString(){
    body
  }
}
```

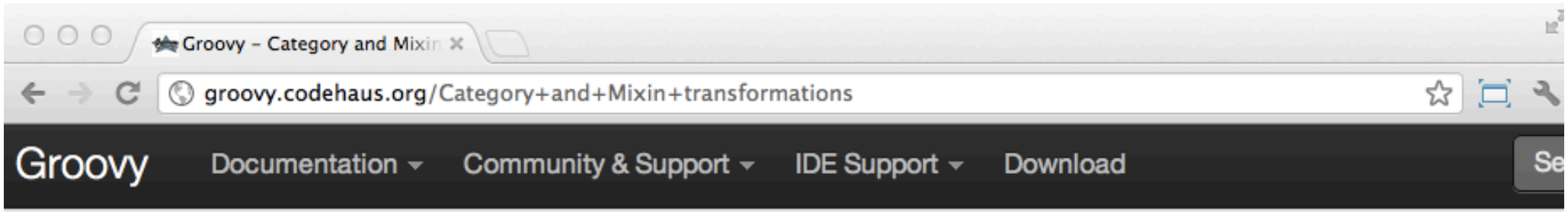
@Category



If you've been using Groovy for a while, you're certainly familiar with the concept of Categories. It's a mechanism to extend existing types (even final classes from the JDK or third-party libraries), to add new methods to them. This is also a technique which can be used when writing Domain-Specific Languages. Let's consider the example below:

```
final class Distance {
    def number
    String toString() { "${number}m" }
}

class NumberCategory {
    static Distance getMeters(Number self) {
        new Distance(number: self)
    }
}

use(NumberCategory) {
    def dist = 300.meters

    assert dist instanceof Distance
    assert dist.toString() == "300m"
}
```

We have a simplistic and fictive Distance class which may have been provided by a third-party, who had the bad idea of making the class final so that nobody could ever extend it in any way. But thanks to a Groovy Category, we are able to decorate the Distance type with additional methods. Here, we're going to add a getMeters() method to numbers, by



The downside of this category system and notation is that to add instance methods to other types, you have to create static methods, and furthermore, there's a first argument which represents the instance of the type we're working on. The other arguments are the normal arguments the method will take as parameters. So it may be a bit less intuitive than a normal method definition we would have added to Distance, should we have had access to its source code for enhancing it. Here comes the @Category annotation, which transforms a class with instance methods into a Groovy category:

```
@Category(Number)
class NumberCategory {
    Distance getMeters() {
        new Distance(number: this)
    }
}
```

No need for declaring the methods static, and the this you use here is actually the number on which the category will apply, it's not the real this of the category instance should we create one. Then to use the category, you can continue to use the use(Category) {} construct. What you'll notice however is that these kind of categories only apply to one single type at a time, unlike classical categories which can be applied to any number of types.

Runtime

Categories
ExpandoMetaClass
MethodMissing

Compile-time

AST Transformations

Compile-time and Runtime Metaprogramming with Groovy

Scott Davis, ThirstyHead.com

scott@thirstyhead.com

Questions?
Thanks for your time.

Image Credits

http://www.flickr.com/photos/timothymorgan/75294154/in/set-1615269/
http://www.flickr.com/photos/cwalker71/2175839970/
http://www.flickr.com/photos/34517346@N02/3464812992/
http://www.flickr.com/photos/chrishedgate/3044800149/
http://www.flickr.com/photos/andrewbain/3649614402/