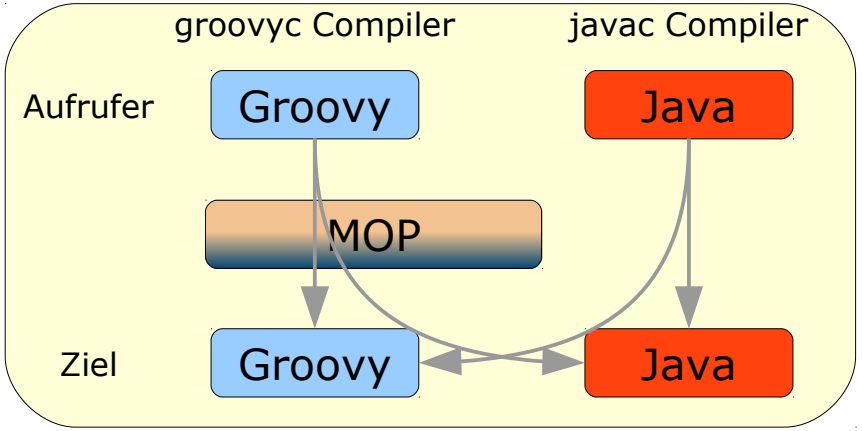
Groovy Advanced

Metaprogrammierung und andere weiterführende Konzepte

©Prof. Dr. Georg Beier

Dynamische Programmierung

- Veränderung von Klassen und Objekten zur Laufzeit
  - „dynamisch" erzeugte Objekte mit speziellen Methoden und Properties
  - Klassen und Objekte zur Laufzeit ändern
    - Methoden ändern oder hinzufügen
    - Felder hinzufügen
- Meta Object Protocol (MOP) macht es möglich
  - Methoden werden in Groovy nicht direkt aufgerufen, sondern über die Metaklasse
  - verschiedene „hook" Methoden können genutzt werden, um einzugreifen
    - invokeMethod, methodMissing, propertyMissing



Expando

- Expando Objekte können zur dynamischen Konstruktion von Objekten genutzt werden
  - Felder
  - Methoden (über Closures)
- Ähnlich zu Maps, aber mit synthetischen Methoden

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

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

Groovy Meta-Programmierung

- Meta-Programmierung := Programierung auf Ebene der Metaklassen
  - dynamisch (d.h. unter Programmkontrolle) Programmcode verändern
  - sinnvoll für Tools und Programmgeneratoren
- Groovys ExpandoMetaClass macht dieses einfach
  - ähnlich einfach wie das ExpandoObjekt auf Instanzebene

- ExpandoMetaClass - Borrowing Methods — Borrowing methods from other classes
- ExpandoMetaClass - Constructors — Adding or overriding constructors
- ExpandoMetaClass - Dynamic Method Names — Dynamically creating method names
- ExpandoMetaClass - GroovyObject Methods — Overriding invokeMethod, getProperty and setProperty
- ExpandoMetaClass - Interfaces — Adding methods on interfaces
- ExpandoMetaClass - Methods — Adding or overriding instance methods
- ExpandoMetaClass - Overriding static invokeMethod — Overriding invokeMethod for static methods
- ExpandoMetaClass - Properties — Adding or overriding properties
- ExpandoMetaClass - Runtime Discovery — Overriding invokeMethod for static methods
- ExpandoMetaClass - Static Methods — Adding or overriding static methods

ExpandoMetaClass

- Aus der JavaDoc:

```
// defines or replaces instance method:
metaClass.myMethod = { args -> }

// defines a new instance method
metaClass.myMethod << { args -> }

// creates multiple overloaded methods of the same name
metaClass.myMethod << { String s -> } << { Integer i -> }

// defines or replaces a static method with the 'static' qualifier
metaClass.'static'.myMethod = { args ->  }

// defines a new static method with the 'static' qualifier
metaClass.'static'.myMethod << { args ->  }

// defines a new constructor
metaClass.constructor << { String arg -> }

// defines or replaces a constructor
metaClass.constructor = { String arg -> }

// defines a new property with an initial value of "blah"
metaClass.myProperty = "blah"
```

ExpandoMetaClass

1. Beispiel

   Ergänze String-Klasse um Methode swapCase
   delegate referenziert das aktuelle (String-) Objekt

```
String.metaClass.swapCase = {->
      def sb = new StringBuffer()
      delegate.each {
            sb << (Character.isUpperCase(it as char) ?
                  Character.toLowerCase(it as char) :
                  Character.toUpperCase(it as char))
      }
      sb.toString()
}

println 'Ääh, Hallo Welt 1234'.swapCase()
```

```
äÄH, hALLO wELT 1234
```

ExpandoMetaClass

2. Beispiel

   String-Klasse bekommt neue Property

```
def s2 = 'kerk'
String.metaClass.lambSpeach =
    'Und das Lamm sprach "hurz"'
println s2.lambSpeach
s2.lambSpeach = 'silence'
println s2.lambSpeach
```

```
Und das Lamm sprach "hurz"
silence
```

ExpandoMetaClass

- Alle Methodenaufrufe gehen über invokeMethod
  - kann überschrieben werden
- Erweiterung der Java Reflection Api
  - Auch Zugriff auf dynamische Groovy Metaeigenschaften

```
class Stuff {
   def invokeMe() { "foo" }
}

Stuff.metaClass.invokeMethod = { String name, args ->
   def metaMethod = Stuff.metaClass.getMetaMethod(name)
   def result
   if(metaMethod) result = metaMethod.invoke(delegate,args)
   else {
      result = "no such method"
   }
   result
}

def stf = new Stuff()

println stf.invokeMe()
println stf.doStuff()
```

```
foo
no such method
```

ExpandoMetaClass


```groovy
class Stuff {
   String shout
   def invokeMe() { "foo" }
   def cry() {System.out.println "$shout" }
}
Stuff.metaClass.swapCase = {->
      def sb = new StringBuffer()
      delegate.shout.each {
           sb << (Character.isUpperCase(it as char) ?
                    Character.toLowerCase(it as char) :
                    Character.toUpperCase(it as char))
      }
      println sb
      delegate.shout = sb.toString()
}

Stuff.metaClass.invokeMethod = { String name, args ->
    def metaMethod = Stuff.metaClass.getMetaMethod(name)
    def result
    if(metaMethod) {
        println "before invoking $name"
        result = metaMethod.invoke(delegate,args)
        println "after invoking $name"
    }
    result
}

def sl = new Stuff(shout: 'Ääh, Hallo Welt 1234')
sl.cry()
println ''
sl.swapCase()
```

```
before invoking cry
Ääh, Hallo Welt 1234
after invoking cry

before invoking swapCase
äÄH, hALLO wELT 1234
after invoking swapCase
```

3. Beispiel
   - Ein Interceptor kann Methodenaufrufe abfangen
   - Für Tracing, Logging, Security
   - Allgemein: Proxy-Methoden

Groovy Categories

- Mit *use* können Objekte lokal um neue Methoden erweitert werden
  - Category- Klasse stellt statische Methoden zur Verfügung
  - use(Category) { ... } macht diese Methoden innerhalb des Closure verfügbar
  - Beispiel:
    numerische Addition von Ziffern-Strings

```
groovy> class StringAddCategory {
groovy>     static def plus(String self, def other) {
groovy>         try {
groovy>             return self.toInteger() + other.toString().toInteger()
groovy>         } catch (NumberFormatException e) {
groovy>             return self << other.toString()
groovy>         }
groovy>     }
groovy> }
groovy> use(StringAddCategory) {
groovy>     println '21' + '15'
groovy>     println '21' + 5
groovy>     println '21' + 5.7
groovy>     println 'siebzehn' + '15'
groovy> }

36
26
215.7
siebzehn15
Execution complete. Result was null.                    16:2
```

Wozu Categories?

- Um lokal spezialisierte Methoden bereitzustellen
- Um Utility Klassen aus der Standardbibliothek einfach zu nutzen
- Um existierende Klassenbibliotheken für einen ganz spezifischen Einsatzzweck zu erweitern
- Um Erweiterungen auf unterschiedlichen Klassen bereitzustellen, die in Kombination sinnvoll sind
  - z.B.: writeEncrypted → OutputStream,
    readEncrypted → InputStream
- Als leichtgewichtige Implementierung für das Decorator-Pattern
- Um die Funktionalität einer "übergewichtigen" Klasse kontrolliert in verschiedene Kategorien von Methoden aufzuspalten

Categories zusammengefasst

- use wendet die Category Methoden im runtime scope des Closure an (im Gegensatz zum literal scope). Daher kann Code aus dem Closure in Methoden faktorisiert werden.
- Die Category Verwendung ist auf den aktuellen Thread beschränkt.
- Category Verwendung ist nicht intrusiv.
- Wenn der receiver type (→ self) eine Superklasse oder sogar ein Interface ist, können die Category-Methoden in allen Subklassen/Implementierungen ohne weitere Konfiguration genutzt werden.
- Namen von Category-Methoden können wie property accessors (simulieren property Zugriff), Operator Methoden und GroovyObject Methoden aussehen. MOP hook Methoden können nicht hinzugefügt werden.
- Category-Methoden können Methodendefinitionen der Metaklasse überschreiben.
- Category-Methoden können die Performance reduzieren.
- Categories können keine neuen Properties mit einem dahinterliegenden Feld einführen.

Groovy Mixins

- Mixins können einer Klasse Fähigkeiten verleihen, ohne Vererbung zu nutzen
  - Viele Interfaces definieren Fähigkeiten:
    Appendable, Adjustable, Activatable, Callable, Cloneable, Closeable, …
- Methodisch sollte Vererbung nur im Fall einer ist-ein Beziehung verwendet werden
- Syntaktisch hat Vererbung Beschränkungen in Java und Groovy
  - Einfachvererbung
  - Intrusiv
  - Nicht mit final Klassen (viele Bibliotheksklassen sind final)
- Groovy unterstützt zwei unterschiedliche Arten von Mixins
  - @Mixin Annotation erzeugt eine AST Transformation (d.h. beeinflusst die Compilation) (deprecated seit 2.3)
  - mixin()-Methode auf Klassenebene (nicht deprecated)

Runtime mixin

- Runtime mixins und die @Mixin Transformation sind sehr unterschiedlich. Runtime mixins sind nicht deprecated in groovy 2.3.
- Runtime mixins können Methoden zu existierenden Klassen hinzufügen, z.B. im JDK. Auf diese Weise erweitert Groovy den JDK.
- Beispiel:

```
class MyStringExtension {
    public static String hello(String self) {
        return "Hello $self!"
    }
}

String.mixin(MyStringExtension)

assert "Vahid".hello() == 'Hello Vahid!'
```

aus Steinars Post auf stackoverflow: http://stackoverflow.com/questions/23121890/difference-between-delegate-mixin-and-traits-in-groovy

Runtime mixin

- Zusammenfassung:
- Add methods to any existing class
  - any classes in the JDK
  - any 3rd party classes
  - or any of your own classes
- Overrides any existing method with the same signature
- Added methods are not visible in Java
- Typically used to extend existing/3rd party classes with new functionality

aus Steinars Post auf stackoverflow: http://stackoverflow.com/questions/23121890/difference-between-delegate-mixin-and-traits-in-groovy

Traits (Merkmale)

Traits sind die bessere Alternative zur @Mixin-Transformation

- Neues Keyword seit Groovy 2.3.1
- Wie Interfaces mit Implementierungen und States
- Erlauben Mehrfachvererbung
- Können nicht instanziiert werden
- Können abstrakte Methoden besitzen
- Klassen sind instanceof traits, die sie implementieren
- Dokumentation in der Groovy "New Documentation"

```
trait Name {
    abstract String name()
    String myNameIs() {
        "My name is ${name()}!"
    }
}
trait Age {
    int age() { 42 }
}

class Person implements Name, Age {
    String name() { 'Vahid' }
}

def p = new Person()
assert p.myNameIs() == 'My name is
Vahid!'
assert p.age() == 42
assert p instanceof Name
assert p instanceof Age
```

Code aus Steinars Post auf stackoverflow: http://stackoverflow.com/questions/23121890/difference-between-delegate-mixin-and-traits-in-groovy

Traits

Zusammenfassung:

- Traits provide an interface with implementation and state.
- A class can implement multiple traits.
- Methods implemented by a trait are visible in Java.
- Compatible with type checking and static compilation.
- Traits can implement interfaces.
- Traits can not be instantiated by themselves.
- A trait can extend another trait.
- Handling of the diamond problem is well-defined.
- Typical usage:
  - add similar traits to different classes.
    - (as an alternative to AOP)
  - compose a new class from several traits

aus Steinars Post auf stackoverflow: http://stackoverflow.com/questions/23121890/difference-between-delegate-mixin-and-traits-in-groovy

Groovy Builder

- Unterstützung bei der Konstruktion von Baumstrukturen
  - Node Builder: allgemeine baumförmige Datenstrukturen
  - MarkupBuilder: HTML und XML Dokumente
  - AntBuilder: Ant Tasks mit Groovy erstellen
  - SwingBuilder: Swing GUIs in Groovy erstellen
- Builder nutzen die Metaprogrammierungsfähigkeiten von Groovy

```
def html   = new groovy.xml.MarkupBuilder()
html.html {
  head {
    title 'Constructed by MarkupBuilder'
  }
  body {
    h1 'What can I do with MarkupBuilder?'
    form (action:'whatever') {
      for (line in ['Produce HTML','Produce XML','Have some fun']){
        input(type:'checkbox',checked:'checked', id:line, '')
        label(for:line, line)
        br('')
} } } }
```

```
<html>
  <head>
    <title>Constructed by MarkupBuilder</title>
```

Swing Builder

- Swing (auch AWT, SWT) GUIs besitzen eine Baumstruktur
- programmatische Konstruktion mit dem SwingBuilder
  - Container und enthaltene Steuerelemente
  - Properties
  - Event Handler Methoden als Closures

```
import groovy.swing.SwingBuilder

swing = new SwingBuilder()
frame = swing.frame(title:'Password') {
    passwordField(columns:10, actionPerformed: { event ->
        println event.source.text
        // any further processing is called here
        System.exit(0)
        }
    )
}
frame.pack()
frame.show()
```

SwingBuilder

- Factory Methoden erzeugen die Steuerelemente (Widgets)
  - die meisten Swing Elemente werden unterstützt

Layouts

- borderLayout
- boxLayout
- cardLayout
- flowLayout
- gridBagLayout
  - gridBagConstraints
  - gbc - alias for GridBagConstraints
- gridLayout
- overlayLayout
- springLayout

- box
  - hbox
  - hglue
  - hstrut
  - vbox
  - vglue
  - vstrut
  - glue
  - rigidArea

Models

- boundedRangeModel
- buttonGroup

- spinnerDateModel
- spinnerListModel
- spinnerNumberModel

- tableModel
  - tableColumn
  - propertyColumn
  - closureColumn

Other

- action
- actions
- imageIcon
- map

Root Windows

- dialog
- frame
- window

Embeddable Windows

- optionPane
- fileChooser
- colorChooser

Containers

- box
- desktopPane
  - internalFrame
- layeredPane
- panel
- scrollPane
  - viewport
- splitPane
- tabbedPane
- toolBar
- container - returns value argument or container attribute

Menus

- menuBar
- popupMenu
- menu
- menuItem
- checkBoxMenuItem
- radioButtonMenuItem

Widgets

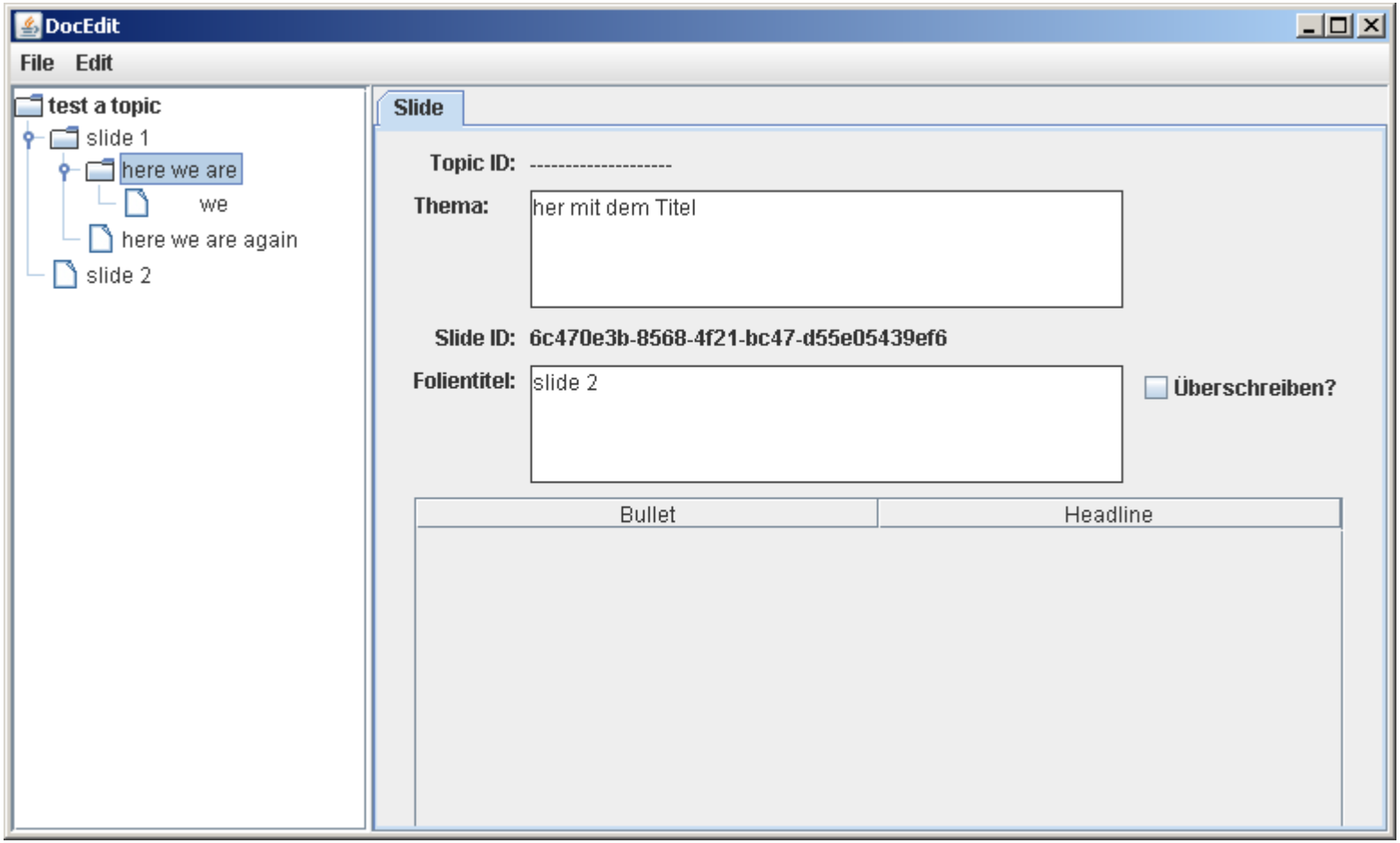
- button
- checkBox
- comboBox
- editorPane
- formattedTextField
- label
- list
- passwordField
- progressBar
- radioButton
- scrollBar
- separator
- slider
- spinner
- table
- textArea
- textPane
- textField
- toggleButton
- tree
- widget - returns value argument or widget attribute

Borders

- compoundBorder
- emptyBorder
- etchedBorder
- lineBorder
- loweredBevelBorder
- loweredEtchedBorder - alias for etchedBorder
- matteBorder
- raisedBevelBorder
- raisedEtchedBorder
- titledBorder

SwingBuilder

- Auch komplexe GUIs lassen sich konstruieren



Komplexe GUIs mit SwingBuilder

- TreeModel wird derzeit nicht direkt in Groovy unterstützt
  - Java DefaultTreeModel leicht zu verwenden
- Zerlegung in überschaubare Teilbäume
- Viel einfacher als programmatische Erstellung in Java
  - aber immer noch umfangreich
  - Standard Layouts aufwändig, Alternative TableLayout

```
def frame = guiBuilder.frame(title: 'DocEdit', defaultCloseOperation: JFrame.EXIT_ON_CLOSE, size: [800, 600]) {
    widget(menuBar)
    splitPane(dividerLocation: 200, dividerSize: 3) {
        scrollPane {
            widget(docTree)
        }
        tabbedPane {
            widget(slidePanel, title: 'Slide', tabToolTip: 'Slide content')
            //                widget(subtopicPanel, title: 'Topic', tabToolTip: 'Topic Properties')
        }
    }
}
```

Komplexe GUIs mit SwingBuilder


```groovy
/**
 * panle to edit topic properties
 */
def slidePanel = guiBuilder.panel() {
    textBorder = lineBorder(color: Color.darkGray)
    tableLayout(cellpadding: 4) {
        //            gbc = gbc(insets:[4,4,4,4])
        tr {
            td(rowfill: false, align: 'RIGHT') {label('Topic ID:')}
            td(rowfill: true, align: 'LEFT') {label(id: 'topicId', '-' * 20, toolTipText: 'immutable ID of this topic')}
        }
        tr {
            td(colfill: true, align: 'RIGHT') {label('Thema:')}
            td(colfill: true, align: 'CENTER', rowspan: 2) {textArea(id: 'topicTitle', border: textBorder, columns: 30, rows: 4, text: 'her mit dem Titel')}
        }
        tr {}
        tr {
            td(rowfill: false, align: 'RIGHT') {label('Slide ID:')}
            td(rowfill: true, align: 'LEFT') {label(id: 'slideId', '-' * 20)}
        }
        tr {
            td(rowfill: false, align: 'RIGHT', valign: 'TOP') {label('Folientitel:')}
            td(colfill: true, align: 'LEFT', valign: 'TOP') {textArea(id: 'slideTitle', border: textBorder, columns: 30, rows: 4)}
            td(rowfill: false, align: 'LEFT', valign: 'TOP') {checkBox(id: 'slideTitleCheckBox', label: 'Überschreiben?')}
        }
        tr {
            td(colspan: 3, colfill: true) {
                scrollPane {
                    subTopicTable = table(id: 'subTopicTable') {
                        subTopicModel = tableModel(id: 'subTopicModel', list: subtopics) {
                            closureColumn(header: 'Bullet', type: Boolean.class, read: readSubTopicBullit, write: writeSubTopicBullit)
                            closureColumn(header: 'Headline', type: String.class, read: {}, write: {})
                        }
                    }
                }
            }
        }
    }
}
```