Groovy Einführung groovy.codehaus.org

Dynamische Skriptsprache in der Java Umgebung

©Prof. Dr. Georg Beier

Ein besseres Java?

- Javas statische Typisierung hat auch Nachteile
  - Teilweise strukturell gleiche Klassen ohne gemeinsame Oberklasse können nicht polymorph genutzt werden:
    Strings können bspw. nicht als Arrays von Char betrachtet werden
- Methoden gibt es nur in Klassen, auch wenn man diese nicht braucht
  - Resultiert in zahlreichen „Utility Klassen"
- Inkonsistente Behandlung von Arrays, Collections und Strings
  - length, size(), length() usw.
- Dynamische Bindung berücksichtigt nicht Methodenparameter
- Java ist „verbose": viel Code mit wenig Inhalt
- Die Java Standardbibliothek lässt Wünsche offen
  - Date Klasse ist sehr unhandlich
  - String könnte viel mehr können
  - Collections werden in anderen Sprachen eleganter behandelt
- Neue Sprachfeatures kommen nur sehr langsam in den Standard
  - z.B. Closures

(einige) Groovy Eigenschaften

- "Dynamische" Skriptsprache auf der JVM Plattform
  - Unterstützt Java Syntax → einfacher Einstieg
  - dynamische und statische Typisierung
  - Zusätzliche Sprachfeatures inspiriert von Python, Ruby, Smalltalk
- reduziert immer gleichen Infrastruktur-Code für höhere Produktivität
  - Erweiterung der Java Standardbibliothek
  - vereinfachte Arbeit mit Collections, Arrays und Maps
  - leistungsfähigere Groovy Strings (GString)
  - GUI, Web, Datenbank- und Konsolenanwendungen
- Vollständig kompatibel zu Java Klassen und Bibliotheken
  - Grundlage für die Grails Web Application Plattform

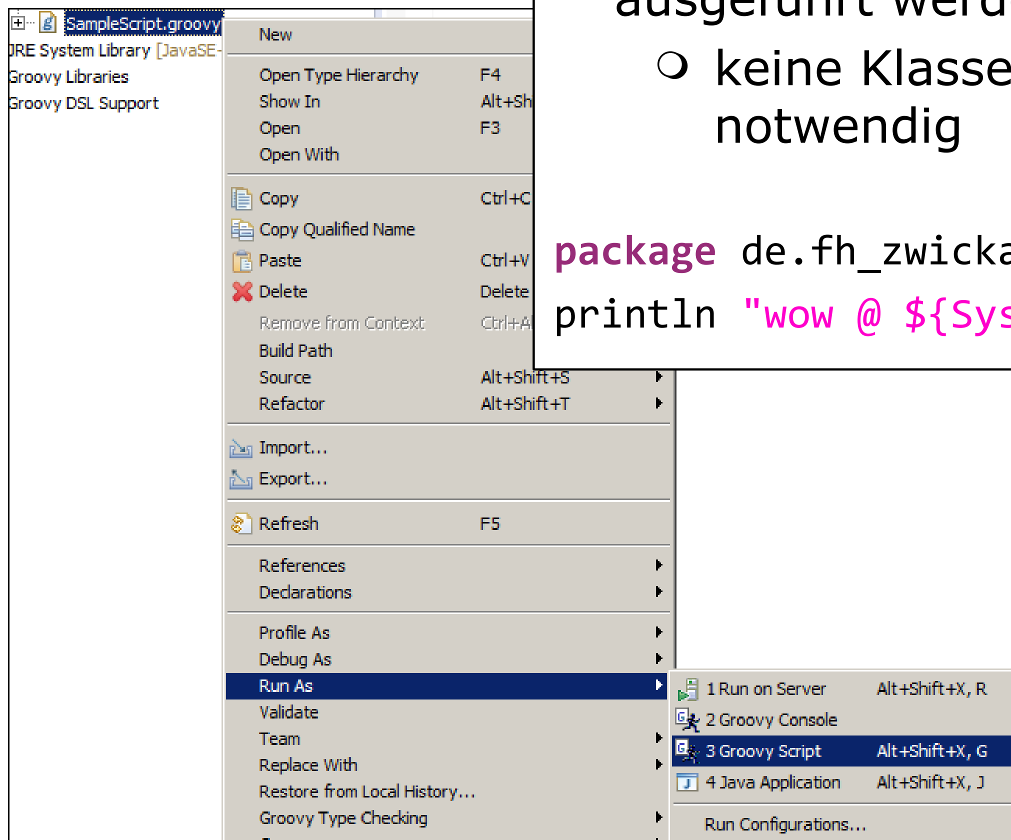
Groovy Programmierung

- Groovy wird in vielen modernen IDEs unterstützt
  - Eclipse, IntelliJ, NetBeans sowie Editor-Plugins
- Mainstream der IDE Entwicklung hat sich auf Eclipse verschoben (war einmal IntelliJ)
- Entwicklung von Sprache und Eclipse Support durch open source community und Pivotal
  - Pivotal ⊂ SpringSource ⊂ VmWare
- Eclipse für Groovy (und Grails):
  - Groovy/Grails Tool Suite 3.6.0.RELEASE - based on Eclipse Luna 4.4: Konfigurierte und „gebrandete" IDE von SpringSource
  - Alternativ Groovy Plugin in vorhandene Eclipse Installation installieren
  - Beides führt zu großen Eclipse-Installationen, die den Wunsch nach einem schnelleren Rechner aufkommen lassen können ☺☻☺

Groovy Klasse oder Script ausführen

- In Groovy können Klassen und Skripte direkt ausgeführt werden
  - keine Klasse mit main() Methode notwendig

```
package de.fh_zwickau.pti.groovyintro
println "wow @ ${System.currentTimeMillis()}"
```



Groovy Console

- Die Groovy Console ermöglicht direktes interaktives Arbeiten und Ausprobieren mit den in Eclipse entwickelten Dateien

SampleScript.groovy - GroovyConsole

File Edit View History Script Help

```
1 package de.fh_zwickau.pti.groovyintro
2
3 println "wow @ ${System.currentTimeMillis()}"
```

```
groovy> package de.fh_zwickau.pti.groovyintro
groovy> println "wow @ ${System.currentTimeMillis()}"

wow @ 1407836950521
```

Execution complete. Result was null. 1:1

Profile As
Debug As
Run As
Validate
Team
Replace With
Restore from Local History...
Groovy Type Checking
Groovy

1 Run on Server Alt+Shift+X, R
2 Groovy Console
3 Groovy Script Alt+Shift+X, G
4 Java Application Alt+Shift+X, J
Run Configurations...

Hello World – "eingedampft"

Java Beispiel

```
public class HelloWorld {
  String name;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public String greet() {
    return "Hello " + name;
  }

  public static void main(String args[]) {
    HelloWorld hello = new HelloWorld();
    hello.setName("Java");
    System.out.println(hello.greet());
  }
}
```

Hello World – "eingedampft"

Groovy

Java Code läuft zu 98% unverändert

Fehler in Groovy:

`main(String args[])`

```
public class HelloWorldG{
  String name;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public String greet() {
    return "Hello " + name;
  }

  public static void main(String[] args) {
    HelloWorldG hello = new HelloWorldG();
    hello.setName("Groovy");
    System.out.println(hello.greet());
  }
}
```

Hello World – "eingedampft"

Shortcuts

- Semikolon am Zeilenende unnötig, wenn Semantik unzweideutig
- get/set Methoden automatisch erzeugt
- Argumenttyp von main ist immer String[]
- Alles ist public, wenn nicht anders definiert
- Konsolenausgabe mit println

```
public class HelloWorldG {
  String name;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public String greet() {
    return "Hello " + name;
  }

  public static void main(String[] args) {
    HelloWorldG hello = new HelloWorldG();
    hello.setName("Groovy");
    System.out.println(hello.greet());
  }
}
```

Hello World – "eingedampft"

Shortcuts

- Semikolon am Zeilenende unnötig, wenn Semantik unzweideutig
- get/set Methoden automatisch erzeugt
- Argumenttyp von main ist immer String[]
- Alles ist public, wenn nicht anders definiert
- Konsolenausgabe mit println

```
class HelloWorldG {
  String name

  String greet() {
    return "Hello " + name
  }

  static void main(args) {
    HelloWorldG hello = new HelloWorldG()
    hello.setName("Groovy")
    println(hello.greet())
  }
}
```

Hello World – "eingedampft"

dynamische Typen

- Variablentypen können dynamisch bestimmt werden (def)
- Methoden-Rückgabe-typen können dynamisch bestimmt werden
- return oft unnötig, letztes Ergebnis einer Methode wird zurückgegeben

```
class HelloWorldG {
  String name

  String greet() {
    return "Hello " + name
  }

  static void main(args) {
    HelloWorldG hello = new HelloWorldG()
    hello.setName("Groovy")
    println(hello.greet())
  }
}
```

Hello World – "eingedampft"

dynamische Typen

- Variablentypen können dynamisch bestimmt werden (def)
- Methoden-Rückgabe-typen können dynamisch bestimmt werden
- return oft unnötig, letztes Ergebnis einer Methode wird zurückgegeben

```
class HelloWorldG {
  def name

  def greet() {
    "Hello " + name
  }

  static main(args) {
    def hello = new HelloWorldG()
    hello.setName("Groovy")
    println(hello.greet())
  }
}
```

Hello World – "eingedampft"

POGOs, GString

- automatische Generierung initialisierender Konstruktoren
- Zugriff auf Bean-Properties (get/set Methoden):
  - obj.name
  - obj["name"]
- ${expression} kann in GString Objekte eingefügt werden

```
class HelloWorldG {
  def name

  def greet() {
    "Hello " + name
  }

  static void main(args) {
    def hello = new HelloWorldG()
    hello.setName("Groovy")
    println(hello.greet())
  }
}
```

Hello World – "eingedampft"

Groovy Beans, GString

- automatische Generierung initialisierender Konstruktoren
- Zugriff auf Bean-Properties (get/set Methoden):
  - obj.name
  - obj[name]
- ${expression} kann in GString Objekte eingefügt werden
- == Operator überprüft Gleichheit, nicht Identität
- mit assert kann man logische Ausdrücke überprüfen

```
class HelloWorldG {
  def name

  def greet() {
    "Hello ${name}"
  }

  static void main(args) {
    def hello = new HelloWorldG(name:"Groovy")
    println(hello.greet())

    assert hello.name == "Groovy"
    assert hello["name"] == "Groovy"
  }
}
```

Java-Beans in Groovy

- Java-Beans
  - Zugrif auf Felder mit get/set Methoden
  - Parameterloser Konstruktor

Groovy:

- Felder mit default visibility (nur dann!) erzeugen
  - ➔ private Instanzvariable
  - ➔ public getter und setter
  - ➔ accessor methods werden auch bei scheinbar direktem Feldzugriff aufgerufen
- Modifier final vor Feld
  - ➔ nur getter, Variable wird final
- mit .@ wird direkt auf das Feld zugegriffen

```
/**
 * angepasstes Beispiel aus Groovy in Action 2 Ed.
 */
class MyDemoBean implements Serializable {
  def untyped
  /*public final*/ String typed  // erzeugen Fehler
  def item1, item2
  def assigned = 'default value' // kann final werden?
}

def bean = new MyDemoBean()
assert 'default value' == bean.getAssigned()
bean.setUntyped('some value')
assert 'some value' == bean.getUntyped()
bean = new MyDemoBean(typed:'another value')
assert 'another value' == bean.getTyped()
println "alles OK soweit"
println bean.class.methods.name.grep(~/[gs]et.*/).sort()
/*
bean.assigned = 'new value'
println 'auch das noch?'
*/
```

Strings in Groovy

- 2 Arten von Strings:
  - Java Strings: java.lang.String
  - Groovy GStrings: groovy.lang.GString

- Strings fast wie in Java
  - Strings werden mit '...' begrenzt: 'Java ist nett'
  - Mehrzeilige Strings werden mit '''...''' begrenzt: '''Ein String kann über\ mehrere Zeilen gehen, ein \ am Zeilenende verhindert den Umbruch'''
- Verwendung von \ wie in Java

- Groovy GStrings können mehr
  - Begrenzt mit "..."
  - Ergebnis beliebiger Ausdrücke in ${...} werden als String eingefügt, Felder auch einfach mit $feldname
  - Mehrzeilige GStrings mit """..."""
  - Slashy Strings werden mit /.../ begrenzt: keine spezielle Bedeutung von \
  - Besonders geeignet für Regular Expressions

Closures: Code als Objekt

- Auch Code kann in Groovy als ein Objekt behandelt werden
  - Code-Blöcke (d.h. Inhalt von { … }) sind Objekte und können in Variablen gespeichert werden
  - Dies nennt man Closure
- Closures können als wiederverwendbare Codeblöcke verstanden werden
  - Gibt es auch z.B. in JavaScript, Ruby
- Closures können (fast immer) innere Klassen ersetzen
  - Die gibt es (seit Version 1.7) auch in Groovy
- Closures können benannte Parameter haben
  - Default Parameter namens **it**, falls <u>kein</u> Parameter definiert ist
- Closures können in Groovy auf den Typ eines Interface mit einer Methode "gezwungen" (coerced) werden

Hello World – "eingedampft"

Closures

- Auch Code-Blöcke sind Objekte: Closures
- Variablen können Closures aufnehmen
- Aufhebung der Unterscheidung von Methode und Datenfeld
- Definitionskontext von Closures ist wichtig

```
class HelloWorldG {
  def name
  def greet

  static void main(args) {
    def hello = new HelloWorldG(name:"Groovy")
    hello.greet = {"Hello $hello.name"}
    println(hello.greet())
    hello.greet = {"Bye bye $hello.name"}
    println(hello.greet())
  }
}
```

Closure Beispiele

```
def greet = { name -> println “Hello $name” }
greet( “Groovy” )
// prints Hello Groovy

def greet = { println “Hello $it” }
greet( “Groovy” )
// prints Hello Groovy

def iCanHaveTypedParametersToo = { int x, int y ->
   println “coordinates are ($x,$y)”
}

def myActionListener = { event ->
   // do something cool with event
} as ActionListener
```

Closures und "currying"

Currying ist eine Programmiertechnik, die eine Funktion in eine andere transformiert, indem sie einen oder mehrere Input-Parameter festhält (Idee: Konstanten)

Die Methode curry(…) von Closure gibt eine neue Closure zurück und bindet Parameter von links nach rechts (→ rcurry(…), ncurry(…))

```
// a closure with 3 parameters, the third one is optional
// as it defines a default value
def getSlope = { x, y, b = 0 ->
   println "x:${x} y:${y} b:${b}"
    (y - b) / x
}

assert 1 == getSlope( 2, 2 )
def getSlopeX = getSlope.curry(5) // x = 5
assert 1 == getSlopeX(5)
assert 0 == getSlopeX(2.5,2.5)
// prints
// x:2 y:2 b:0
// x:5 y:5 b:0
// x:5 y:2.5 b:2.5
```

Closure Kontext

- Closures können auf den Kontext zugreifen, in dem sie erzeugt wurden, sogenannte „freie" Variablen
  - lokale Variablen und Methodenparameter
  - bleiben als Kopie erhalten, auch wenn Closure den Definitionsbereich ihrer Definition verlässt, bspw. als Rückgabewert einer Methode
- Implizite Variablen
  - **it**: impliziter erster Closure-Parameter, wenn nicht angegeben
  - **this**: die Instanz der umgebenden Klasse
  - **owner**: das umgebende Objekt (this oder umgebende Closure)
  - **delegate**: standardmäßig gleich owner, kann aber auf ein anderes Objekt gesetzt werden

Interfaces und Closures

- Interfaces können in Groovy direkt durch Closures implementiert werden
  - Analogie: lokale innere Klassen in Java
- Interface mit einer Methode: passendes Closure mit *as* auf diesen Typ abbilden
- Interface mit mehreren Methoden: Map von Closures mit *as* auf diesen Typ abbilden

```
groovy> interface TrafficLight {
groovy>     def free(boolean f);
groovy> }
groovy> interface Pedestrian {
groovy>     void stop();
groovy>     void go();
groovy> };
groovy> def ampel = {boolean f ->  f ? "green" : "red" } as TrafficLight
groovy> def maxMuster = [stop:{println "I always stop on ${ampel.free(false)}"},
groovy>                     go:{println "I only go on ${ampel.free(true)}"}] as Pedestrian
groovy> maxMuster.go()
groovy> maxMuster.stop()

I only go on green
I always stop on red
Execution complete. Result was null.                                                    10:4
```

Groovy Functions (→ Methoden?)

- Groovy Functions werden außerhalb von Klassen definiert
- Groovy Functions ähneln Closures mit ein paar fundamentalen Unterschieden
    - Functions können nicht auf einen Kontext zugreifen
    - Functions müssen benannt und mit **def** definiert werden
    - Function Definitionen können nicht wie Closures geschachtelt werden
    - Functions können rekursiv aufgerufen werden
    - Functions können mit dem .& Operator in Closures umgewandelt werden

Kollektive Datentypen: Ranges

- Erzeugen eine Liste sequentieller Werte (Objekte!)
  - .. Notation für inklusiven Range
  - ..< Notation für exklusiven Range
- für alle Objekte verfügbar, die java.lang.Comparable implementieren
- sind Objekte → Ranges als Parameter, Variablen

```
// an inclusive range
def range = 'a'..'d'
assert range.size() == 4
assert range.get(2) == 'c'
```

```
// lets use an exclusive range
range = 5..<8
assert range.size() == 3
assert range.get(2) == 7
assert range[2] == 7
assert range instanceof java.util.List
assert range.contains(5)
assert ! range.contains(8)
```

```
for (i in 1..10) {
  println "Hello ${i}"
}
```

Kollektive Datentypen: Listen

- Erweiterung von java.util.List
- Konstruktion mit []
- Indizierung wie Array möglich, Index ist zyklisch (keine Exception)
- Viele zusätzliche Operatoren und Methoden
  - Methoden mit Closures: collect, find, findAll, groupBy, max, min, sort, unique, reverseEach, sort

```
def list = [5, 6, 7, 8]
assert list.size == 4
assert list.size() == 4
assert list.class == ArrayList
```

```
def list = []; assert list.size() == 0
list << 5; assert list.size() == 1
```

```
assert ! [] //an empty list evaluates as false
assert [1] && ['a'] && [0] && [0.0] && [false] && [null]
    //all other lists, irrespective of contents, evaluate as true
```

```
assert [1,2] + 3 + [4,5] + 6 == [1, 2, 3, 4, 5, 6]
assert [1,2].plus(3).plus([4,5]).plus(6) == [1, 2, 3, 4, 5, 6]
```

Kollektive Datentypen: Maps

- Erweiterung von java.util.Map
- Definition durch *[key:value, key:value, key:value]*
  - key Typ ist per default String
- Leere Map: *[:]*
- Zugriff auf Values mit *aMap[key]* oder *aMap.get(key)*
- Viele zusätzliche Operatoren und Methoden
  - Methoden mit Closures: collect, find, findAll
  - Auch Methoden auf keySet und values
- Ähnliches Verhalten wie Beans

```
def map = [name:"Gromit", likes:"cheese", id:1234]
assert map.get("name") == "Gromit"
assert map.get("id") == 1234
assert map["name"] == "Gromit"
assert map['id'] == 1234
```

```
def map = [name:"Gromit", likes:"cheese", id:1234]
assert map.name == "Gromit"
assert map.id == 1234

def emptyMap = [:]
assert emptyMap.size() == 0
emptyMap.foo = 5
assert emptyMap.size() == 1
assert emptyMap.foo == 5
```

Überall Iteratoren

- Iteratoren können in fast jedem Kontext verwendet werden, Groovy findet heraus, was zu tun ist (wie in Ruby)
- Iteratoren machen die Ausdrucksstärke von Closures für alle iterierbaren Konstrukte verfügbar, alle Iteratoren akzeptieren ein Closure als Parameter
- Iteratoren entlasten die Programmierung von Schleifenkonstrukten

```
def printIt = { println it }
// 3 ways to iterate from 1 to 5
[1,2,3,4,5].each printIt
1.upto 5, printIt
(1..5).each printIt

// compare to a regular loop
for( i in [1,2,3,4,5] ) printIt(i)
// same thing but use a Range
for( i in (1..5) ) printIt(i)

[1,2,3,4,5].eachWithIndex
      { v, i -> println "list[$i] => $v"
   }
// list[0] => 1
// list[1] => 2
// list[2] => 3
// list[3] => 4
// list[4] => 5
```

Regular Expressions

- Match Operator: ==~, Find Operator: =~
  - Anwendung auf beliebige Objekte → toString()
  - rechte Seite wird als Regular Expression interpretiert
- Regular Expressions können sehr einfach mit ~/.../ erzeugt werden
- Viele weitere Möglichkeiten (→ Buch, Website)

```
groovy> def pattern = /.*\d\d.*/     // 2 Ziffern im Text?
groovy> def str = "Wohnen im 12. Stock!"
groovy> print str ==~ pattern

true
Execution complete. Result was n
```

```
groovy> def pattern = /\d\d/     // 2 Ziffern
groovy> def str = "Wohnen im 12. Stock!"
groovy> print str =~ pattern ? "gefunden" : "nicht da"

gefunden
Execution complete. Result was n
```

```
groovy> def pattern = 12     // 2 Ziffern
groovy> def str = "Wohnen im 12. Stock!"
groovy> print str =~ pattern ? "gefunden" : "nicht da"

gefunden
Execution complete. Result was null.
```

Mehr Groovy Features

- Default Werte für Parameter wie in PHP
- Benannte Parameter wie in Ruby
- Operator overloading, realisiert mit Namenskonvention, z.B.

+	plus()
[ ]	getAt() / putAt()
<<	leftShift()

```
 1 class Complex {
 2     Double real
 3     Double imag
 4
 5     Complex plus(Complex c) {
 6         new Complex(real: (this.real + c.real), imag: (this.imag + c.imag))
 7     }
 8
 9     String toString() {
10         "($real, $imag)"
11     }
12
13     static void main(args) {
14         def c1 = new Complex(real: 17.0 , imag: 4.0)
15         def c2 = new Complex(real: 7.7 , imag: 14.0)
16         println(c1 + c2)
17     }
18 }
```

Groovy Truth

- Konditionale Ausdrücke sind in Groovy mit allen Typen außer null möglich (nicht nur mit Boolean)
- Regeln für Boole'sche Tests:

Laufzeit-Typ	**Bedingung für "True"**
Boolean	Wert ist true
Matcher (Regular Expression)	Mindestens ein Match
Collection	Collection nicht leer
Map	Map nicht leer
String, GString	String nicht leer
Number, Character	Wert ungleich 0
Alle weiteren	Referenz nicht null

Spezielle Operatoren

** Potenzierung

<=> Vergleich (siehe compareTo())

.@ direkter Zugriff auf Felder

objekt.feld ruft implizit getter/setter auf

.& Zugriff auf eine vorhandene Methode als Closure

?. Sichere Dereferenzierung, fängt möglichen Zugriff auf *null* ab

*. sicherer Zugriff auf das referenzierte Feld für alle Elemente einer Menge

```
groovy> def list = [10, "Mond", Math.PI]
groovy> println list*.class.name

["java.lang.Integer", "java.lang.String", "java.lang.Double"]
```

Execution complete. Result was null. 3:25

Navigation in Objektgraphen: GPath

- Einfacher Zugriff auf verbundene Objekte
  - Dereferenzierungsoperatoren: .   *.   ?.
  - selektierende Methoden (meist mit Closure als Parameter)
- Zugriff auf einfache Felder (fast wie in Java)
- aber auch Zugriff auf alle oder ausgewählte Elemente in Collections

```
groovy> println this.class.methods.name.grep(~/[gs]et.*/).sort()

["getBinding", "getClass", "getMetaClass", "getProperty", "setBinding",
"setMetaClass", "setProperty"]
Execution complete. Result was null.                                    1:13
```

Multimethods

- Bei polymorphen Methoden wird in Groovy der Laufzeit-Typ der Parameter berücksichtigt, bei Java der statische Typ

```java
1  public class PolymorphicExample {
2      public String oracle(Object o) {
3          return "got Object " + o + " of type " + o.getClass();
4      }
5      public String oracle(String o) {
6          return "got String " + o + " of type " + o.getClass();
7      }
8      public static void main(String[] args) {
9          PolymorphicExample p = new PolymorphicExample();
10         Object x = 1;
11         Object y = "foo";
12         String s = "bar";
13         System.out.println(p.oracle(x));
14         System.out.println(p.oracle(y));
15         System.out.println(p.oracle(s));
16     }
17 }
```

```
got Object 1 of type class java.lang.Integer
got Object foo of type class java.lang.String
got String bar of type class java.lang.String
```

```groovy
groovy> class PolymorphicExample {
groovy>     def oracle(Object o)
groovy>         { "got Object $o of type ${o.getClass()}" }
groovy>     def oracle(String o)
groovy>         { "got String $o of type ${o.getClass()}" }
groovy> }
groovy> def p = new PolymorphicExample()
groovy> Object x = 1
groovy> Object y = "foo"
groovy> String s = "bar"
groovy> println(p.oracle(x))
groovy> println(p.oracle(y))
groovy> println(p.oracle(s))

got Object 1 of type class java.lang.Integer
got String foo of type class java.lang.String
got String bar of type class java.lang.String
```