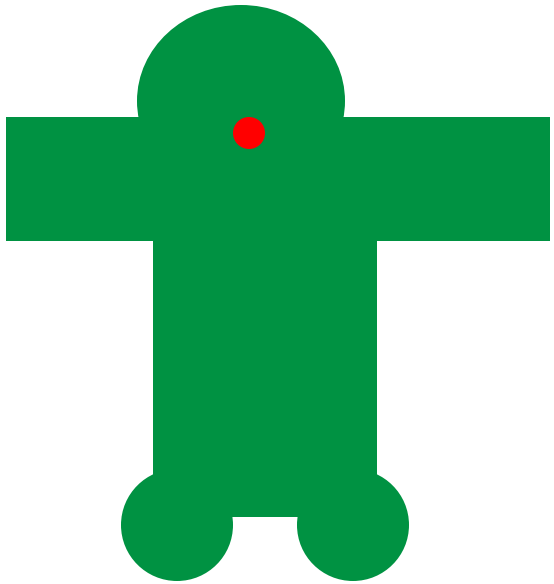
PM2 – Programmieren in Java

Unterschiede zwischen Java und Ruby am Figurenbeispiel

Das Figurenbeispiel

- Eine Figur ist entweder ein Kreis, ein Rechteck oder eine Überlagerung von zwei Figuren.
- Aufgabe: Schreiben Sie ein Programm, das für Punkte im kartesischen Koordinatensystem berechnet, ob der Punkt innerhalb einer Figur liegt.



Vorgehen

- Ausgangspunkt ist die Implementierung des Figurenbeispiels in Ruby.
- Diese wird in der Vorlesung beispielhaft in ein gültiges und korrektes Java-Programm transformiert.
- Der Quelltext der Ruby-Implementierung ist dem der Vorlesung beigefügten *v1r-java-vs-ruby(r).zip* zu entnehmen.
- **Ziel:** Die wesentliche Unterschiede zwischen den beiden Sprachen am Beispiel erfahren.

Abstrakte Klassen und abstrakte Methoden

Java: Abstrakte Klassen und Methoden sind Bestandteil der Sprache

- abstrakte Klassen sind Klassen, von denen keine Instanzen erzeugt werden können.
- Syntax am Bsp.: ***abstract class Figur{...}***
- abstrakte Methoden sind Methoden, die in den Subklassen implementiert sein müssen.
- Syntax am Bsp.: ***abstract boolean in(Punkt p);***
- die Überprüfung der Regeln übernimmt der Compiler vor der Laufzeit.

Ruby: Module sind ein Äquivalent für abstrakte Klassen

- das Äquivalent zu abstrakten Klassen in Ruby sind Module. Auch von Modulen lassen sich keine Instanzen erzeugen.
- abstrakte Methoden existieren in Ruby nicht.
- um zu erzwingen, dass Methoden abstrakter Klassen implementiert werden, muss in Ruby
  - die Klasse ***Object*** erweitert und
  - eine Methode programmatisch als ***abstract*** markiert werden.
- die Überprüfung der Regeln finden ausschließlich zur Laufzeit statt.

Abstrakte Klassen und abstrakte Methoden

Java:

```
package figuren;

public abstract class Figur {

  public abstract boolean in(Punkt p);
}
```

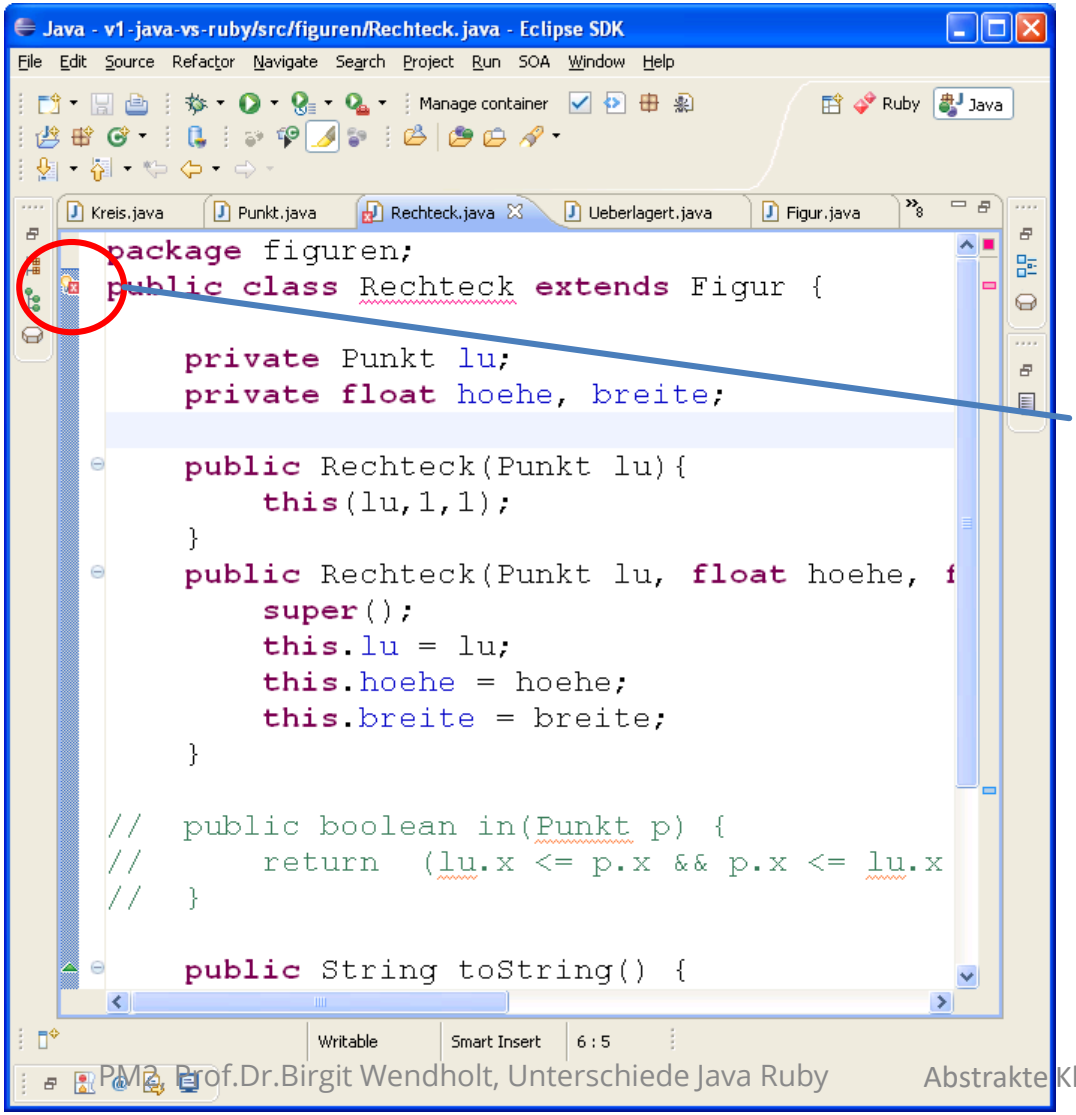
Ruby:

```
require "Object"
module Figur
  def in(p)
    abstract
  end
end
class Object
  def abstract()
    raise AbstractMethodError
  end
end
class AbstractMethodError < StandardError
end
```

Abstrakte Klasse *Figur* und abstrakte Methode *in(Punkt p)*. Wenn Subklassen von *Figur* *in(Punkt p)* nicht überschreiben, gibt es einen Compilerfehler. Die Source wird nicht in ein lauffähiges Programm übersetzt.

Abstrakte Klasse *Figur* (*module*), abstrakte Methode *in(p)*: Methodenaufruf *abstract* aus der erweiterten Klasse *Object* erzeugt einen Laufzeitfehler, wenn Subklassen von *Figur* *in* nicht überschreiben.

Compilerfehler bei nicht implementierter abstrakter Methode *in(Punkt p)*



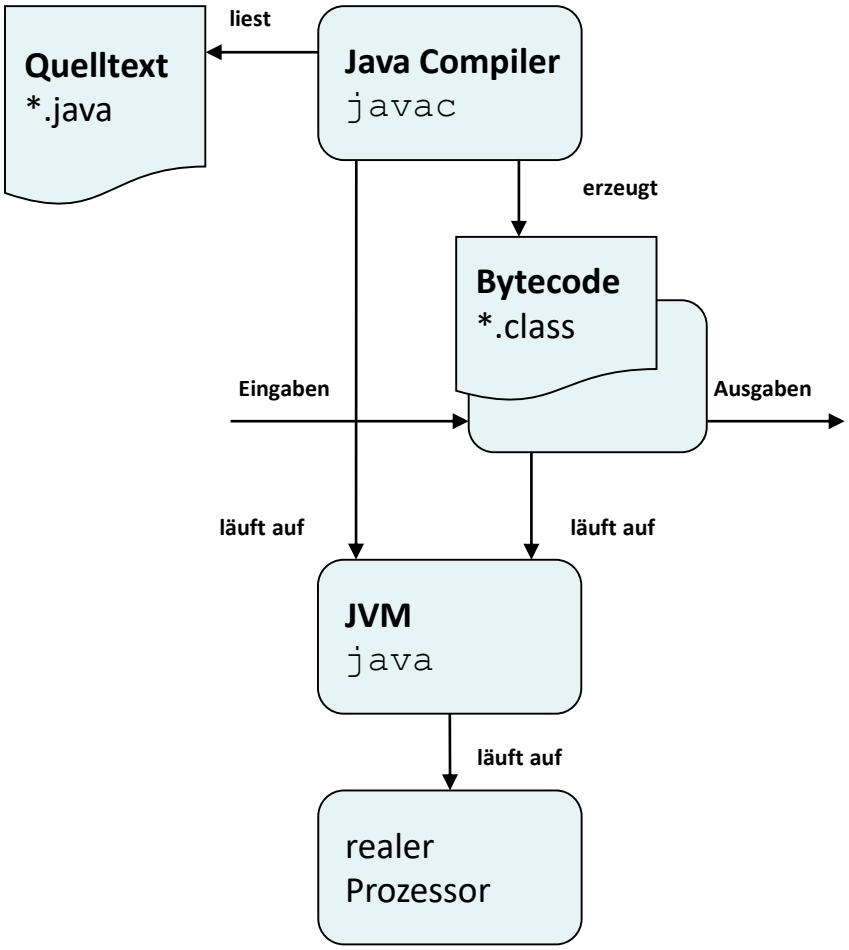
Compilerfehler – Meldungstext:

the type Rechteck must implement the inherited abstract method Figur.in(Punkt)

Java Compiler – Quellcode zu Bytecode
JVM – Laufzeitumgebung für Bytecode

- **Java Compiler (javac.exe)**
  - Übersetzt Java-Quellcode in Bytecode. Dabei werden Syntaxregeln und z.B. Typregeln überprüft.
  - Fehler bei der Übersetzung (Compilerfehler) verhindern das Erzeugen von ausführbarem Bytecode.
  - Java-Quellcode: .java Dateien
  - Java-Bytecode: .class Dateien

- **JVM (Java Virtuelle Maschine) Laufzeitumgebung (java.exe)**
  - führt den Java-Bytecode auf verschiedenen Betriebssystemen und Prozessoren aus.
  - Laufzeitfehler treten in der JVM auf.



CLASSPATH
wie findet die Java-Umgebung Klassen

- Sowohl der Compiler (***javac***) als auch die virtuelle Maschine (***java***) müssen im Bytecode referenzierte Klassen auflösen können.
- Die Klassen der Standardbibliotheken werden immer gefunden.
- Zusatzbibliotheken sowie der Bytecode der selbst geschrieben Klassen werden über den ***CLASSPATH*** gefunden.
- Dies ist eine Umgebungsvariable, die im jeweiligen Betriebssystem gesetzt werden kann oder beim Aufruf von ***javac*** oder ***java*** als Option übergeben werden kann.
- Eine Klasse wird in Java eindeutig durch den vollständigen Package-Namen gefolgt von dem Namen der Klasse identifiziert. Dieser Name wird auch voll qualifizierter Name der Klasse genannt.
- In den ***CLASSPATH*** gehören keine Package-Namen.
- Standardmäßig ist das aktuelle Verzeichnis („.“) immer im ***CLASSPATH***.

CLASSPATH
wie findet die Java-Umgebung Klassen

- Ein Beispiel:
  - Im Verzeichnis ***C:\tmp*** liegt ein Unterverzeichnis ***mypack\sub*** mit der Datei ***C1.class*** (Bytecode Klasse ***C1***).
  - Das Package von ***C1*** ist ***mypack.sub***, der voll qualifizierte Name ist ***mypack.sub.C1***.
  - Um die Klasse ***C1*** von beliebigen Stellen ansprechen zu können fügen wir das Verzeichnis ***C:\tmp*** (den Präfix für die Package-Struktur) dem CLASSPATH hinzu. (Windows: ***set classpath=%classpath%;C:\temp***)
  - Dann wird die Klasse ***C1*** von beliebigen Orten gefunden.

```
package mypack.sub;

public class C1 {
  public static void main(String[] args) {
         System.out.println("C1 gefunden");
         System.out.println(System.getProperty("java.class.path"));
  }
}
```

Programme Ausführen von Konsole

- Wir fügen den Pfad-Präfix zu unseren kompilieren Java-Dateien dem ***classpath*** hinzu
- Wir wechseln in das User Verzeichnis: ***cd %userprofile%***
- Und starten dann die virtuelle Maschine mit dem voll qualifizierten Namen der Java Datei: ***java mypack.sub.C1***
- Voilà, die Klasse wird gefunden, obwohl wir uns nicht in ***C:\temp*** befinden.

```
c:\temp>dir
 Datenträger in Laufwerk C: ist Windows7_OS
 Volumeseriennummer: D29C-CA3F

 Verzeichnis von c:\temp

21.09.2017  14:23    <DIR>          .
21.09.2017  14:23    <DIR>          ..
21.09.2017  14:23    <DIR>          mypack
06.07.2017  17:36    <DIR>          NVIDIA
               0 Datei(en),              0 Bytes
               4 Verzeichnis(se), 305.572.040.704 Bytes frei

c:\temp>set classpath=%classpath%;C:\temp

c:\temp>cd %userprofile%

C:\Users\Birgit Wendholt>java mypack.sub.C1
C1 gefunden
%classpath%;C:\temp
```

Programme Ausführen von Konsole

- Alternativ lässt sich der CLASSPATH beim Aufruf von ***java*** als Option übergeben:
- ***java –classpath c:\temp mypack.sub.C1***
- **Kurzform**: ***java –cp c:\temp mypack.sub.C1***
- Voilà, auch hier wird die Klasse gefunden, obwohl wir uns nicht in ***C:\temp*** befinden.

```
17.09.2017  17:42    <DIR>          Windows Mail
20.03.2017  06:41    <DIR>          Windows Media Player
18.03.2017  23:03    <DIR>          Windows Multimedia Platfor
m
06.07.2017  18:10    <DIR>          Windows NT
17.09.2017  17:42    <DIR>          Windows Photo Viewer
18.03.2017  23:03    <DIR>          Windows Portable Devices
18.03.2017  23:03    <DIR>          Windows Security
18.03.2017  23:03    <DIR>          WindowsPowerShell
               0 Datei(en),              0 Bytes
              47 Verzeichnis(se), 305.567.498.240 Bytes frei

C:\Program Files>java -cp c:\temp mypack.sub.C1
C1 gefunden
c:\temp

C:\Program Files>
```

Nützliches
Java Programm mit Parametern starten

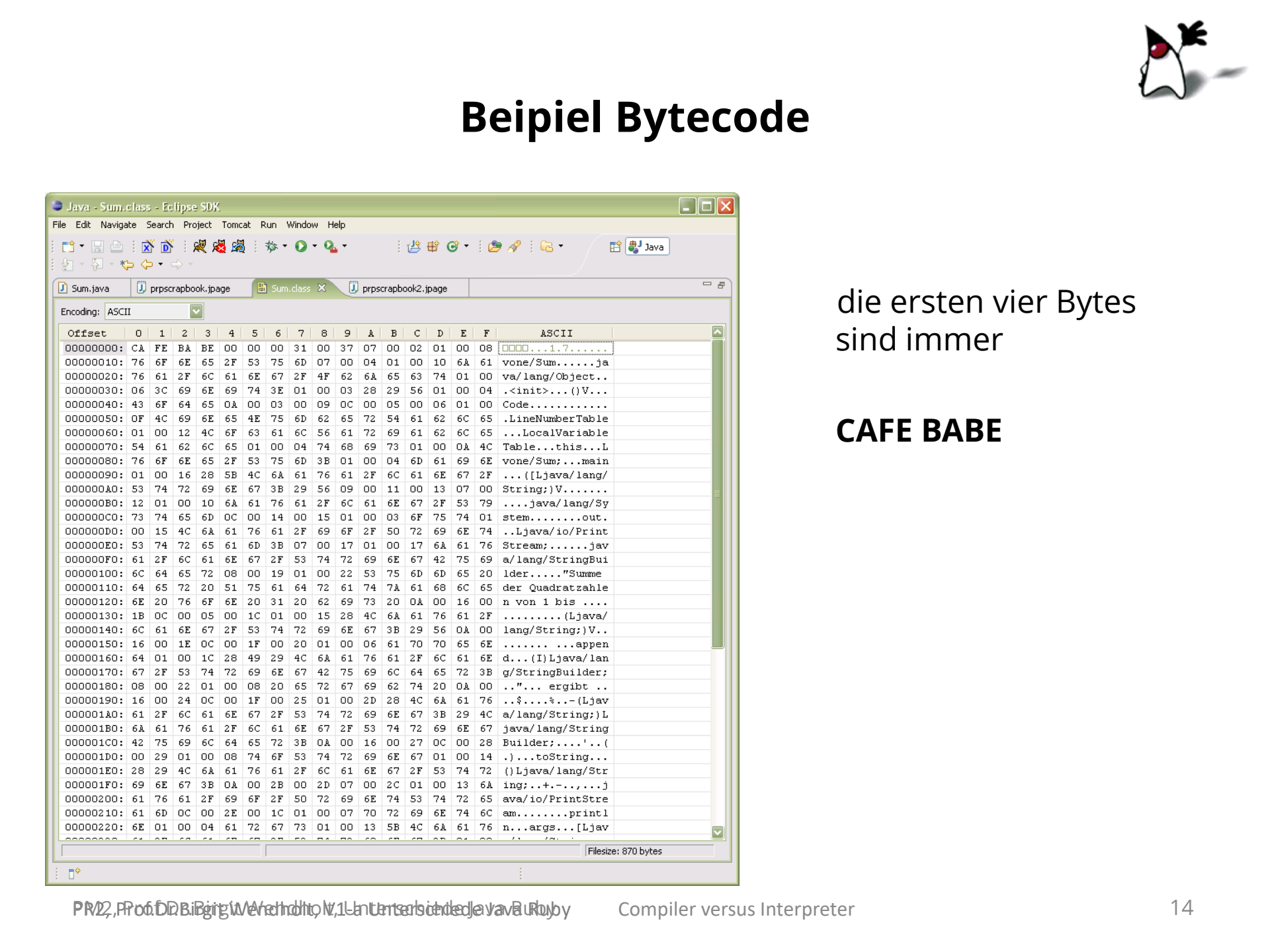
- Wir starten ein Java-Programm von der Konsole und übergeben dabei Parameter (siehe unten):
  - **Frage**: wie können wir im Java-Quelltext auf die Parameter zugreifen?
  - **Antwort**: über den Parameter der ***main*** Methode des Programms / der Klasse.

```
               0 Datei(en),              0 Bytes
              47 Verzeichnis(se), 305.567.498.240 Bytes frei

C:\Program Files>java -cp c:\temp mypack.sub.C1
C1 gefunden
c:\temp

C:\Program Files>java -cp c:\temp mypack.sub.KommandoZeilenPar
ameter 88 zz vv pp ÜWED
Programm Parameter:
0:88
1:zz
2:vv
3:pp
4:ÜWED

C:\Program Files>
```

Nützliches Programm-Parameter lesen

- Der Parameter ***args*** der ***main*** Methode mit Typ ***String[]*** (lies String Array) enthält die beim Programm-Aufruf übergebenen Werte.
- Im ***else*** Zweig werden diese über indizierten Zugriff auf das ***args*** Array ausgelesen.

```
package mypack.sub;
public class KommandoZeilenParameter {
  public static void main(String[] args) {
         if (args.length==0) {
           System.out.println("Programm wurde ohne Parameter gestartet");
         } else {
           System.out.println("Programm Parameter:");
           for (int i= 0; i < args.length; i++){
                  System.out.println(i+":"+args[i]);
           }
         }
  }
}
```



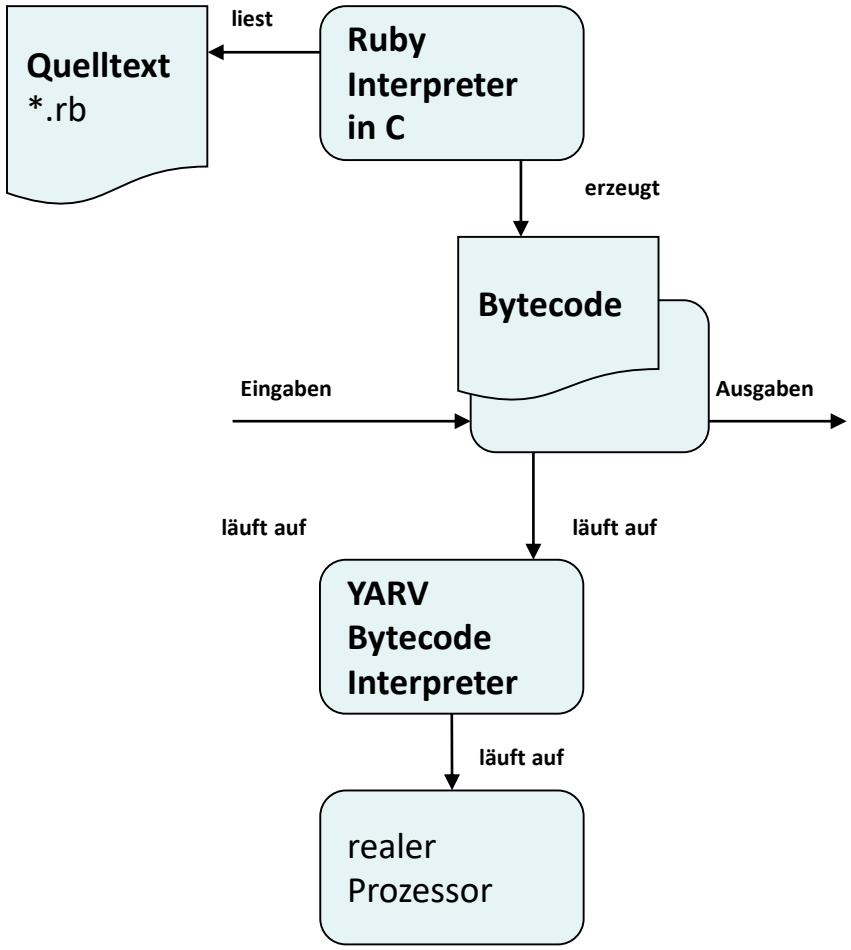
Beipiel Bytecode

die ersten vier Bytes sind immer

CAFE BABE

Ruby Interpreter

- **Ruby Interpreter in C**
  - liest Ruby Quellcode und übersetzt in Bytecode.
  - Bei der Übersetzung werden keine Sprachregeln geprüft.
- **YARV Ruby Bytecode Interpreter**
  - liest Bytecode und führt diesen dabei Zeile für Zeile aus.
  - Fehler treten nur bei der Ausführung des Bytecodes auf, also nur zur Laufzeit.



Java ist eine statisch typisierte Sprache

Java ist eine statisch typisierte Sprache

- Variablen haben einen eigenen Typ, der bei der ersten Nennung der Variablen definiert werden muss. (Java 10: Typinferenz für lokale Variablen. Dazu später mehr.)
- Variablen sind
  - lokale Variablen in Blöcken
  - Instanz- und Klassenvariablen
  - Konstanten
  - Eingabe- und Rückgabe-Parameter von Methoden

Ruby ist eine dynamisch typisierte Sprache

- Variablen haben keinen eigenen Typ.
- Der Typ der Variable ergibt sich zur Laufzeit aus dem Typ des der Variable zugewiesenen Objektes.

Java ist eine statisch typisierte Sprache

Java: Variablen haben einen eigenen Typ

```
package figuren;
public class Kreis extends Figur {

private Punkt mitte;
private double radius;

public Kreis(Punkt mitte, double
    radius){
    super();
    this.mitte = mitte;
    this.radius = radius;
}

public boolean enthaelt(Punkt p) {
    return (mitte.abstand(p) < radius);
}
}
```

Typdefinition von Instanzvariablen, Methodenparametern und Rückgabewerten

Ruby: Variablen haben keinen eigenen Typ

```
class Kreis
  include Figur
  def initialize(mitte,radius=1)
    @mitte = mitte
    @radius = radius
  end
  def enthaelt(p)
    @mitte.abstand(p) < @radius
  end
end
```

Java: Instanz- und Klassenvariablen im Scope der Klasse definieren

Java: Instanzvariablen müssen im Body der Klasse definiert sein

```
package figuren;
public class Kreis extends Figur {

private Punkt mitte;
private double radius;

public Kreis(Punkt mitte, double
    radius){
    ...
}

public boolean in(Punkt p) {
    ...
}
}
```

mitte und radius Instanzvariablen im Scope der Klasse Kreis

Ruby: Instanzvariablen im initialize markiert durch Sonderzeichen @

```
class Kreis
  include Figur

  def initialize(mitte,radius=1)
    @mitte = mitte
    @radius = radius
  end

  def in(p)
    @mitte.abstand(p) < @radius
  end
end
```

@mitte und @radius Instanzvariablen im Scope der Methode initialize

Java: Instanz- und Klassenvariablen per default *package private*

Java:

- Instanz- und Klassenvariablen müssen explizit ***private*** deklariert werden.
- Zugriff auf private Instanz- und Klassen-Variablen dann nur durch Reader oder Writer.
- Reader werden immer durch Name des Attributs mit vorangestelltem ***get*** (***is*** bei booleschen Werten) gebildet
- Writer werden immer durch Name des Attributs mit vorangestellten ***set*** gebildet

Ruby:

- Instanz- und Klassenvariablen sind immer ***private***.
- Zugriff nur mittels Readern und -Writern.

Java: Instanz- und Klassenvariablen per default *package private*

Java:

```
public class Punkt {
  private double y;
  private double x;
  public Punkt(double x, double y){
    this.x = x;
    this.y = y;
  }
  public double getY() {
    return y;
  }
  public void setY(double y) {
    this.y = y;
  }
  // analog für x
}
```

getY und setY sind die Reader und Writer der Instanzvar y

Ruby:

```
class Punkt
    attr_accessor :x, :y
    def initialize(x,y)
      @x=x
      @y=y
    end
    def y
      @y
    end
    def y=(val)
      @y = val
    end
end
```

y und y=(val) sind die Reader und Writer der Instanzvar y

Java: direkter Zugriff auf Instanzvariablen mit *default* Sichtbarkeit

- Auf die Instanz- oder Klassen-Variablen mit ***default*** Sichtbarkeit können alle Klassen des gleichen ***package*** zugreifen.

- Da ***Punkt*** und ***Rechteck*** im selben ***package figuren*** sind, darf ***Rechteck*** in der Methode ***in*** auf die Koordinaten ***x, y*** der Punkte ***lu*** und ***p*** zugreifen.

```
package figuren;
public class Punkt {

  double y;
  double x;
}
```

```
package figuren;
public class Rechteck extends Figur
    {
  private Punkt lu;
  private double hoehe, breite;

  public boolean in(Punkt p) {
    return  (lu.x <= p.x &&
        p.x <= lu.x + breite &&
        lu.y <= p.y &&
        p.y <= lu.y + hoehe);
}
```

Java: *packages* sind Namensräume für Klassen

Java

- Klassen gleichen Namens in unterschiedlichen ***packages*** sind unterschiedliche Klassen.
- ***figuren.Kreis*** ist eine andere Klasse als ***geometrie.Kreis.***
- Der vollständige Name einer Klasse ist der Name der ***Klasse*** mit dem ***package*** Präfix. Der vollständige Name von ***Kreis*** ist ***figuren.Kreis*** .
- Klassen desselben ***package*** sehen sich gegenseitig und müssen sich nicht gegenseitig importieren.
- Klassen desselben ***package*** haben besondere Zugriffrechte für Attribute mit ***default*** Sichtbarkeit.
- Klassen aus anderen ***packages*** müssen importiert werden.

Ruby

- Ruby kennt keine ***packages.***
- Klassen in unterschiedlichen Dateien gleichen Namens sind dieselben Klassen.
- Die Definition einer Klasse wird jedes Mal verändert oder erweitert, wenn die Klasse aus unterschiedlichen Dateien geladen wird.
- Jede Klasse muss sicherstellen, dass alle Definitionen geladen sind, die die Klasse in ihrer Implementierung verwendet. Dies geschieht über die ***require*** Direktive zu Beginn eines Skriptes. Dies gilt auch für Klassen im gleichen Verzeichnis.

Java: *packages* sind Namensräume für Klassen

Am Anfang einer Datei muss das *package* der Klasse definiert sein.
Hier *package figuren*
der Klassen *Punkt* und *Rechteck*

```
package figuren;
public class Punkt {

  double y;
  double x;
}
```

```
package figuren;
public class Rechteck extends Figur
    {
  private Punkt lu;
  private double hoehe, breite;

  public boolean in(Punkt p) {
    return  (lu.x <= p.x &&
          p.x <= lu.x + breite &&
          lu.y <= p.y &&
           p.y <= lu.y + hoehe);
}
```

Klassen desselben *package* müssen sich nicht gegenseitig importieren. Klasse *Rechteck* kann Klasse *Punkt* direkt verwenden, ohne zu importieren.

Ruby kennt keine *packages*

- Die Definition einer Klasse wird jedes Mal verändert, wenn die Klasse aus unterschiedlichen Dateien geladen wird.

Mit require "Object" wird die Definition der Klasse Object aus dem lokalen Script Object.rb gelesen.

Die Definition erweitert die Klasse Object um die Methode abstract.

```
require "Object"
module Figur
  def in(p)
    abstract
  end
end
```

```
class Object
  def abstract()
    raise AbstractMethodError
  end
end

class AbstractMethodError <
        StandardError
end
```

Objektinitialisierung in Java mit Konstruktoren

Java: Konstruktoren haben immer den Namen der Klasse.

```
package figuren;
public class Kreis extends Figur {
  private Punkt mitte;
  private double radius;
  public Kreis(Punkt mitte,double radius){
    super();
    this.mitte = mitte;
    this.radius = radius;
  }
}
public class Punkt {
  protected double y;
  protected double x;
  public Punkt(double x, double y){
    this.x = x;
    this.y = y;
  }
}
```

Kreis und Punkt Initialisierung

Ruby: Alle Objekte werden im *initialize* initialisiert.

```
class Kreis
  include Figur
  def initialize(mitte,radius=1)
    @mitte = mitte
    @radius = radius
  end
end


class Punkt
    attr_reader :x, :y
    def initialize(x,y)
      @x=x
      @y=y
    end
end
```

Kreis und Punkt Initialisierung

Defaultwerte für Methodenparameter

Java: keine Defaultwerte aber überladene Methoden

- Java kennt keine Defaultwerte.
- Defaultwerte lassen sich in Java bedingt mit ***überladenen*** Methoden nachbilden.
- ***Überladene*** Methoden sind Methoden gleichen Namens und gleichem Rückgabetyp aber mit unterschiedlichen Parameterlisten in Länge oder Typ der Parameter
- Überladene Methoden werden in Ruby nicht unterstützt.

Ruby: Defaultwerte, keine überladenen Methoden

- Im Beispiel wird im ***initialize*** der Kreisradius mit Default 1 belegt.

```
class Kreis
  include Figur
  def initialize(mitte,radius=1)
    @mitte = mitte
    @radius = radius
  end
end
```

Simulieren von Defaultwerten mit überladenen Konstruktoren in Java

Java

```
public class Kreis extends Figur {

  private Punkt mitte;
  private double radius;

  public Kreis(Punkt mitte, double
    radius){
    super();
    this.mitte = mitte;
    this.radius = radius;
  }

  public Kreis(Punkt mitte) {

    this(mitte,1);
  }
}
```

Ruby:

```
class Kreis
  include Figur
  def initialize(mitte,radius=1)
    @mitte = mitte
    @radius = radius
  end
end
```

Kreis hat zwei Konstruktoren mit unterschiedlich langen Parameterlisten.

this(mitte,1) im einstelligen Konstruktor ruft den zweistelligen Konstruktor mit dem Default *1* für den Radius auf.

this – super versus self - super

Java

- ***this*** ist wie ***self*** in Ruby.
- ***this(mitte, 1)*** ist eine Kurzform für: Rufe den Konstruktor mit demselben Namen und den gegebenen Parametern auf. Nur im Konstruktor zulässig.
- ***super*** ist die Referenz auf den Objektkontext der Superklasse.
- ***super()*** im Konstruktor ruft den Konstruktor der Superklasse ohne Argumente auf. ***super(...)*** mit mehr als einem Argument ist zulässig. Nur im Konstruktor zulässig.

Ruby

- ***self*** ist die Referenz auf das aktuelle Objekt, das ist das Objekt, das die Methode, in der ***self*** auftritt, ausführt.
- keine Entsprechung.
- ***super*** ruft die aktuelle Methode mit den gleichen Parameter in der Enclosing-Class auf.
- ***super()*** verhält sich wie ***super***. ***super (...)*** mit Argumenten ruft die aktuelle Methode mit den übergebenen Argumenten in der enclosing class auf.

Programmausführung

Java: Nur Klassen mit *main* sind ausführbar

Ruby: Jedes Script ist ausführbar.

```
package figuren;

public class FigurenTest {

public static void main(String[]
    args) {
    Rechteck r1;

    r1 = new Rechteck(new Punkt(3,
    3));

    System.out.println(r1);

    Punkt punkt = new Punkt(3, 4);
    System.out.println(r1.in(punkt));
}
}
```

Genauer: Nur Klassen, die die public static void main Methode enthalten, sind ausführbar. main ist eine Klassenmethode, was durch das Schlüsselwort static erreicht wird.

Objekte als String darstellen

Java: toString

- ***System.out.println*** ruft die Methode ***toString*** eines Objektes auf und gibt den Ergebnisstring auf der Konsole aus.

- Die Default-Implementierung von ***toString*** ist in der Klasse ***Object*** definiert und gibt die Klasse sowie die Objektidentifikation aus.

- Objekte, die eine lesbare Darstellung benötigen, müssen ***toString*** überschreiben.

Ruby: to_s, inspect

- ***puts*** ruft ***to_s*** auf den Objekten auf, und gibt den Ergebnisstring auf der Konsole aus.
- ***p*** ruft ***inspect*** auf den Objekten auf, und gibt den Ergebnisstring auf der Konsole aus.

- Die Default-Implementierung von ***to_s*** und ***inspect*** ist in der Klasse ***Object*** definiert.

- Objekte, die eine lesbare Darstellung benötigen, müssen ***to_s*** überschreiben.

Objektausgabe

Java: Ausgabe von r1

```
package figuren;

public class FigurenTest {

public static void main(String[]
    args) {
    Rechteck r1;

    r1 = new Rechteck(new Punkt(3,
    3));

    System.out.println(r1);

    Punkt punkt = new Punkt(3, 4);
    System.out.println(r1.in(punkt));
}
}
```

Ergebnis der Ausgabe von r1

```
r((3,0,3,0),1,0,1,0)
```

Dafür haben wir zuvor die toString Methode in Rechteck überschrieben.

Objekte als String darstellen

Java: toString

```
package figuren;
public class Rechteck extends Figur
    {

  private Punkt lu;
  private double hoehe, breite;
  public String toString() {
     return  "r("+ lu + "," +
                    hoehe + "," +
                breite+ ")";
   }
}
```

Keine Auswertung von Ausdrücken in Strings. Daher mühsames Konkatenieren der Bestandteile (Operator "+").

Ruby: to_s

```
class Rechteck
  include Figur
  def to_s
    return
     "r(#{@lu},#@hoehe,#{@breite})"
  end
```

Durch Auswertung von Ausdrücken in Strings ist die Implementierung in Ruby kompakter und weniger schreibintensiv.

Kompaktere Schreibweise in Java

Java: toString

```
package figuren;

public class Rechteck extends Figur
    {

  private Punkt lu;
  private double hoehe, breite;

  public String toString() {
    return
    String.format("r(%s,%.1f,%.1f)",
          lu.toString(),hoehe,breite);
}
}
```

Mit *String.format* wird auch die Darstellung in Java kompakter. Der Preis: das Erlernen von Formatanweisungen (z.B *%.1f* für die Darstellung einer Dezimalzahl mit einer Nachkommastelle).

Ruby: to_s

```
class Rechteck
  include Figur
  def to_s
    return
     "r(#{@lu},#@hoehe,#{@breite})"
  end
```

Durch Auswertung von Ausdrücken in Strings ist die Implementierung in Ruby kompakter und weniger schreibintensiv.

Methoden "ohne" Ergebnis

Java: Methoden ohne Ergebnis werden mit *void* im Ergebnisparameter markiert.

```
package figuren;

public class FigurenTest {

  public static void main(String[]
   args) {
    Rechteck r1;

    r1 = new Rechteck(new Punkt(3,
   3));

    System.out.println(r1);

    Punkt punkt = new Punkt(3, 4);
    System.out.println(r1.in(punkt));
  }
}
```

void ist eine reserviertes Wort der Sprache Java (stammt aus C) und kein Objekt.

Ruby: Jede Methode hat ein Ergebnis. Ein nicht spezifisches Ergebnis ist *nil*, das Objekt, das für „Nichts“ steht.

```
p u5.enthaelt(punkt)
puts u4.enthaelt(punkt)
```

p und puts liefern nil.

Java ist eine statisch typisierte Sprache

Java: statisch typisiert

- Der Compiler prüft zur Übersetzungszeit des Programms, ob der Typ der Variable und der Typ des Objektes, das der Variable zugewiesen wird, kompatibel sind.
- Zur Laufzeit sollten daher in der Regel keine Typfehler auftreten. *Es gibt Ausnahmen (später → casten)*

Ruby: dynamisch typisiert

- Typprüfungen finden nur zur Laufzeit statt.
- Typfehler treten daher auch nur zur Laufzeit auf.

Java ist eine statisch typisierte Sprache

Java: Typprüfungen durch den Compiler bei der Übersetzung

```
package figuren;
public class Kreis extends Figur {
    public boolean in(Punkt p) {
          ...
    }
}


Kreis  k2 =
    new Kreis(new Punkt(2, 2), 3);
k2.in(4);
```

Compilerfehler: Argument 4 beim Aufruf der Methode k2.in(4), ein int ist nicht typkompatibel mit Punkt.

➔ Programm wird nicht übersetzt.

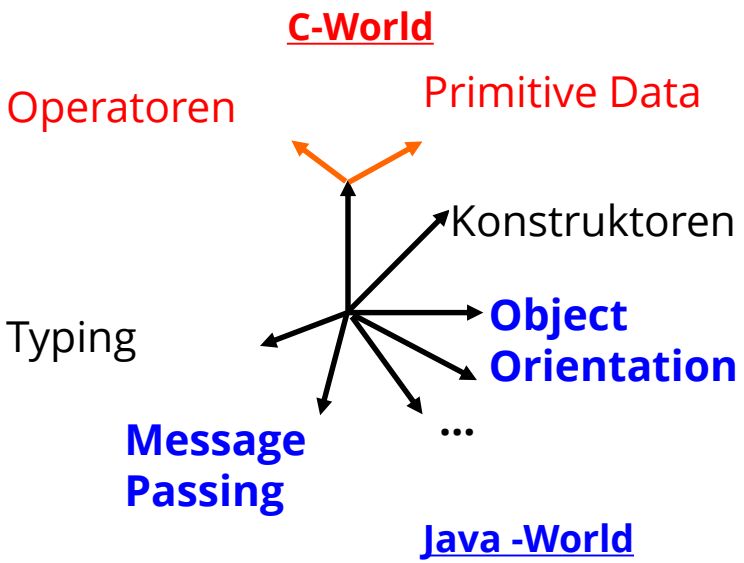
Ruby: Typprüfungen zur Laufzeit

```
class Kreis
  include Figur
  def initialize(mitte,radius=1)
  end
  def in(p)
    @mitte.abstand(p) < @radius
  end
end

k2 = Kreis.new(Punkt.new(2,2),3)
k2.in(4)
```

Laufzeitfehler: Wenn k2.in(4) ausgeführt wird, erzeugt die Methode abstand einen Laufzeitfehler, da 4 keine x und y Koordinaten hat.

Nicht alles in Java ist objektorientiert

- Basisdatentypen (primitive data) sind keine Objekte
- Operatoren sind keine Methoden.
- Konstruktoren sind nicht polymorph und werden nicht vererbt.
- statische Typisierung
  - Variablen haben Typen
  - Compiler prüft die Kompatibilität von Variablentypen mit den Typen der zugewiesenen Objekte / Basisdatentypen



Datentypen

Java

Typen		**"Instanzen"**
Basisdatentypen		Werte
Referenztypen	Arraytypen	Arrays
	Objekttypen (Klassen)	Objekte

Ruby

Java unterscheidet Basisdaten-, Referenztypen (Arraytypen haben eine besondere Syntax und Semantik).

Für Basisdatentypen gibt es eine Reihe fest definierter Typbezeichnungen
Für Basisdatentypen gibt es nur Operatoren und keine Methoden.

Ruby kennt nur Objekttypen.
Basisdatentypen und Arraytypen sind in Ruby Klassen.

Datentypen

Java

Typen		**"Instanzen"**
Basisdatentypen		Werte
Referenztypen	Arraytypen	Arrays
	Objekttypen (Klassen)	Objekte

Ruby

- Die Instanzen der Basisdatentypen sind Werte, die typischerweise in einem Speicherwort (byte, bit) repräsentiert werden.
- Werte sind unveränderlich.
- Operationen auf Werten erzeugen neue Werte.

Auch wenn Zahlen in Ruby Objekte sind, sind diese unveränderlich.

Basisdatentypen

```
package figuren;

public class Kreis extends Figur {

private Punkt mitte;
private double radius;

public Kreis(Punkt mitte, double radius){
    super();
    this.mitte = mitte;
    this.radius = radius;
}
```

Für Basisdatentypen gibt es eine Reihe fest definierter Typbezeichnungen , die immer klein geschrieben werden (hier *double*).

Basisdatentypen

```
public double abstand(Punkt p) {
    if ((p.x -x) < 10e-5){
          return Math.abs(p.y -y);
    }
    return Math.sqrt(Math.pow(p.y-y,2) + Math.pow(p.x -x, 2));
}
```

Für Basisdatentypen gibt es nur Operatoren.

Operatoren sind keine Methoden und nur für Basisdatentypen definiert.

Arraytypen – dazu später mehr

Java

Typen		**"Instanzen"**
Basisdatentypen		Werte
Referenztypen	Arraytypen	Arrays
	Objekttypen (Klassen)	Objekte

Ruby

- Arraytypen sind in Java Klassen mit einer speziellen Syntax für die Typdefinition / den Elementzugriff und -zuweisung
- Arrays haben eine feste unveränderliche Länge.

- Der Arraytyp ist eine Klasse in Ruby, die wie alle Klassen modifiziert und durch Subklassen erweitert werden können
- Arrays haben keine feste Länge. Sie verhalten sich wie Listen in Java.

Zusammenfassung

Java	Ruby
statisch typisierte Sprache. Variablen haben einen Typ.	dynamisch typisierte Sprache
Instanz- und Klassenvariablen sind im Scope der Klassendefinition enthalten.	Instanzvariablen sind im initialize durch Sonderzeichen **@** markiert.
Instanz- und Klassenvariablen sind per default ***package private***.	Instanz- und Klassenvariablen sind immer private.
Reader und Writer sind ***get***... und ***set***... Methoden.	Zugriff auf Instanz- und Klassenvariablen nur mittels Reader und -Writer.
packages fassen Klassen in einem Namensraum zusammen.	keine Entsprechung.
Objektinitialisierung über Konstruktoren.	Objektinitialisierung immer in der ***initialize*** Methode.
keine Defaultwerte aber überladene Methoden	Defaultwerte, keine überladenen Methoden

Zusammenfassung

Java	Ruby
this	***self***
super ist die Referenz auf den Objektkontext der Superklasse. ***super***() im Konstruktor ist ein Konstruktoraufruf	***super*** ist eine Methodenaufruf ***super***() ist dasselbe wie super
nur Klassen mit ***main*** sind ausführbar.	jedes Script ist ausführbar.
main ist eine statische Klassenmethode. Klassenmethoden werden mit ***static*** markiert.	Klassenmethoden werden durch Referenz auf des Klassenobjekt definiert.
Ausgabe auf der Konsole: ***System.out.println***	Ausgabe auf der Konsole: ***puts, p,*** ...
Objektausgabe als String verwendet ***toString***	Objektausgabe als String verwendet ***to_s***, ***inspect***
Keine Auswertung von Ausdrücken in Strings.	Auswertung von Ausdrücken in double-quoted Strings.

Zusammenfassung

Java	Ruby
Abstrakte Klassen und Methoden sind Bestandteil der Sprache.	Module sind ein Äquivalent für abstrakte Klassen.
Der Java Compiler prüft u. A. Syntaxregeln und Typregeln vor der Übersetzung in Bytecode.	Der Rubyinterpreter führt alle Prüfungen zur Laufzeit durch.
Methoden müssen ihr Ergebnis explizit mit return zurückgeben.	Das Ergebnis einer Methode ist der Wert der letzten Anweisung einer Methode
Methoden ohne Ergebnis werden mit void im Ergebnisparameter markiert.	Jede Methode hat ein Ergebnis.
Basisdaten-, Array- und Objekttypen.	Objekttypen.
Operatoren sind keine Methoden und nur für Basisdatentypen definiert (Ausnahmen + für String) Operatoren können nicht in eigenen Klassen überschrieben werden.	Ruby kennt nur Methoden. + kann in eigenen Klassen überschrieben werden ebenso = (Zuweisung)

Und was noch?

Java	**Ruby**
jedes Statement endet mit ***";"*** Jede Klassen und Methodendefinition ist in geschweiften Klammern eingeschlossen.	Syntax ist freier, aber auch variantenreicher.
keine Mehrfachvererbung der Imple-mentierung (vor Java 8 / mit Java 8 wird das aufgeweicht)	Mehrfachvererbung der Implementierung durch Mixins
private, protected, public und ***package*** private Sichtbarkeiten	nur ***private, protected, public***
private Sichtbarkeit erlaubt Zugriff auf andere Objekte desselben Typs	***private*** Sichtbarkeit erlaubt nur Zugriff auf ***self***
Strings sind immutable.	Strings sind mutable.
umfangreiche Collection Klassen	Collection Klassen: ***Array, Set, Hash***
Arrays sind Builtin-Typen mit fester Länge	Arrays sind erweiterbare Typen variabler Länge.
generische Typen (Generics)	kein Äquivalent
equals prüft Inhaltsgleichheit	***eql?*** prüft Inhaltsgleichheit
== prüft Identität	***equal?*** prüft Identität
Jede public Klasse benötigt eine eigene Datei	Beliebig viele Klassen in einem Ruby Script

Testen

Java - JUnit

- Testklasse ist eine normale Klasse.
- Fixture wird mit ***@Before*** annotiert.
- Testmethoden werden mit ***@Test*** annotiert.
- Assertions werden statisch importiert.
- ***assertTrue*** oder ***assertFalse***

Ruby -RUnit

- Testklasse leitet von ***Test::Unit::TestCase*** ab.
- Fixture in der ***setup*** Methode.
- Testmethoden beginnen mit ***test.***
- Assertions werden über ***test/unit*** gela-den.
- ***assert***

Testen mit JUnit

```
package figuren;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class FigurenTestJUnit {
    Rechteck r1;
    Punkt punkt;

    @Before
     public void fixture(){
          r1 = new Rechteck(new Punkt(3,3));
          punkt = new Punkt(3,4);
    }

    @Test
    public void punktIn() {
          assertTrue(r1.in(punkt));
    }
}
```

- Statisches Importieren der Assertions
- Annotationen repräsentieren Klassen

- Fixture

- Testmethode
- Assertion assertTrue

Uebungen:

- **ue-1-1**:
  - Implementieren Sie das Figurenbeispiel mit Tests vollständig in Java nach.
  - Verwenden Sie die Übersetzungen zwischen Ruby und Java, die in der Vorlesung vorgestellt wurden. Diese reichen aus.

- **ue-1-2**:
  - Kopieren Sie die Package-Struktur des ***bin*** Verzeichnisses der Lösung zu **ue-1-1** in ein neues Verzeichnis. Starten Sie das Programm von der Konsole und übergeben Sie dabei den korrekten ***CLASSPATH***.