Einführung in die Informatik

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Sommersemester 2014

Konstruktoren

Konstruktoren

Bisher haben wir neue Objekte so erzeugt:

```
Test object = new Test();
```

Tatsächlich ist Test eine Konstruktor, d.h. eine spezielle Methode.

Standardmäßig ist die Methode leer, man kann aber eine eigene schreiben:

```
public class Test {
    private int x;
    public Test() {
        x = IOTools.readInteger("x:");
    }
    public static void main(String[] param) {
        Test object = new Test();
        System.err.println(object.x);
    }
}
```

Klasseninitialisierung

Die Aufgabe eines Konstruktors ist es die Klasse zu initialisieren.

- Jedes Objekt muss über einen Konstruktor erzeugt werden.
- Der Konstruktor initialisiert zunächst alle Felder.
- Dann wird der Programmcode im Konstruktor ausgeführt.

Konstruktoren

```
public class Test {
    private int x;
    public Test() {
        x = IOTools.readInteger("x:");
    }
    ...
}
```

- Konstruktoren haben keinen Rückgabetyp (nicht einmal `void`).
- Konstruktoren können `public` oder `private` sein. Man kann so verhindern, dass andere Programmteile Objekte erzeugen.
- Konstruktoren können auch Parameter haben.
- Eine Klasse kann mehrere Konstruktoren haben.
- Diese können sich gegenseitig mit `this()` aufrufen, aber nur in der ersten Zeile!

Beispiel

```
public class Value {
    public int v;
    public Value() {
        this(-1);
    }
    public Value(int i) {
        v = i;
    }
}
```

- Der zweite Konstruktor kann zum Beispiel mit `x = new Value(5)` aufgerufen werden.
- Der erste Konstruktor kann mit `x = new Value()` aufgerufen werden.
- Der erste Konstruktor ruft den zweiten mit `-1` auf.

Initialisierte Felder

Man kann den Komponenten Initialwerte geben:

```
public class Car {
    public String model;
    public int tires = 4;

    public Car(String model, int tires) {
        this.model = model;
        this.tires = tires;
    }
    public Car(String model) {
        this.model = model;
    }
}
```

- Wird eine Variable nicht initialisiert, hat sie den Wert 0 (oder null).
- Die Variable wird am Anfang des Konstruktors initialisiert.

Erweiterte Initialisierung

Man kann sogar beliebigen Code hinzufügen, der am Anfang jedes Konstruktors ausgeführt wird.

```
public class Car {
    public String model;
    public int tires = 4;
    {
        System.out.println("model: " + model
            + " tires: " + tires);
    }

    public Car(String model, int tires) {
        this.model = model;
        this.tires = tires;
    }
    public Car(String model) {
        this.model = model;
    }
}
```

Der this()-Aufruf

Im Allgemeinen ist es lesbarer gemeinsame Initialisierungen explizit in einen Konstruktor zu schreiben. Die anderen Konstruktoren können diesen dann aufrufen.

```
public class Car {
    public String model;
    public int tires = 4;

    public Car(String model, int tires) {
        this(model);
        this.tires = tires;
    }
    public Car(String model) {
        System.out.println("model: " + model
            + " tires: " + tires);
        this.model = model;
    }
}
```

Statische Initialisierung

Auch statische Felder können Initialwerte haben. Man kann auch hier beliebigen Code einfügen.

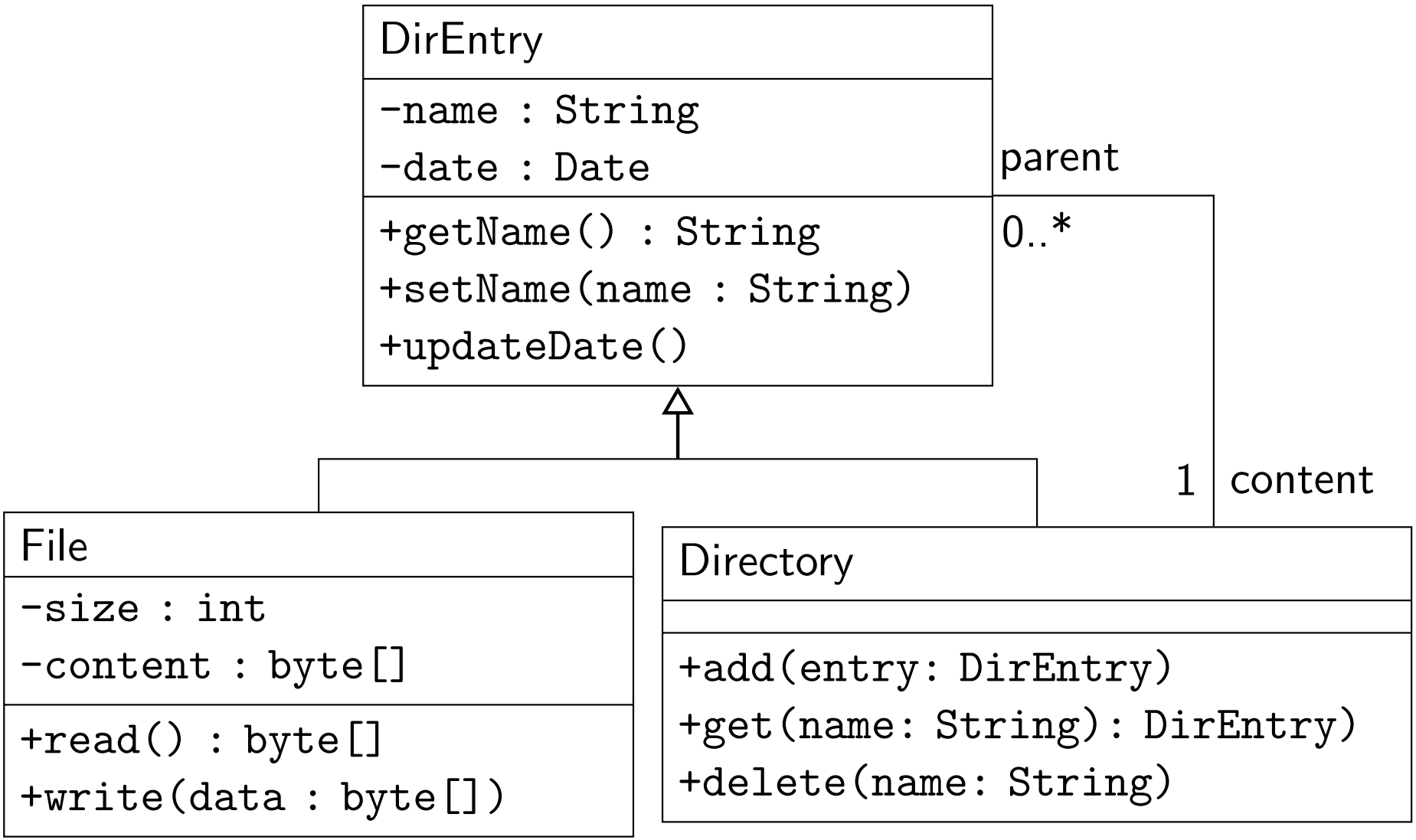
```
public class Car {
    final static Car PHANTOM = new Car();
    static {
        PHANTOM.tires = 0;
    }

    ...
}
```

Achtung

Ein `static`-Block oder ein Initialwert kann Seiteneffekte haben, die andere Klassen laden und deren `static`-Blöcke ausführen. Dann kann man sich nicht auf die Reihenfolge verlassen.

Vererbung

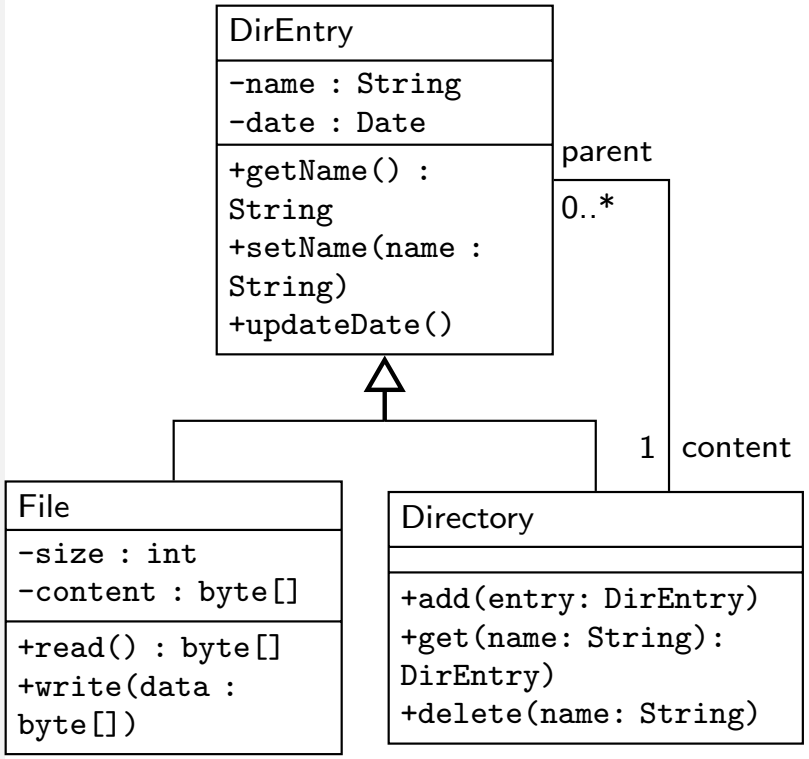
Vererbung in Java



Vererbung in Java

In Java wird die Vererbung mit dem Schlüsselwort `extends` ausgedrückt.

```
public class DirEntry {
    private String name;
    public String getName()
       ...
}

public class File extends DirEntry {
    private byte[] content;
    ...
}

public class Directory extends
    DirEntry {
    private DirEntry[] content;
    ...
}
```



Wann benutzt man Vererbung?

- `File extends DirEntry`, heißt dass `File` von `DirEntry` erbt.
- Man sollte es nur benutzen, wenn es eine "ist ein"-Beziehung gibt:
  Eine Datei ist ein Directoryeintrag.
  Ein Verzeichnis ist ein Directoryeintrag.
- Die Subklasse (die erbende Klasse) ist *spezieller* als die Superklasse.
- Jeder Code der einen Directoryeintrag erwartet, kann jetzt mit einer Datei oder einem Verzeichnis aufgerufen werden.

Automatische Typkonvertierung

Eine Datei ist ein Directoryeintrag. Daher kann man folgendes schreiben:

```
Directory root = new Directory();
Directory home = new Directory();
// Directory erbt Methode von DirEntry
home.setName("home");
// root.add erwartet DirEntry, aber
// es gibt automatische Konvertierung
root.add(home);
DirEntry entry = home;      // okay
//Directory home = entry; // Compiler-Fehler
Directory home = (Directory) entry; //okay
File home = (File) entry; // Laufzeitfehler
```

Die Klasse `Object`

Jede Klasse erbt automatisch von `java.lang.Object`.

- `Object` ist die Superklasse von allen Klassen und sogar von Feldern (Arrays).
- `Object` definiert ein paar Methoden, die jedes Objekt hat, zum Beispiel
  - `String toString()`: gibt eine lesbare Repräsentation des Objekts aus.
  - `boolean equals(Object o)`: testet ob zwei Objekte gleich sind.
  - `Class getClass()`: liefert den Laufzeittyp des Objekts.

Compile-Typ und Laufzeittyp

Weil ein `File` ein `DirEntry` ist, ist die folgende Zuweisung okay:

```
DirEntry entry = new File();
```

Die Variable `entry` hat den Typ `DirEntry`. Sie speichert eine Referenz auf ein Objekt, das den Typ `DirEntry` hat. `DirEntry` ist der Compile-Typ, denn er steht schon beim Kompilieren des Programms fest.
Zur Laufzeit kann die Variable aber eine Referenz auf ein Objekt vom Typ `File` enthalten.

- Der Laufzeittyp muss nicht mit dem Compile-Typ übereinstimmen.
- Der Laufzeittyp ist immer eine Subklasse vom Compile-Typ.

Der Operator `instanceof` kann benutzt werden, um den Laufzeittyp zu überprüfen. Außerdem hat jedes Objekt die Methode `getClass()`, die den Laufzeittyp zurück gibt.

Der Operator `instanceof`

Der Operator `instanceof` erwartet einen Ausdruck (vom Typ `Object`) und einen Klassentyp:

Syntax

Ausdruck ::= *Ausdruck* `instanceof` *Klassentyp*

Der Operator liefert ein Boolesches Ergebnis:

- `true`, falls der Laufzeittyp ein Subtyp von dem Klassentyp ist.
- `false` sonst, oder falls der Ausdruck `null` ist.

Der Klassentyp kann auch ein Feldtyp (array type) sein.

Explizite Typkonvertierung

Weil ein Directoryeintrag eine Datei oder ein Verzeichnis sein kann, erlaubt der Compiler Typkonvertierungen.
Mit `instanceof` kann man sich vergewissern, ob ein Objekt einen Typ hat.

```
void test(DirEntry entry) {
    if (entry instanceof Directory) {
        Directory dir = (Directory) entry;
        ...
    } else if (entry instanceof File) {
        File file = (File) entry;
        ...
    }
}
```

Wenn bei einer Typkonvertierung der Typ nicht stimmt, stürzt das Programm ab (ClassCastException).

Konstruktoren in Subklassen

```
public class DirEntry {
    String name;
    public DirEntry(String name) {
      this.name = name;
    }
}

public class File extends DirEntry {
    private byte[] content;
    public File(String name, byte[] content) {
        super(name);
        this.content = content;
    }
    public File(String name) {
        this(name, new byte[0]);
    }
}
```

Konstruktoren und Vererbung

Jeder Konstruktor muss entweder

- einen anderen Konstruktor der gleichen Klasse `this(...)` aufrufen,
- oder einen Konstruktor der Superklasse `super(...)` aufrufen.
- Wenn es nicht explizit angegeben ist, wird `super()` aufgerufen.

Initialisierung von Klassen

Die Initialisierung läuft wie folgt:

1. Das Objekt wird mit Nullen initialisiert,
2. der Super-Konstruktor wird aufgerufen,
3. die Felder werden initialisiert und expliziten Initialisierungen werden ausgeführt.
4. der eigentliche Konstruktor wird ausgeführt.

Ruft ein Konstruktor einen anderen Konstruktor mit `this(...)` auf, kümmert sich der aufgerufene Konstruktor um den zweiten und dritten Punkt.

Das Schlüsselwort `final`

Mit dem Schlüsselwort `final` hat drei Bedeutungen:

- Vor einer Klasse (z.B. `public final class String`) verbietet es Subklassen zu erzeugen.
  Weil die Klasse `String` final ist, gibt es einen Compiler-Fehler, wenn man von dieser Klasse ableitet.
- Vor einer Methode verbietet es Subklassen, diese Methode zu überschreiben.
- Vor einer Komponente verbietet es den Wert zu ändern. Damit werden Konstanten definiert (z.B.
  `public static final double PI=3.141592653589793`).
  Konstanten sind meist statisch. Nicht statische final Komponenten kann man benutzen, um ein Objekt unveränderlich (immutable) zu machen. `String` ist ein Beispiel für ein unveränderliches Objekt.

Polymorphie

Motivation

Wir wollen unsere Verzeichniseinträge mit Icons ausstatten

- Es soll eine Methode geben, die das Icon zurückgibt.
- Dateien haben andere Icons als Verzeichnisse.
- Wenn man den Inhalt eines Verzeichnis ausgibt, will man keine Fallunterscheidung machen.

Datei-Icons

```
public class DirEntry {
    public Icon getIcon() {
        return UNKNOWN;
    }
}
public class Directory
    extends DirEntry {
    public Icon getIcon() {
        return DIRECTORY;
    }
}
```

```
public class File
    extends DirEntry {
    public Icon getIcon() {
        return FILE;
    }
}
void test() {
    DirEntry entry =
        new Directory();
    Icon i = entry.getIcon();
}
```

Welche `getIcon()`-Methode wird in `test` aufgerufen?

Polymorphie

- Die erbende Klasse kann eine Methode der Superklasse überschreiben (Englisch: override).
- Welche Methode aufgerufen wird, hängt davon ab, welchen Laufzeittyp das Objekt hat.
- Die überschreibende Methode kann das Originalmethode aufrufen.

```
public Icon getIcon() {
  if (!hasIcon)
    return super.getIcon();
  ...
}
```

Abstrakte Methoden und Klassen

In unserem Beispiel gibt es keine Objekte vom Typ DirEntry.
Man kann das durch das Schlüsselwort `abstract` explizit machen.

```
public abstract class DirEntry {
    public abstract Icon getIcon();
}
public class Directory
    extends DirEntry {
    public Icon getIcon() {
        return DIRECTORY;
    }
}
```

- Eine abstrakte Methode darf nur in einer abstrakten Klasse sein.
- Die erbenden Klassen müssen die Methode überschreiben (es sei denn, sie sind selbst abstrakt)
- Eine abstrakte Methode hat keinen Code (Implementierung).
- `new` darf nicht auf abstrakten Klassen benutzt werden.

Interfaces (Schnittstellen)

Der Extremfall einer abstrakten Klasse ist ein Interface.

- In einem Interface sind alle Methoden abstrakt und öffentlich. Man kann `public abstract` weglassen.
- Alle Komponenten sind statisch, konstant und öffentlich (`public static final`).
- Eine Klasse kann nur von einer anderen Klasse erben, aber von beliebig viele Interfaces.

Bei einem Interface spricht man in Java nicht von erben oder erweitern, sondern von implementieren. Es wird das Schlüsselwort `implements` statt `extends` benutzt.

Wann benutzt man Interfaces?

Anwendungszwecke für Interfaces:

- Um mehrere austauschbare Komponenten zu erlauben. Alle Komponenten implementieren das gleiche Interface.
  Beispiel: Ein Schachprogramm könnte ein Interface zur Anbindung von Schach-KIs definieren. Dann kann man die KI einfach austauschen.
- Um Rückmeldungen zu geben.
  Beispiel: Eine Methode die länger läuft, könnte eine Objekt als Parameter erwarten, dass ein Interface `ProgressListener` implementiert. Diese Interface enthält eine Methode `progress(double percentage)`, die z.B. einen Fortschrittsbalken anzeigt.