

Einführung in die Informatik

Definition von Klassen

Wolfram Burgard

Motivation

- Auch wenn **Java** ein **große Zahl von vordefinierten Klassen und Methoden** zur Verfügung stellt, sind dies nur Grundfunktionen für eine Modellierung vollständiger Anwendungen.
- Um **Anwendungen** zu **realisieren**, kann man diese **vordefinierten Klassen nutzen**.
- Allerdings **erfordern** manche **Anwendungen Objekte und Methoden**, für die es **keine vordefinierten Klassen** gibt.
- **Java erlaubt** daher dem Programmierer, **eigene Klassen zu definieren**.

Klassendefinitionen: Konstruktoren und Methoden

```
class Laugher1 {  
    public Laugher1() {  
    }  
    public void laugh() {  
        System.out.println("haha");  
    }  
}
```

- Durch diesen Code wird eine Klasse `Laugher1` definiert.
- Diese Klasse stellt eine einzige Methode `laugh` zur Verfügung

Anwendung der Klasse `Laugher1`

Auf der **Basis dieser Definition** können wir ein `Laugher1`-**Objekt deklarieren**:

```
Laugher1 x;
```

```
x = new Laugher1();
```

Dem durch `x` referenzierten **Objekt** können wir anschließend die **Message** `laugh` **schicken**:

```
x.laugh();
```

Aufbau einer Klassendefinition

- Das Textstück

```
class Laugher1 {
```

- läutet in unserem Beispiel die Definition der Klasse `Laugher1` ein.
- Die Klammern `{` und `}` werden **Begrenzer** oder **Delimiter** genannt, weil sie den Anfang und das Ende der Klassendefinition markieren.
- Zwischen diesen **Delimitern** befindet sich die Definition des **Konstruktors**

```
    public Laugher1() {  
    }
```

und einer **Methode**.

```
        public void laugh() {  
            System.out.println("haha");  
        }
```

Aufbau der Methodendefinition laugh

Die Definition der Methode besteht aus einem **Prototyp**

```
public void laugh()
```

und dem **Methodenrumpf**

```
{  
    System.out.println("haha");  
}
```

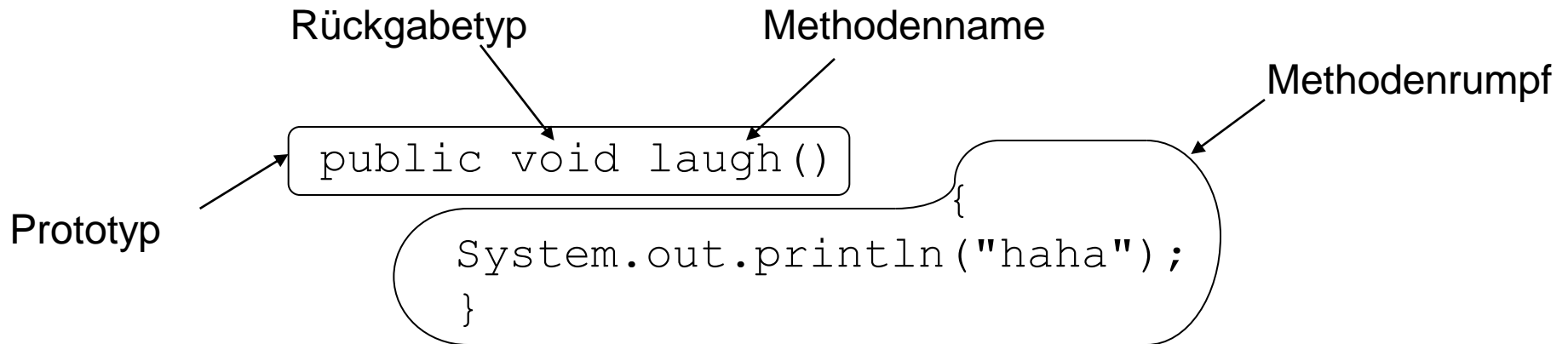
1. Der **Prototyp** der Methode beginnt mit dem **Schlüsselwort** `public`.
2. Danach folgt der **Typ des Return-Wertes**.
3. Dann wird der **Methodenname** angegeben.
4. Schließlich folgen **zwei Klammern** `()`, zwischen denen die Argumente aufgelistet werden.

Der Rumpf der Methode laugh

Der **Methodenrumpf** enthält die **Statements**, die ausgeführt werden, wenn die Methode aufgerufen wird.

Die Methode `laugh` druckt den Text "haha" auf den Monitor.

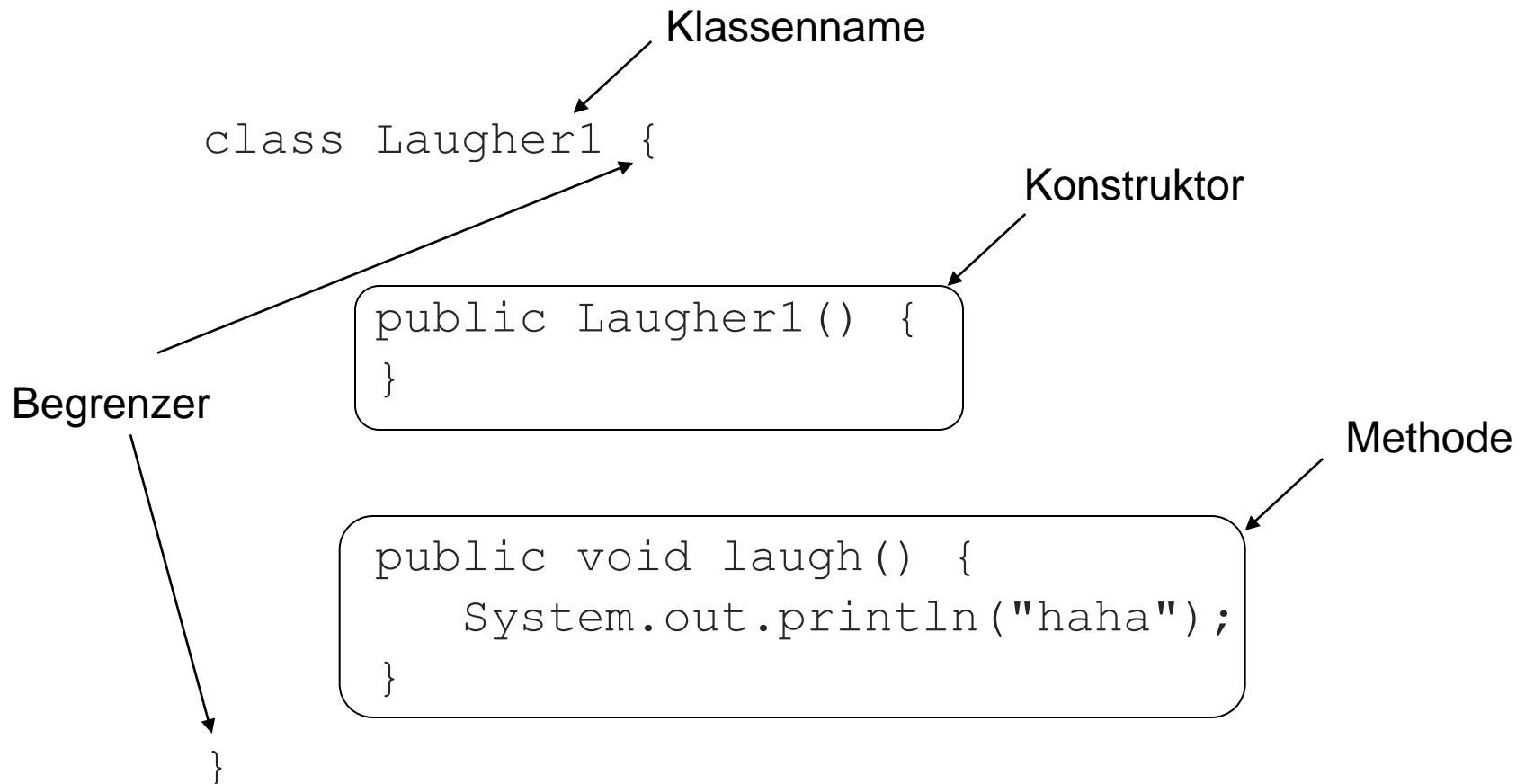
Zusammenfassend ergibt sich:



Der Konstruktor der Klasse `Laugher1`

- Die Form eines **Konstruktors** ist **identisch zu einer Methodendefinition**.
- Lediglich der **Return-Typ wird ausgelassen**.
- Konstruktoren werden immer mit dem Schlüsselwort `new` aufgerufen.
- Dieser Aufruf gibt eine **Referenz auf ein neu erzeugtes Objekt** zurück.
- Der Konstruktor `Laugher1` tut nichts.

Struktur der Klassendefinition `Laugher1`



Parameter

- In der Methode `laugh` wird der auszugebende Text vorgegeben.
- Wenn wir dem Sender einer Nachricht erlauben wollen, die Lachsilbe festzulegen, (z.B. ha oder he), müssen wir eine Methode mit **Argument** verwenden:

```
x.laugh("ha");
```

oder

```
x.laugh("yuk");
```

- **Parameter** sind **Variablen**, die im **Prototyp** einer Methode spezifiziert werden.

Definition einer Methode mit Argument

- Da unsere neue Version von `laugh` ein `String`-Argument hat, müssen wir den **Prototyp** wie folgt ändern:

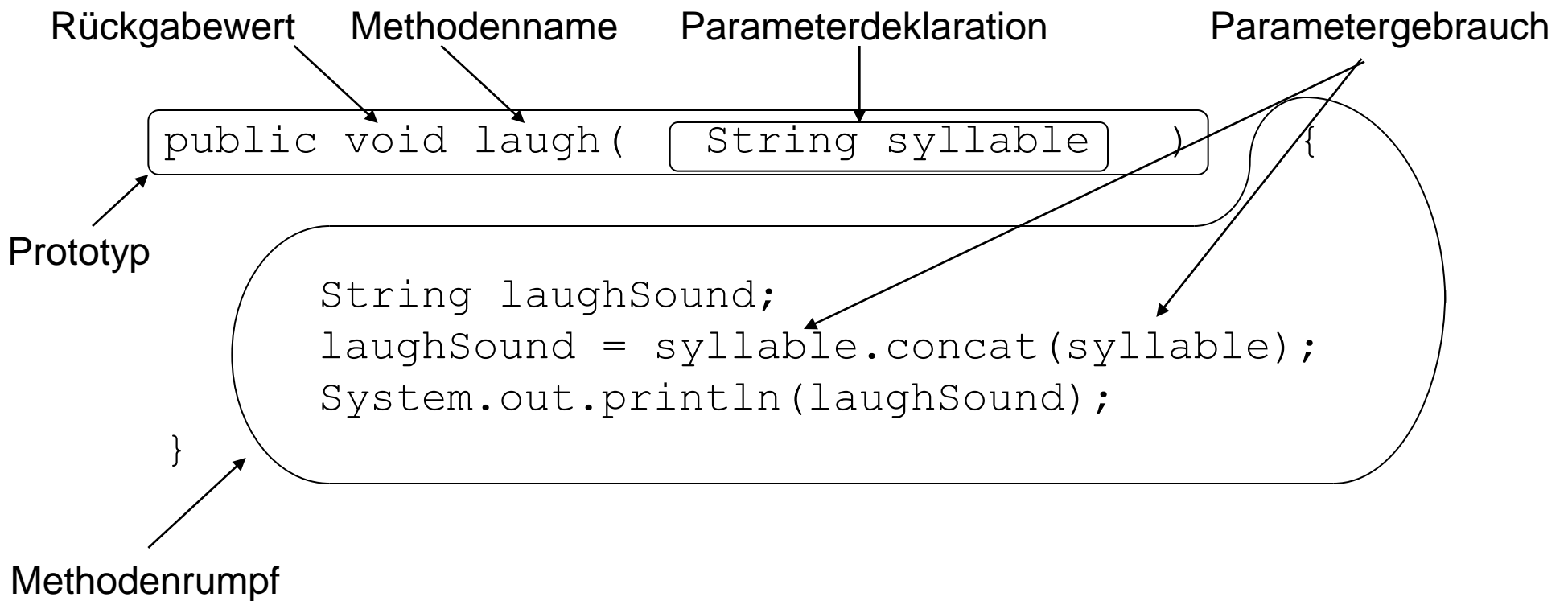
```
public void laugh(String syllable)
```

- Der Rumpf kann dann z.B. sein:

```
    {  
        String laughSound;  
        laughSound = syllable.concat(syllable);  
        System.out.println(laughSound);  
    }
```

- Wird diese Methode mit dem Argument `"ho"` aufgerufen, so gibt sie den Text `hoho` auf dem Bildschirm aus.

Struktur einer Methode mit Parametern



Eine erweiterte Laugher2-Klasse

```
class Laugher2 {
    public Laugher2() {
    }
    public void laugh() {
        System.out.println("haha");
    }

    public void laugh(String syllable) {
        String laughSound;
        laughSound = syllable.concat(syllable);
        System.out.println(laughSound);
    }
}
```

Overloading

- Diese Definition von `Laugher2` stellt **zwei Methoden** mit dem **gleichen Namen** aber **unterschiedlichen Signaturen** zur Verfügung:

```
laugh ()  
laugh (String syllable)
```

- In diesem Fall ist die Methode `laugh` **überladen** bzw. **overloaded**.
- Wenn wir `haha` ausgeben wollen, genügt der Aufruf

```
x.laugh ();
```
- Um einen anderes Lachen (z.B. `yukyuk`) zu erzeugen, verwenden wir die zweite Methode:

```
x.laugh ("yuk");
```
- Die **Methode ohne Parameter** repräsentiert das **Standardverhalten** und heißt daher **Default**.

Variante 3:

Veränderbare Standardsilbe

- Am Ende wollen wir auch die Möglichkeit haben, die Standardlachsilbe des `Laugher`-Objektes im **Konstruktor** anzugeben.
- Die gewünschte Anwendung ist:

```
Laugher3 x;  
x = new Laugher3("ho");  
x.laugh("heee");  
x.laugh();
```

- Um dies zu erreichen, erhält der Konstruktor jetzt ein `String`-Argument, so dass er folgende Signatur hat:

```
Laugher3(String s)
```

Instanzvariablen

- Wie können wir jetzt in der Methode `laugh()` auf dieses `String`-Objekt zugreifen?
- Die Lösung besteht darin, in der Klasse eine `String`-**Variable** zu definieren, die **außerhalb der Methoden der Klasse** steht.
- Eine solche Variable heißt **Instanzvariable**.
- Sie gehört zu dem **gesamten Objekt** und nicht zu einer einzelnen Methode.
- **Auf Instanzvariablen kann von jeder Methode aus zugegriffen werden.**
- **Instanzvariablen werden genauso deklariert wie andere Variablen.** In der Regel geht der Deklaration jedoch das **Schlüsselwort** `private` voraus.

Deklaration von und Zugriff auf Instanzvariablen

```
class Laugher3{  
    public Laugher3 ( String s) {  
        ...  
    }  
    public void laugh () {  
        ...  
    }  
    ...  
    private String defaultSyllable;  
}
```

Zu beachten :
defaultSyllable
kann in jedem
Rumpf einer Methode
benutzt werden

Instanzvariablen-Deklaration

Verwendung der Instanzvariable

- In unserem Beispiel ist die Aufgabe des Konstruktors, die mit dem Argument erhaltene Information in der Instanzvariablen `defaultSyllable` zu speichern:

```
public Laugher3(String s) {  
    defaultSyllable = s;  
}
```

- Anschließend kann die `laugh()`-Methode auf `defaultSyllable` zugreifen:

```
public void laugh() {  
    String laughSound;  
    laughSound =  
        defaultSyllable.concat(defaultSyllable);  
    System.out.println(laughSound);  
}
```

Die komplette Laugher3-Klasse

```
class Laugher3 {
    public Laugher3(String s) {
        defaultSyllable = s;
    }
    public void laugh() {
        String laughSound;
        laughSound = defaultSyllable.concat(defaultSyllable);
        System.out.println(laughSound);
    }
    public void laugh(String syllable) {
        String laughSound;
        laughSound = syllable.concat(syllable);
        System.out.println(laughSound);
    }
    private String defaultSyllable;
}
```

Verwendung einer Klassendefinition

1. Wir kompilieren `Laugher3.java`.
2. Wir schreiben ein Programm, das die `Laugher3`-Klasse benutzt:

```
class LaughALittle {
    public static void main(String[] a) {
        System.out.println("Live and laugh!!!");
        Laugher3 x,y;
        x = new Laugher3("yuk");
        y = new Laugher3("harr");
        x.laugh();
        x.laugh("hee");
        y.laugh();
    }
}
```

Der Klassentwurfprozess

Im vorangegangenen Beispiel haben wir mit einer einfachen Klasse begonnen und diese **schrittweise verfeinert**.

Für große Programmsysteme ist ein solcher Ansatz **nicht praktikabel**.

Stattdessen benötigt man ein **systematischeres Vorgehen**:

1. Festlegen des **Verhaltens** der Klasse.
2. Definition des **Interfaces** bzw. der **Schnittstellen** der Klasse, d.h. der Art der Verwendung. Dabei werden die **Prototypen** der Methoden festgelegt.
3. Entwicklung eines kleinen **Beispielprogramms**, das die Verwendung der Klasse demonstriert und gleichzeitig zum Testen verwendet werden kann.
4. Formulierung des **Skelettes** der Klasse, d.h. die **Standard-Klassendefinition** zusammen mit den **Prototypen**.

Festlegen des Verhaltens einer Klasse am Beispiel `InteractiveIO`

Für eine Klasse `InteractiveIO` wünschen wir das folgende Verhalten:

- **Ausgeben einer Meldung auf dem Monitor** (mit der Zusicherung, dass sie unmittelbar angezeigt wird).
- Von dem Benutzer einen **`String` vom Keyboard einlesen**.

Außerdem sollte der Programmierer die Möglichkeit haben, `InteractiveIO`-Objekte zu erzeugen, ohne `System.in` oder `System.out` verwenden zu müssen.

Interfaces und Prototypen (1)

Die **Schnittstelle** einer Klasse beschreibt die Art, in der die Objekte dieser Klasse verwendet werden können.

Für unsere `InteractiveIO`-Klasse wären dies:

- Deklaration einer `InteractiveIO`-Referenzvariablen:

```
InteractiveIO interIO;
```

- Erzeugen eines `InteractiveIO`-Objektes:

```
interIO = new InteractiveIO();
```

In diesem Beispiel benötigt der Konstruktor kein Argument.

Interfaces und Prototypen (2)

- Senden einer Nachricht zur Ausgabe eines `String`-Objektes an ein `InteractiveIO`-Objekt.

```
interIO.write("Please answer each question");
```

Resultierender Prototyp:

```
public void write(String s)
```

- Ausgeben eines Prompts auf dem Monitor und Einlesen eines `String`-Objektes von der Tastatur. Dabei soll eine Referenz auf den `String` zurückgegeben werden:

```
String s;  
s = interIO.promptAndRead("What is your first name? ");
```

Resultierender Prototyp:

```
public String promptAndRead(String s)
```


Ein Beispielprogramm, das InteractiveIO verwendet

Aufgaben des Beispielprogramms:

1. Demonstrieren, wie die neue Klasse verwendet wird.
2. Prüfen, ob die Prototypen so sinnvoll sind.

```
class TryInteractiveIO {  
    public static void main(String[] arg) throws Exception {  
        InteractiveIO interIO;  
        String line;  
  
        interIO = new InteractiveIO();  
        line = interIO.promptAndRead("Please type in a word: ");  
        interIO.write(line);  
    }  
}
```

Das Skelett von InteractiveIO

Gegenüber der kompletten Klassendefinition fehlt dem **Skelett** der Code, der die Methoden realisiert:

```
// Easy to use class for communicating with the user
class InteractiveIO {
    public InteractiveIO() {
    }

    //Write s to the monitor
    public void write(String s) {
    }

    //Write s to the monitor, read a string from the
    keyboard,
    //and return a reference to it.
    public String promptAndRead(String s) throws Exception {
    }
}
```

Implementierung von InteractiveIO (1)

Die **Implementierung** einer Klasse besteht aus dem **Rumpf der Methoden** sowie den **Instanzvariablen**.

Dabei spielt die **Reihenfolge**, in der Methoden (weiter-) entwickelt werden, **keine Rolle**.

Der Konstruktor tut nichts:

```
public InteractiveIO() {  
}
```

Als nächstes definieren wir die Methode `write`:

```
public void write(String s) {  
    System.out.println(s);  
    System.out.flush();  
}
```

Implementierung von InteractiveIO (2)

Schließlich implementieren wir `promptAndRead`.

Um in einer Methode den **Return-Wert** zurückzugeben, verwenden wir das **Return-Statement**:

```
return Wert;
```

Dies ergibt:

```
public String promptAndRead(String s) throws Exception {
    System.out.println(s);
    System.out.flush();

    BufferedReader br;
    br = new BufferedReader(new InputStreamReader(System.in));

    String line;
    line = br.readLine();
    return line;
}
```

Die komplette Klasse InteractiveIO

```
import java.io.*;

class InteractiveIO {
    public InteractiveIO() {
    }
    public void write(String s) {
        System.out.println(s);
        System.out.flush();
    }
    public String promptAndRead(String s) throws Exception {
        System.out.println(s);
        System.out.flush();
        BufferedReader br;
        br = new BufferedReader(new InputStreamReader(System.in));
        String line;
        line = br.readLine();
        return line;
    }
}
```

Siehe: [examples/InteractiveIO_1.java](#)

Verbessern der Implementierung von InteractiveIO

- Häufig ist eine **erste Implementierung** einer Klasse noch nicht **optimal**.
- Nachteilhaft an unserer Implementierung ist, dass bei jedem Einlesen einer Zeile ein `BufferedReader` und ein `InputStreamReader` erzeugt wird.
- Es wäre viel günstiger, diese Objekte einmal zu erzeugen und anschließend wiederzuverwenden.
- Die entsprechenden Variablen können wir als **Instanzvariablen deklarieren** und die Erzeugung der Objekte können wir **in den Konstruktor verschieben**.

Prinzip der Verbesserung von InteractiveIO

```
class InteractiveIO{
    public InteractiveIO() {
        br = new BufferedReader(
            new InputStreamReader(System.in));
        ...
    }
    public String promptAndRead(String s) throws Exception {
        System.out.println(s);
        System.out.flush();
        ...
    }
    private BufferedReader br;
}
```

Weitere Vereinfachungen

1. Wir können die von `readLine` erzeugte Referenz auch direkt zurückgeben:

```
return br.readLine();
```

2. Beide Methoden `write` und `promptAndRead` geben etwas auf dem Monitor aus und verwenden `println` und `flush`. Dies kann in einer Methode zusammengefasst werden:

```
private void writeAndFlush(String s) {  
    System.out.println(s);  
    System.out.flush();  
}
```


Das Schlüsselwort `this`

Mit der Methode `writeAndFlush` können wir sowohl in `write` als auch in `promptAndRead` die entsprechende **Code-Fragmente** ersetzen.

Problem: Methoden werden aufgerufen, indem Nachrichten an Objekte gesendet werden. Aber an welches Objekt können wir aus der Methode `write` eine Nachricht `writeAndFlush` senden?

Antwort: Es ist **dasselbe Objekt**.

Java stellt das **Schlüsselwort `this`** zur Verfügung, damit eine **Methode das Objekt, zu dem sie gehört, referenzieren** kann:

```
this.writeAndFlush(s);
```

Die komplette, verbesserte Klasse InteractiveIO

```
import java.io.*;

class InteractiveIO {
    public InteractiveIO() {
        br = new BufferedReader(new InputStreamReader(System.in));
    }
    public void write(String s) {
        this.writeAndFlush(s);
    }
    public String promptAndRead(String s) throws Exception {
        this.writeAndFlush(s);
        return br.readLine();
    }
    private void writeAndFlush(String s) {
        System.out.println(s);
        System.out.flush();
    }
    private BufferedReader br;
}
```

Siehe: [examples/InteractiveIO_2.java](#)

Deklarationen und das `return`-Statement

Reihenfolge der Vereinbarungen: Die Reihenfolge von Variablen- und Methodendeklarationen in einer Klasse ist beliebig. Es ist jedoch eine gängige Konvention, erst die Methoden zu deklarieren und dann die Variablen.

Das `return`-Statement: Der Sender einer Nachricht kann nicht fortfahren, bis die entsprechende Methode beendet ist. In Java geschieht die Rückkehr zum Sender durch ein `return`-Statement oder, sofern die Methode den Typ `void` hat, am Ende der Methode. Allerdings können auch `void`-Methoden mit `return` beendet werden:

```
private void writeAndFlush(String s) {  
    System.out.println(s);  
    System.out.flush();  
    return;  
}
```

Zugriffskontrolle

- Eine **Klassendefinition** besteht aus **Methoden** und **Instanzvariablen**.
- Der Programmierer kann einen unterschiedlichen **Zugriff** auf **Methoden** oder **Variablen** gestatten, indem er die Schlüsselwörter `public` oder `private` verwendet.
- Als `public` deklarierte Methoden können **von außen aufgerufen** werden. Als `private` vereinbarte Methoden sind jedoch **nur innerhalb der Klasse bekannt**.
- Gleiches gilt für **Instanzvariablen**.

Variablen und ihre Lebensdauer

Wir haben **drei verschiedene Arten von Variablen** kennen gelernt:

1. als **Parameter im Kopf der Definition einer Methode**,
2. als **lokale Variable** definiert **innerhalb des Rumpfes einer Methode** und
3. als **Instanzvariablen in der Klassendefinition**.

Variablen als Parameter

Variablen, die **Parameter einer Methode** sind, werden **beim Aufruf der Methode automatisch erzeugt** und sind **innerhalb der Methode bekannt**. Ist die Methode beendet, kann auf sie nicht mehr zugegriffen werden. Ihre Lebenszeit entspricht der der Methode.

Parameter erhalten ihren **initialen Wert vom Sender** der Nachricht und die **Argumente des Aufrufs müssen exakt mit den Argumenten der Methode übereinstimmen**.

Für `void f(String s1, PrintStream p)` ist der Aufruf

```
f("hello", System.out)
```

zulässig. Die folgenden Aufrufe hingegen sind alle unzulässig:

```
f("hello")
```

```
f("hello", "goodbye")
```

```
f("hello", System.out, "bye")
```

Lokale Variablen

- **Lokale Variablen** die **in Methoden definiert** werden.
- Sie haben die gleiche Lebenszeit wie Parameter.
- Sie werden beim Aufruf einer Methode erzeugt und beim Verlassen der Methode gelöscht.
- Lokale Variablen müssen innerhalb der Methode initialisiert werden.
- Parameter und Variablen sind außerhalb der Methode, in der sie definiert werden, nicht sichtbar, d.h. auf sie kann von anderen Methoden nicht zugegriffen werden.
- Wird in verschiedenen Methoden derselbe Bezeichner für lokale Variablen verwendet, so handelt es sich um jeweils verschiedene lokale Variablen.

Beispiel

```
String getGenre() {  
    String s = "classic rock/".concat(getFormat());  
    return s;  
}
```

```
String getFormat() {  
    String s = "no commercials";  
    return s;  
}
```

Die Wertzuweisung an `s` in `getFormat` hat keinerlei Effekt auf die lokale Referenzvariable `s` in `getGenre`.

Rückgabewert von `getGenre` ist somit eine Referenz auf

```
"classic rock/no commercials"
```


Instanzvariablen

- **Instanzvariablen** haben dieselbe Gültigkeitsdauer wie das Objekt, zu dem sie gehören.
- Auf Instanzvariablen kann **von jeder Methode** eines Objektes aus **zugegriffen werden**.
- Der **Zugriff von außen** wird, ebenso wie bei den Methoden, durch die Schlüsselworte `public` und `private` geregelt.

Lebensdauer von Objekten

- **Objekte** können in den Methoden einer Klasse durch Verwendung von `new` oder durch den Aufruf anderer Methoden **neu erzeugt** werden.

- **Java löscht nicht referenzierte Objekte automatisch.**

```
public void m2() {  
    String s;  
    s = new String("Hello world!");  
    s = new String("Welcome to Java!");  
    ...  
}
```

- Nach der zweiten Wertzuweisung gibt es **keine Referenz** auf "Hello world!" mehr.
- **Konsequenz:** Objekte bleiben so lange erhalten, wie es eine Referenz auf sie gibt.

Das Schlüsselwort `this`

Mit dem Schlüsselwort `this` kann man innerhalb von Methoden einer Klasse das **Objekt selbst referenzieren**. Damit kann man

1. dem **Objekt selbst eine Nachricht schicken** oder
2. bei **Mehrdeutigkeiten** auf das Objekt selbst referenzieren.

```
class ... {  
    ...  
    public void m1 () {  
        String s;  
        ...  
    }  
    ...  
    private String s;  
}
```

Innerhalb der Methode `m1` ist `s` eine **lokale Variable**. Hingegen ist `this.s` die **Instanzvariable**.

Der Konstruktor

- Der **Konstruktor ist immer die erste Methode, die aufgerufen wird.**
- Die Aufgabe eines Konstruktors ist daher, dafür zu sorgen, dass das entsprechende Objekt „sein Leben“ mit den **richtigen Werten beginnt.**
- Insbesondere soll der **Konstruktor** die **notwendigen Initialisierungen der Instanzvariablen vornehmen.**

Zusammenfassung (1)

- Eine **Klassendefinition** setzt sich zusammen aus der Formulierung der **Methoden** und der Deklaration der **Instanzenvariablen**.
- Methoden und Instanzvariablen können als `public` oder `private` deklariert werden, um den **Zugriff** von außen festzulegen.
- Es gibt **drei Arten von Variablen**: **Instanzenvariablen**, **lokale Variablen** und **Parameter**.
- **Lokale Variablen** sind Variablen, die **in Methoden deklariert** werden.

Zusammenfassung (2)

- **Parameter** werden **im Prototyp** einer Methode **definiert** und **beim Aufruf** durch die Wertübergabe **initialisiert**.
- **Instanzvariablen** werden **außerhalb der Methoden** aber **innerhalb der Klasse** definiert.
- **Instanzvariablen speichern Informationen, die über verschiedene Methodenaufrufe hinweg benötigt werden.**