

# Einführung in die Ideen von OO mit einem Beispiel



**Irene Rothe**

[irene.rothe@h-brs.de](mailto:irene.rothe@h-brs.de)

# Vorwissen?

## Lernerfolgsevaluation:

- Ich kann 2 Unterschiede zwischen der OO-Programmierung und der prozeduralen Programmierung erklären.
- Ich kenne die 4 OO-Basics.
- Ich kann die 4 OO-Basics erklären.
- Ich kenne mindestens 4 OO-Prinzipien.
- Ich kann mindestens 3 OO-Prinzipien erklären.
- Ich kann mindestens 2 UML-Diagrammtypen zur Lösung/Beschreibung eines Problems benutzen.
- Ich kann einen groben Entwurf für die Implementierung eines gegebenen Problems objektorientiert erstellen.
- Ich weiß, wofür Entwurfsmuster da sind.
- Ich kann mindestens zwei Entwurfsmuster erklären.




# Planung

- Einstiegsbeispiele: Swimmingpool, Schiffe versenken
- Klasse: Datei mit Eigenschaften (Attributen) und Fähigkeiten (Methoden) möglicher Objekte
- **OO-Basics**: Abstraktion, Kapselung, Vererbung, Polymorphie
- IDEs: Eclipse, javac-Editor, IntelliJ
- GitHub
- Abstrakte Klassen und Interfaces
- **OO-Prinzipien**:
  - Kapseln, was sich ändert
  - Programmieren auf Schnittstelle
  - Schwache Koppelung und starke Kohäsion
  - Subklassen sollten ihre Superklasse vertreten können
- UML
- Entwurfsmuster
- OOA
- Heuristiken
- Parallelprogrammierung mit Java

# Softwarearchitektur



# Design der Folien

-  hinterlegte Informationen sind sehr wichtig und klausurrelevant.
- Alles hinter „**Achtung**“ unbedingt beachten!
-  verwende ich, wenn Überraschende Probleme auftreten können.
- „Tipp“ benutze ich, um Ihnen einen Weg zu zeigen, wie ich damit umgehen würde.
- „Bemerkung“ in Folien beziehen sich meist auf Sonderfälle, die nicht unbedingt klausurrelevant sind, aber für Sie beim Programmieren eine Bedeutung haben könnten.
-  hinter diesem Symbol ist ein Link fürs Anhören bzw. Gucken weiterer Infos

# Motivation für die Informatik

- Top 10 der Programmiersprachen (IEEE)
- Linus Torvalds: Ich weiß nicht, wie ich es erklären soll, was mich am Programmieren so fasziniert, aber ich werde es versuchen. Für jemanden, der programmiert, ist es das Interessanteste auf der Welt. Es ist ein Spiel, bei dem du deine eigenen Regeln aufstellen kannst, und bei dem am Ende das herauskommt, was du daraus machst. Der Reiz besteht dann, dass der Computer das tut, was du ihm sagst. Unbeirrbar. Für immer. Ohne ein Wort der Klage. Du kannst den Computer dazu bringen, das er tut, was du willst, aber du musst herausfinden, wie. Programmieren ist eine Übung der Kreativität.
- Alan Turing (Gründer der Informatik): if thoughts (that is, information) can be broken up into simple constructs and algorithmic steps, then machines can add, subtract or rearrange them as our brains do.

Informatik =  
Lösen von Problemen mit dem Rechner

# Evaluation

→ Lernzielevaluation





# Mein Teaching Style

ICH BIN ANSPRECHBAR für so gut wie alles!!

Ich wäre gern: Ihr Coach, Ihr Motivator, Ihr Unterstützer

Meine Methode: learning by doing, Lernen aus Fehlern



# IDE für JavaSoftwarearchitektur

Extra Vorlesung: JavaIDES



# GitHub („Blödmann“ auf Englisch und kein Linux-Befehl bis dahin)

1. Registration bei github.com mit Free Plan
2. Getting Started lesen aus Interesse
3. Benutzernamen schicken an [irene.rothe@h-brs.de](mailto:irene.rothe@h-brs.de)
4. Geben Sie mir (irenerothe) Leserechte (Read) auf ihr Repository
5. Installieren Sie eine IDE zur Java-Programmierung
6. Binden Sie Ihr Repository in die IDE ein
7. Zugriff auf GitHub über github.com oder IDE (Netbeans, Eclipse)
8. Netbeans: Team→Git→Clone...



# GitHub – Erste Schritte

1. Erstellen Sie eine erste Java-Datei im Hauptverzeichnis des Repositories.

2. Kopieren Sie folgende Zeilen rein:

```
class HelloWorld{  
    public static void main(String[] args){  
        System.out.println("Hallo, alles perfekt!");  
    }  
}
```

3. Machen Sie einen Commit und einen Push in den Master-Branch.

Arbeit mit GitHub: über git cmd für Windows (<https://git-for-windows.github.io/>)  
oder App runterladen: „Set up in Desktop“

Super Webseite: <https://rogerdudler.github.io/git-guide/index.de.html>



# Arbeit mit GitHub (GitCMD)

Arbeit mit der GitHub-App oder git CMD für Windows:

1. Öffnen der Git CMD
2. Bekanntmachen meiner email-Adresse, mit der ich bei GitHub bekannt bin (EINMALIG!):  
`git config --global user.email <email adresse>`
3. **clone** (Kopieren lokal) repository in einen Ordner Test, den man später dann wieder löschen kann, z.B.  
`git clone https://github.com/irenerothe/TestRep GitHub/Test`
4. Neue Datei in den Ordner hauen oder was ändern
5. In den entsprechenden Ordner wechseln, z.B. `cd GitHub/Test`
6. `git add` (Datei wird in die gitattributes eingetragen), z.B. `git add *`
7. **commit** (Zeug ist immer noch nicht auf Server=bei github), z.B.  
`git commit -m "Ausprobieren"`
8. push, puh, nun ist es da: `git push origin master`



# Arbeit mit Git im Team

- Anlegen der ersten Dateien von einem collaborator, der sie dann pushed
- Woraufhin alle anderen einen pull machen können und anfangen können mit der Arbeit
- Oft commit ausführen!!

Achtung: Es kann passieren, dass 2 an der gleichen Datei arbeiten, dann wird neuer Branch eventuell aufgemacht, dann kann es Merge-Konflikte geben, die man aber auch beheben kann....

# Readme in GitHub bzgl. License

Ich empfehle es, eine Datei anzulegen, z.B. mit Namen License.txt mit folgendem Text:

The MIT License

Copyright (c) 2020 <Eure Namen>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Mehr: [The Legal Side of Open Source | Open Source Guides](#)



# GitLab: nicht automatisch öffentlich

- Server mit Lizenz an HS im FB EMT vorhanden
- Link:



# UML zeichnen mit ...

- draw.io (benötigt z.B. google-Konto)
- UMLet (<https://www.umlet.com/>)

# Empfehlenswerte Kurse zum Lernen vom HassnoPlattnerInstitut: open.hpi.de

Zum Beispiel der Javakurs dort:  
<https://open.hpi.de/courses/javaeinstieg2020>



# Unterlagen im DMS (DigitalMakerspace von Andre Kless)

- Vorlesung OO: [https://ccmjs.github.io/digital-makerspace/app.html?app=app\\_collection,1646838136468X5798972600492583](https://ccmjs.github.io/digital-makerspace/app.html?app=app_collection,1646838136468X5798972600492583)
- DMS selbst zur Nutzung, um aus vorgefertigten Webbausteinen (sogenannte Apps) beliebige Webanwendungen zusammen zu bauen oder selbst zu programmieren: <https://ccmjs.github.io/digital-makerspace>

# Prozedurale Programmierung

- Umsetzung eines Algorithmus
- Entwurf von Funktionen
- Aufruf der Funktionen im Hauptprogramm in einer bestimmten Reihenfolge

## Nachteile:

Das ganze Programm gehört zusammen, deshalb sind Einzelteile nur anstrengend woanders verwendbar (in der Regel entstehen dabei eine Menge Fehler).

Will man am Programm etwas nachbessern oder hinzufügen, muss man schon lauffähigen (getesteten) Code anfassen und macht ihn in der Regel kaputt => „verdammt, eben lief doch noch alles!!!“

# Objektorientierte Programmierung

- Es werden viele kleine Komponenten programmiert (Klassen/Objekte), die man extra testen kann.
- Aus den Objekten können dann viele verschiedene Algorithmen zusammengebaut werden, je nach Lust und Laune.

# Ziel

Lehre der Paradigmen von OO.

**KEINE** Lehre einer konkreten Programmiersprache!

# Ein Beispiel für alles: Schiffe versenken

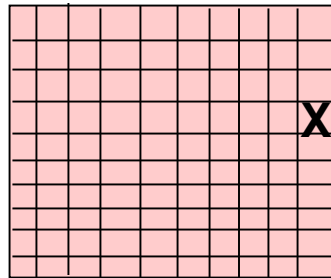
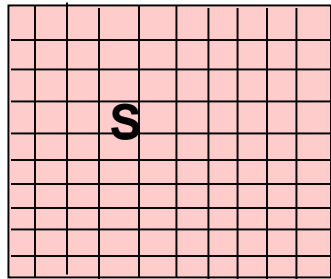
Das Spiel Schiffe versenken wird im Laufe der Vorlesung programmiert und immer mehr verbessert durch Anwendung neu gelernter Konzepte. Dies entspricht aber nicht dem Vorgehen von OO-Designern, sondern wird aus Lehrgründen so vollzogen.

# Aufgabe: Programmierung vom Spiel "Schiffe versenken"

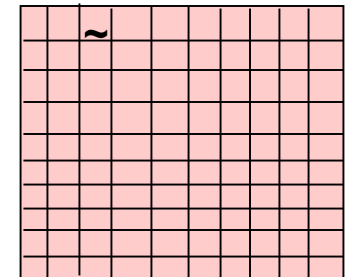
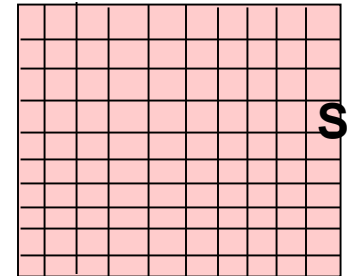
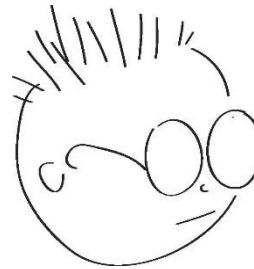
**Zwei Spieler:** A und B

**Zwei Karten pro Spieler:** Seekarte und Feindeskarte

**Spieler A**



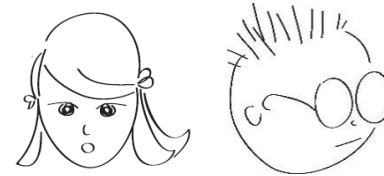
**Spieler B**



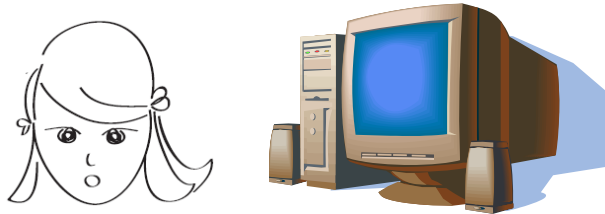


# Programmieraufgaben

**1. Zwei Spieler gegeneinander**

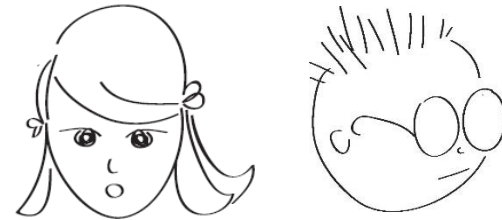


**2. Ein Spieler gegen den Computer**



# Spiel mit zwei Spielern

Wie könnte man das machen?



- Aufteilung in Funktionen?
- Welche?
- Funktionseingabewerte: Name des Spielers, Name der jeweiligen Karte
- ...

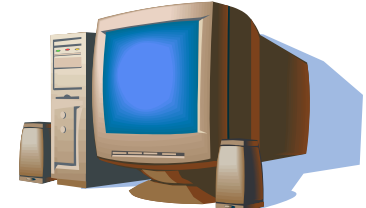
➔ **Chaos!!**

- Welcher Spieler darf nun auf welche Karte schreiben?
- Welcher Spieler ist überhaupt dran?

➔ in OO existiert dafür eine Zugriffskontrolle, genannt  
**KAPSELUNG**



# Spiel Computer gegen Mensch



Implementation neuer Funktionen für den „Computerspieler“

→ mit **Copy & Paste** brauchbare Codezeilen aus den Mensch-gegen-Mensch Funktionen holen

→ **Vorprogrammiertes Chaos!**

Dann gibt es eventuell Spielregeländerungen. Die muss man dann bei allen „Mensch“-Funktionen einarbeiten und zusätzlich auch bei den Computer-Funktionen.... **never ending problems!!**

→ noch mehr Chaos!

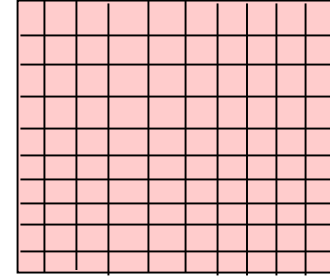
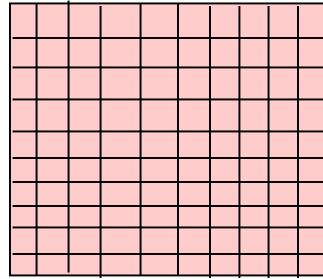
→ in OO existiert dafür eine Hierarchiebildung durch Vererbung

# Alles besser mit OO:

## Beschreibung eines Schiffeversenk-Spielers

### Besitztümer:

- Seekarte
- Feindeskarte



### Fähigkeiten:

- Schiffe setzen auf die Seekarte
- Schießen auf die Schiffe des Gegners
- Angabe über den Ausgang eines gegnerischen Schusses (getroffen oder Wasser)
- Eintrag des Schussresultates in die eigene Feindeskarte
- Angucken der eigenen Feindeskarte

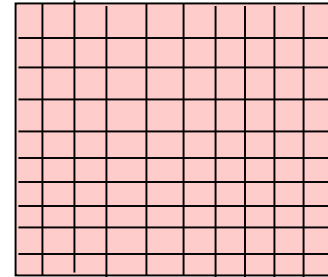
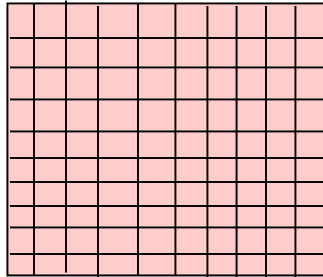
# Objekte in OO

- ➔ haben bestimmte Eigenschaften (**Attributwerte**)
- ➔ reagieren mit definiertem Verhalten (Menge von **Methoden**)
- ➔ werden beschrieben in einer **Klasse**: Datei, die mit **class** anfängt

# Definition einer Klasse Spieler

Besitztümer:

- **seekarte**
- **feindeskarte**



Fähigkeiten:

- **setzenSchiffe:** Schiffe setzen auf die Seekarte
- **schiessen:** Schießen auf die Schiffe des Gegners
- **auswerten:** Angabe über den Ausgang eines gegnerischen Schusses (getroffen oder Wasser)
- **eintragen:** Eintrag des Schussresultates in die Feindeskarte
- **ausdruckenKarte:** Ausgabe der Feindeskarte auf dem Bildschirm

# Code (Java) von Spieler.java

```
class Spieler{
    //declaration of the maps
    private char [][] seekarte = new char [10][10];
    private char [][] feindeskarte = new char [10][10];
    String name;
    int anzahlsschiffe;
    //Default-constructor
    Spieler(){}
    //constructor
    Spieler (String name, int anzahlsschiffe){
        //local variables and others
        ...
        this.name=name;
        this.anzahlsschiffe = anzahlsschiffe;
        ...
    }
    void setzenSchiffe(){...}
    ... schiessen(){...}
    int auswerten(...){...}
    void eintragen(int result, ...){...}
    void ausdruckenKarten(){...}
}
```



# Bemerkungen zur Klasse: `Spieler.java`

- Karten gehören nur dem Spieler selbst → **private**
- Methoden: das, was der Spieler tun kann
- Keine Kartenübergabe in den eigenen Methoden notwendig, da die Karten der Klasse allgemein bekannt sind (eine Art global-lokal)
- Konstruktor: Dinge, die nur einmal gemacht werden sollen, wie z.B. Objekt wird in gültigen Zustand gesetzt (eventuell die Karten mit Punkten füllen)



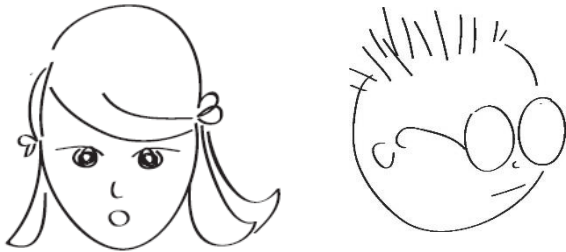
# Spieler (A und B) rufen Methoden auf:

1. Spieler A setzt seine Schiffe: `spielerA.setzenSchiffe()` ;
2. Spieler B setzt seine Schiffe: `spielerB.setzenSchiffe()` ;
3. Spieler A schießt: `spielerA.schiessen()` ;
4. Spieler B meldet das Schussresultat:  
`spielerB.auswerten(...)` ;
5. Spieler A vermerkt dies Resultat: `spielerA.eintragen(...)` ;
6. Ausdruck der Feindeskarte von Spieler A:  
`spielerA.ausdruckenKarten()` ;
7. Weiter gehts mit Spieler B.

# Erzeugung von Spielern (Objekterzeugung)

Anlegen von zwei Spielern im Hauptprogramm:

```
Spieler spielerA = new Spieler("SpielerA",anzahlderschiffe);  
Spieler spielerB = new Spieler("SpielerB",anzahlderschiffe);
```



# Erzeugen von Objekten - genauer

1. **Deklaration** einer Variablen, welche eine Referenz aufnehmen kann
2. **Erzeugung** des Objektes mit dem **new**-Operator und Konstruktor i.d.R. auf dem Heap (Konstruktor ist eine spezielle Methode zur Initialisierung der Speicherplätze eines Objektes. Ein Konstruktor hat den gleichen Namen wie die Klasse und kann eine Parameterliste zur Initialisierung der Objektattribute haben)
3. **Zuweisung** der Referenz auf das Objekt an die zuvor deklarierte Variable

```
Spieler spielerB = new Spieler("SpielerB", anzahlderschiffe) ;
```

Klassenname / Typ      Referenzvariable      Konstruktor      Parameterliste

Danach ist ein Zugriff auf die Variable wie folgt möglich:

**<Referenzvariable>.<Attributvariablen/Methodenname>**

# Bemerkung: Arrays sind auch Objekte

Hier ist nun ganz klar, dass der Arrayname eine Referenz (Adresse) ist.

```
int [] feld = new int[10];
```

Referenzvariable

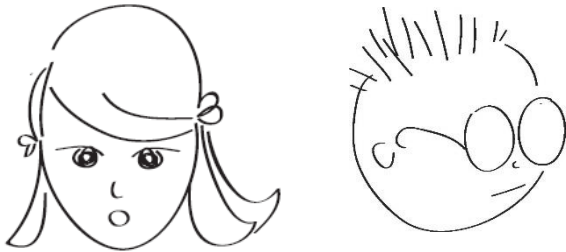


erst ab hier wird Speicher belegt.

# Zeigen und Vorführung des OO-Programms mit zwei Spielern

**Spieler.java**

**SchiffeVersenken.java**



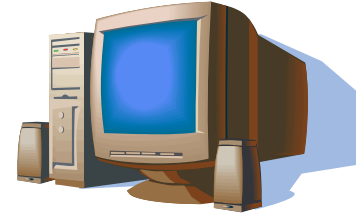
# Ideen für noch mehr Klassen?

- **Karte.java** (da doppelter Code)  
→ wenn z.B. ein Kunde das Aussehen der Karte geändert haben will, ist das sehr einfach durch einfaches Austauschen der Kartenklasse
- **Schiff.java**

Jeglicher doppelter Code ist Indiz für schlechten Klassenentwurf!



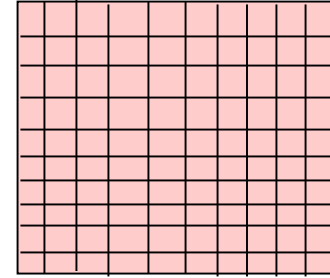
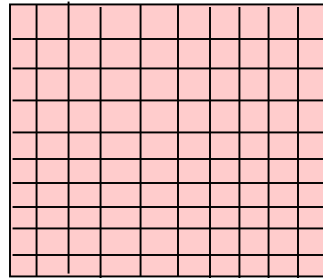
# Erweiterungsspiel: Computer gegen Mensch



# Wiederholung: Schiffeversenk-Spieler

## Besitztümer:

- Seekarte
- Feindeskarte



## Fähigkeiten:

- Schiffe setzen auf die Seekarte
- Schießen auf die Schiffe des Gegners
- Angabe über den Ausgang eines gegnerischen Schusses (getroffen oder Wasser)
- Eintrag des Schussresultates in die eigene Feindeskarte
- Angucken der eigenen Feindeskarte



# Bemerkung

Schlüsselwort **private** muss vor den Karten wieder weg, sonst können sie nicht an den **Computer** vererbt werden.

Oder sie werden extra im Computer neu definiert und dort auch **private** gesetzt

Unterbemerkung: oder über super oder ... ist eh ein schlechter Klassenentwurf



# Mensch gegen Computer

Zwei Fragen:

Was macht der Computer *nicht* wie der Mensch?

➔ Ein Computer schießt und setzt seine Schiffe anders, z.B. zufällig

Was macht der Computer *genauso* wie der Mensch?

➔ alles andere

# Beschreibung eines Schiffeversenk-Computers

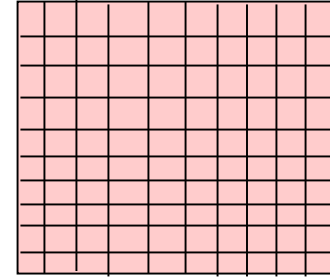
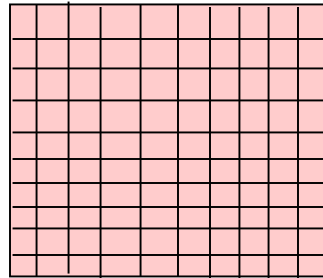
## Besitztümer:



- Seekarte



- Feindeskarte



## Fähigkeiten:

- Schiffe setzen auf die Seekarte
- Schießen auf die Schiffe des Gegners



- Angabe über den Ausgang eines gegnerischen Schusses (getroffen oder Wasser)



- Eintrag des Schussresultates in die eigene Feindeskarte

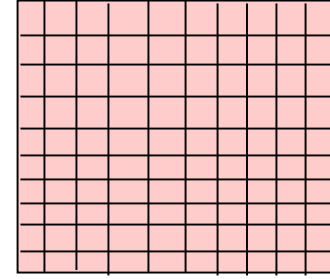
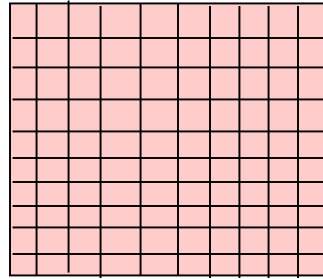


- Angucken der eigenen Feindeskarte

# Beschreibung eines Schiffeversenk-Computers

## Besitztümer:

- Seekarte
- Feindeskarte



## Fähigkeiten:

- ~~• Schiffe setzen auf die Seekarte~~ → z.B. zufällig
- ~~• Schießen auf die Schiffe des Gegners~~ → z.B. zufällig
- Angabe über den Ausgang eines gegnerischen Schusses (getroffen oder Wasser)
- Eintrag des Schussresultates in die eigene Feindeskarte
- Angucken der eigenen Feindeskarte

# Definition einer Klasse Computer

## Besitztümer:

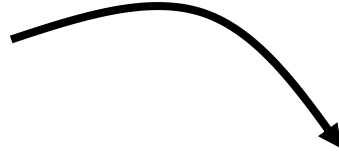
- alles wie in **Spieler.java**

## Fähigkeiten:

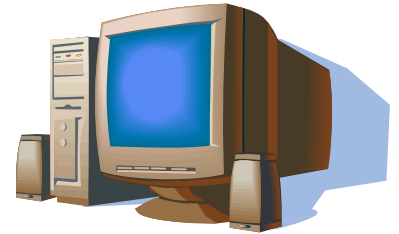
- alles wie in **Spieler.java**
- **setzenSchiffe** neu machen (z.B. zufällig)
- **schieszen** neu machen (z.B. zufällig)

# Definition der Klasse Computer

Vererbung:



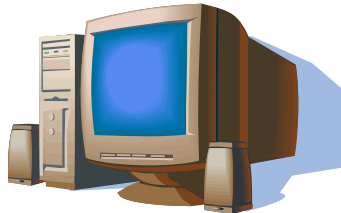
```
class Computer extends Spieler {  
    //Erzeugung zufaelliger Zahlen  
    ...  
    Computer(...) {  
        super(...);  
    }  
    void setzenSchiffe() {...}  
    ...schiessen() {...}  
}
```



# Ersetzung eines Spielers durch einen Computer

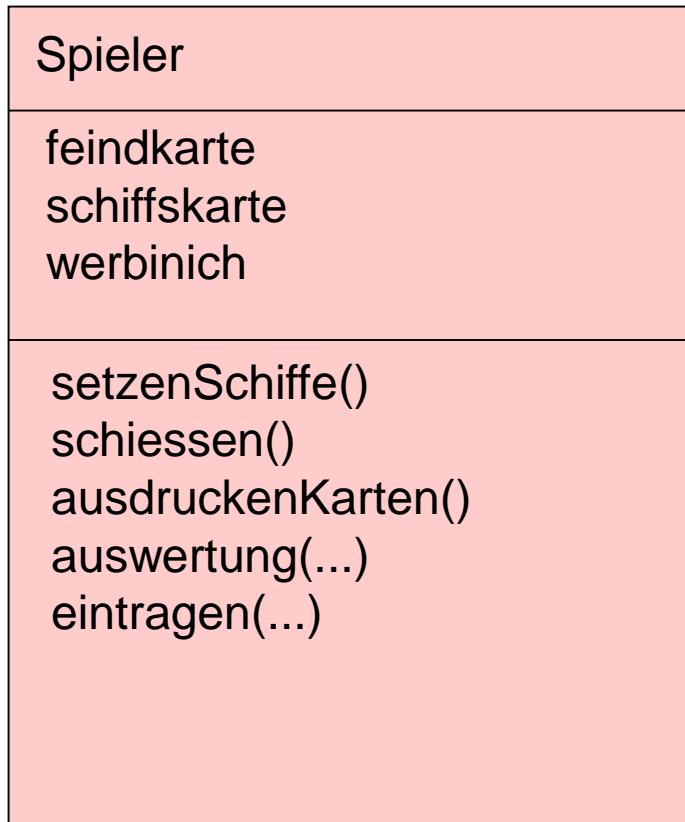
```
Spieler spielerA=new Spieler("SpielerA",anzahlderschiffe);
```

```
Computer spielerB=new Computer("Computer", anzahlerschiffe);
```

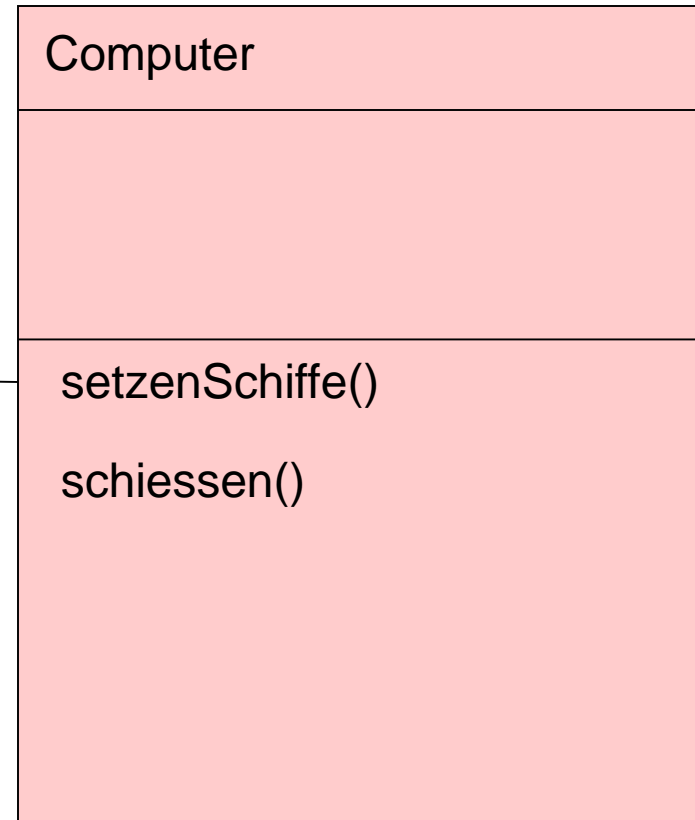


# Schiffe versenken

*Elternklasse*



*Kindklasse*



UML (Unified Modeling Language)



# Das objektorientierte Paradigma

Der Vertrag eines *Algorithmus* mit seinem Nutzer ist wie ein *Verkaufsvertrag*, der für jede Eingabe eine Ausgabe garantiert.

Der Vertrag eines *Objektes* mit seinem Nutzer ist wie ein *Ehevertrag*, der das Verhalten über die Lebenszeit hinweg festlegt.

# Objektorientierung

- Das heißt, zusammengehörige Daten und Funktionen zusammenzufassen (in einer **Klasse**) und weitgehend vor der Welt zu verbergen (Verkapselung/Black Box-Prinzip) -> **Objekte** (**Instanzen** von Klassen)
- Wie man entscheidet, welche Objekte günstig sind, was man nach außen zeigt und was man versteckt, ist die Aufgabe von OO-Programmierern.

# Klassen

sind eine Art Erweiterung von Strukturen (**struct**) aus C, die aber nun auch Funktionen enthalten dürfen

# Klassen

- fassen Daten und Methoden zusammen ➔ *gekapselte* Datenstruktur.
- **Methoden** sind dafür da, dass man diese Daten verändern kann. Sie sind also die *Schnittstelle* der Klasse. Methoden beschreiben das Verhalten der Klasse/Objekt.
- aus Klassen können Objekte erzeugt werden, genannt **Instanzen einer Klasse**.
- Klassen werden oft in eigener Textdatei abgespeichert
- Teile der Klassen können durch Modifizierer wie **public**, **private**, **protected** spezifiziert werden. Dies kann als Schutzmechanismus gesehen werden.

# Objekte

- werden erzeugt bei Ausführung des **new**-Operator
- sind etwas Einmaliges
- werden auch Instanzen einer Klasse genannt
- Zugriff auf Objekte erfolgt durch Referenzen, z.B.

**KlasseA a = new KlasseA(...); //a ist Referenz**

- Objekte kommunizieren durch Versand von Nachrichten miteinander über die in den Klassen definierten **Methoden** (auch „Aufruf von Methoden“ genannt), z.B.  
**a.methodeDerKlasseA(...);**
- werden durch **Klassen** definiert
- Eine Klasse kann viele Instanzen (Objekte) haben

# Statische Methode main

- Hauptprogramm ist auch in einer Klasse eingebettet, es wird aber kein Objekt dieser Klasse erzeugt
- **main** ist Beginn der Programmausführung, z.B.

```
class MainKlasse{  
    public static void main(String[] args) {  
        ...  
    }  
}
```

# OO-Programme

...sind also eine Sammlung von Klassen.

- Es gibt riesige Bibliotheken von fertigen Klassen, die verwendet werden können.
- Während der Programmausführung werden nach Bauplan der Klassen Objekte erzeugt.
- Die Objekte steuern den Programmablauf durch Austausch von Nachrichten.

# OO-Basics

- **Abstraktion** (z.B.: Definition eines Spielers)
- **Kapselung** (z.B.: kann niemand außer der Spieler auf die Seekarte oder Feindeskarte schreiben)
- **Vererbung** (z.B.: Computer ist abgeleitet von Spieler)
- **Polymorphie** (Vielgestaltigkeit)



Stottervideo: <https://youtu.be/aUta4QwF9nU>



# Genauer: Kapselung

- Kompetenz/Fähigkeiten und Verantwortung sind in einer Klasse konzentriert und nicht über das ganze Programm *“verschmiert”*
- Unautorisierter Zugriff von außen ist nicht so einfach möglich
- bietet besonders Vorteile für große Projekte, wo das Hauptproblem ist, den *Überblick* zu behalten
- einfachere Testung, da Klassen einzeln getestet werden können

# Genauer: Vererbung (IST-EIN)

- Copy & Paste nicht nötig für Wiederverwendung von Programmcode
- doppelter Code wird einfach in eine Superklasse verschoben
- falls Änderungen nötig sind, müssen diese nur an einer Stelle implementiert werden (und wie durch Zauberhand reagieren alle abgeleiteten Klassen darauf)
- drückt IST-EIN Beziehung aus

# Genauer: Polymorphie

- Ist Eigenschaft, die bei Vererbung und Schnittstellen (Interfaces) auftritt.
- eine Methode ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist und je nach vorangestellten Objekt anders/neu reagiert.

# Neue Begriffe

- **Vererbung:** Methoden/Variablen aus *Super*klassen können in *Sub*klassen verwendet werden, ohne dass sie dort extra definiert wurden.
- **Überschreiben:** Methoden (Variablen) aus Superklassen können in Subklassen unter gleichem Namen *neu* definiert werden.
- **Polymorphie:** *gleiche* Methodennamen mit *gleichen* Parametern können in *verschiedenen* Klassen verwendet werden.
- **Überladen (auch eine Art Polymorphie):** *gleiche* Methodennamen oder Konstruktorennamen mit *verschiedenen* Parametern können in *einer* Klasse verwendet werden.

# Prozedurale versus OO-Programmierung

	ProzeduralePP	OOD
Design-prinzip	Trennung zwischen Daten und Methoden, welche auf diesen angewendet werden	Daten und darauf angewendete Methoden zusammengefasst zu Objekten

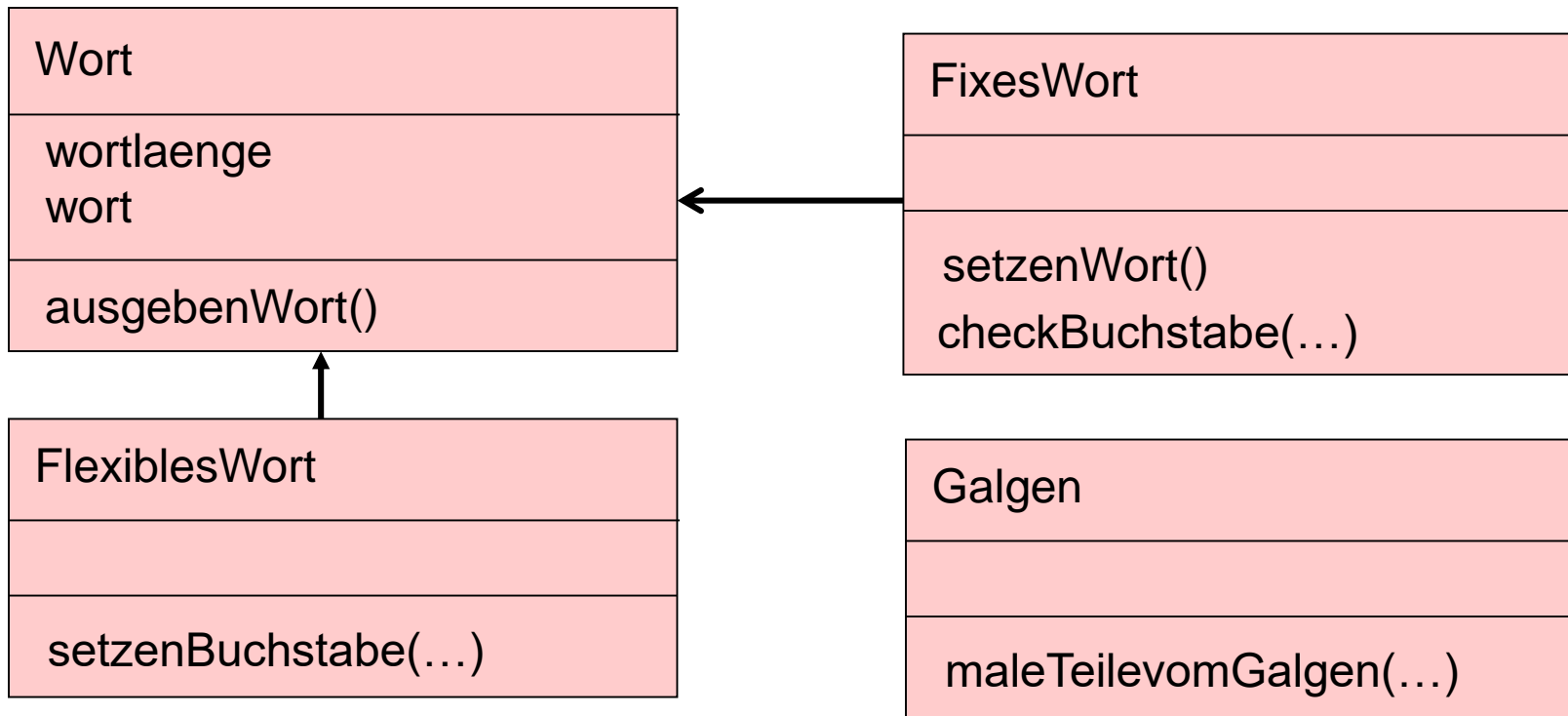
# Methodennamen...

- sollten sich lesen wie ein englischer oder deutscher Satz
- Am besten beginnend mit einem Verb
- sind dadurch gut unterscheidbar von Variablen, weil Methoden was „tun“
- Folgende Methoden sind üblich: get/set/has

Beispiel: **setzenSchiffe()**

# Beispiel: Hangman

(Ordner: Hangman2011)



# Tipp

Wenn Sie eine Klasse entwerfen, sollten Sie nur an das Objekt denken:

- Dinge, die es weiß (Instanzvariablen) und
- Dinge, die es macht (Methoden)



# Praxisbeispiel: Sortieranlage

- Kommunikation zwischen den Objekten über *Methodenaufrufe*
- Aufbau aus unabhängigen *Objekten*: Transportbänder (zum Transportieren), Verteiler (zum Verteilen), Endstellen (zum Zählen oder Fassen)
- beliebige Erweiterungen möglich durch zusätzliche Klassen
- Bedienung über das Hauptprogramm

# Ein paar Worte zu Java:

- erfüllt die Sicherheitsanforderungen für den Entwurf von eingebetteten System, da viele gefährliche Eigenschaften und Fähigkeiten von C und C++ nicht vorhanden sind.
- unterstützt Ausnahmebehandlungen, wodurch die Behandlung von Laufzeitfehlern vereinfacht wird.
- verwaltet den Speicher durch Garbage-Collection, wodurch die Freigabe von Speicherplatz organisiert wird, was gut ist für Systeme, die sehr lange laufen ohne Neustart.
- unterstützt ebenfalls Threads (Nebenläufigkeit).
- ist eine objekt-orientierte Sprache, wodurch Code sehr einfach implementiert werden kann.
- Nachteil: Echtzeitanwendungen (große Anwendungen, keine direkte Kontrolle über Geräte, automatische Speicherverwaltung, keine Aussagen über Thread-Reihenfolge möglich).

# Ein paar Dinge zu Garbage Collection in Java:

Erinnerung: wenn man das nicht macht, ist irgendwann kein Speicher mehr da, auch wenn kaum ein Programm Speicher noch benutzt

- nicht erreichbarer Speicher: es existiert keine Referenz mehr auf die Speicheradressen
- scheinbar legt JVM eine Liste an, wo alle Adressen im Heap eingetragen werden, wo Objekte liegen. Ab und an geht die Garbage Collection durch die Listen und sucht für jede dieser Adressen nach Variablen, die diese Adressen gespeichert haben. Wenn nicht werden diese Objekte gelöscht.
- das klingt extrem aufwändig. Es gibt aber mehrere Bereiche im Speicher, wo Objekte liegen können: die junge Generation, die alte Generation (ab einem bestimmten Schwellenalter kommt man hier rein) und die permanente Generation (nur bis Java 7, dann anders geregelt)
- immer wenn ein Bereich von Speicher voll ist, werden Objekte gelöscht oder verschoben mit dem entsprechenden Nachtrag in den Listen (Variable/Adresse)

# Wie unterscheidet sich Java von C++?

- keine gotos
- keine globalen Variablen
- keine Zeiger
- kein Präprozessor
- keine Mehrfachvererbung
- Garbage collection (deshalb kein Destruktor nötig, also kein delete)
- Übersetzung in Bytecode (also nicht in Maschinensprache, keine exe)
- Funktionsrückgabewerte sind nicht signifikant: Funktionen mit unterschiedlichen Rückgabedatentypen und sonst gleich sind nicht verschieden (könnte auch in C++ so sein?)