

# Aufgabe 1: Erste Objekte

- Erzeuge manuell Objekte der Klasse `Schatz` und `Schatztruhe` und experimentiere mit den vorhandenen Methoden:
  - ▶ Weise den Schätzen jeweils einen unterschiedlichen Wert zu
  - ▶ Lege die Schätze in die Schatztruhe
  - ▶ Berechne den Wert der Schatztruhe
- Ergänze die `Schatztruhen`-Klasse um eine Methode, die die Kapazität der Schatztruhe auf 10 erhöht

## Aufgabe 2: Mehr Sicherheit!

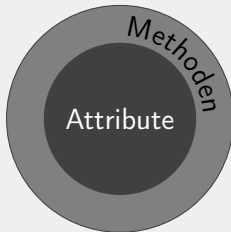
Die Sicherheit der Schatztruhen soll in *Verliese und Objekte* soll erhöht werden.

- a) Erweitere die Klasse Schatztruhe um eine Methode `schliesseAb()` und eine Methode `schliesseAuf()`. Das Hineinlegen in die Truhe soll nur möglich sein, wenn diese nicht aufgeschlossen ist.

Verschließen und Aufschließen soll mit einem *Passwort* funktionieren.

- b) Ändere die beiden Methoden so ab, dass sie einen `int`-Parameter benötigen. Öffne die Truhe nur, wenn das Passwort beim Öffnen mit dem Passwort beim Schließen übereinstimmt.
- \*) Erweitere die Klasse Schatztruhe um eine Methode, die den letzten Schatz wieder aus der Truhe entnimmt.
- \*\*) Jeder Schatz hat ein *Volumen*. Eine Schatztruhe kann maximal 5 Gegenstände aufnehmen und deren Gesamtvolumen darf eine feste Kapazität nicht überschreiten.

- Attribute eines Objekts sollen nur über entsprechende Methoden des Objekts manipuliert werden dürfen



- Explizite Interaktion mit Attributen via getter (auslesen) und setter (setzen)

# Zugriffsmodifikatoren

- Zugriffsmodifikatoren setzen fest, wer/was auf Attribute und Methoden zugreifen kann

**public** Zugriff von überall

**private** Zugriff nur innerhalb der Klasse

- In der Regel sind in Java Attribute `private`

# Zugriffsmodifikatoren: Wieso?

- Wieso nutzt man Zugriffsmodifikatoren?
  - ▶ Interna der Klasse interessieren den „Verwender“ nicht
  - ▶ Interna können geändert werden, ohne dass der „Verwender“ seinen Code anpassen muss (solange das Verhalten „nach außen“ gleich bleibt)
  - ▶ Konsistenzprüfung bei settern
  - ▶ Träge Auswertung bei gettern

## Aufgabe 2: Mehr Sicherheit – Aufräumen

- Räume deinen Code von letztem Mal auf.
  - ▶ Er soll kompilieren und die Schatztruhe soll mit Passwort auf- und abschließbar sein.
  - ▶ Wenn das nicht geht: Zwischenstand in Moodle herunterladen.
- Ergänze Zugriffsmodifikatoren in Schatztruhe und Schatz.

(5 min)

# Aufgabe 3: Rollenspiel

- a) Vervollständige die Methoden in der Klasse `Monster`.
- b) Vervollständige die Methoden in der Klasse `Held`, sodass sich der Held mit den Pfeiltasten bewegen kann.
- c) Vervollständige die vier obersten Methode in der Klasse `Spiel`.
  - 1. Held verlässt das Spielfeld nicht
  - 2. Mehrere Monster sind auf dem Spielfeld platziert
  - 3. Der Held kann Monster bekämpfen (mit Leertaste)
- Erweitere das Spiel um weitere Elemente (und teste sie).
  - \* Rüstungspunkte für Monster
  - \*\* Held schaut in Richtung und greift nur in diese an
  - \*\*\* Schätze auf Karte platzieren mit Grafik
  - \*\*\*\* Verschiedene Angriffsarten, Anzeige von Lebenspunkten, Heiltränke, Waffen...

# Konstruktoren

- Bisheriges Pattern:

```
Schatz penny = new Schatz();  
penny.setzeWert(1);
```

- ▶ Erzeuge Objekt
- ▶ Setze Attribute mit setter-Funktion

- Ab jetzt: Konstruktoren

- `new Schatz()` ist Aufruf eines Konstruktors

```
public Schatz(int gewuenschterWert) {  
    wert = gewuenschterWert;  
}
```

```
Schatz penny = new Schatz(1);
```

- Refaktoriere deinen Code mit Konstruktoren, sodass Schatz und Monster ohne setter auskommen.



## Szenario

Eine Bank verwaltet mehrere Arten von Konten. Jedes Konto wird über seine ID identifiziert und hat einen Kontostand. Kunden können Geld einzahlen und ihren Kontostand abfragen.

Das Tagesgeldkonto ist ein Bankkonto, bei dem ein fester Zinssatz vereinbart wurde. Die Bank schreibt einem solchen Konto in regelmäßigen Abständen die Zinsen gut.

Modelliere das Szenario in einem UML-Klassendiagramm.

## Faustregel

Modelliert man `<Subklasse> extends <Superklasse>`, so muss für jedes Objekt X der Subklasse gelten: „X ist ein `<Superklasse>`“.