19. Zugriffsrechte

ARINKO

- Zugriff auf Programmelemente wie Typen, Methoden, Attribute etc. ist nur dann möglich, wenn ein Zugriffsrecht besteht.
- Andere Sichtweise: man kann Programmelemente nur benützen, wenn sie hinreichend **sichtbar** (visible) sind. (Die JLS spricht sogar von „accessible“)
- Diese Art der Zugriffskontrolle unterliegt komplett dem Compiler.
- Sie hat auch nichts mit Authentication oder Authorization von (Web-)Applikationen zu tun.
- Zugriffsrechte werden mit Hilfe von Zugriffsmodifizierern vergeben.

```
The method accessMeIfYouCan() from the type Zugriff is not visible
```

- Pakete sind immer sichtbar.
- Punkt.
- ...
- Aber...
- Voraussetzung natürlich, dass die Typen in den Paketen auch der virtuellen Maschine bekannt sind.
- Hierfür ist der **Class (Search) Path** zuständig.
- Meistens befinden sich alle Typen eines Paketes oder mehrerer zusammengehöriger Paketgruppen in einer JAR-Datei, die natürlich im jeweiligen Klassenpfad liegen muss.
- Ansonsten gibt es keine Zugriffsregelung für Pakete (außer man nutzt das Java 9 Modulkonzept).

- Die Sichtbarkeit eines Typs hängt vom Standort des Nutzers ab.
- Befindet sich der Nutzer im selben Paket, so hat er uneingeschränkt Sicht/Zugriff.
- Befindet sich der Nutzer in einem anderen Paket, dann muss der Typ mit dem Zugriffsmodifizierer `public` ausgezeichnet sein.

- Kurze Erinnerung, Zugriff auf Klassen eines anderen Pakets:

1. Genützte Klasse/Typ muss public sein (s.o.)
2. Nützende Klasse muss Typ unqualifiziert (Stern-Import) oder qualifiziert importieren oder mit voll-qualifiziertem Namen ansprechen

- Zugriff auf Elemente einer Klasse (Konstruktoren, Attribute, Methoden und innere Typen) werden mit 4 Zugriffsmodifizierern geregelt:
  - `private`
  - `protected`
  - *ohne Modifizierer - default visibility*
  - `public`
- Auf private Elemente kann prinzipiell von außerhalb nicht zugegriffen werden.
- Auf public Elemente kann prinzipiell von außerhalb zugegriffen werden.

- **Merke:** dies sind die zwei einfachsten Möglichkeiten der Zugriffssteuerung. Versuchen Sie immer, wo möglich mit diesen beiden auszukommen. Wo nötig kann dann eine der beiden anderen vergeben werden.

- Hier beeinflusst sehr wesentlich der „Standort“ der nutzenden Typs, ob und was er sehen darf.
- Erfolgt der Zugriff aus einem **Subtyp** oder von einem **Benutzer** des Typs?
- Erfolgt der Zugriff aus dem **selben Paket** oder aus einem **anderen Paket**?

- Eine Klasse ist Benutzer einer anderen Klasse, wenn Sie a) eine Instanz der anderen Klasse besitzt hat und auf deren Elemente zugreift oder b) wenn sie auf deren statische Elemente zugreift.
- Eine Klasse ist Subklasse einer anderen Klasse, wenn sie mit dieser in einer Vererbungsbeziehung steht.

- Objekte von Subklassen können in beiden Rollen auftreten!

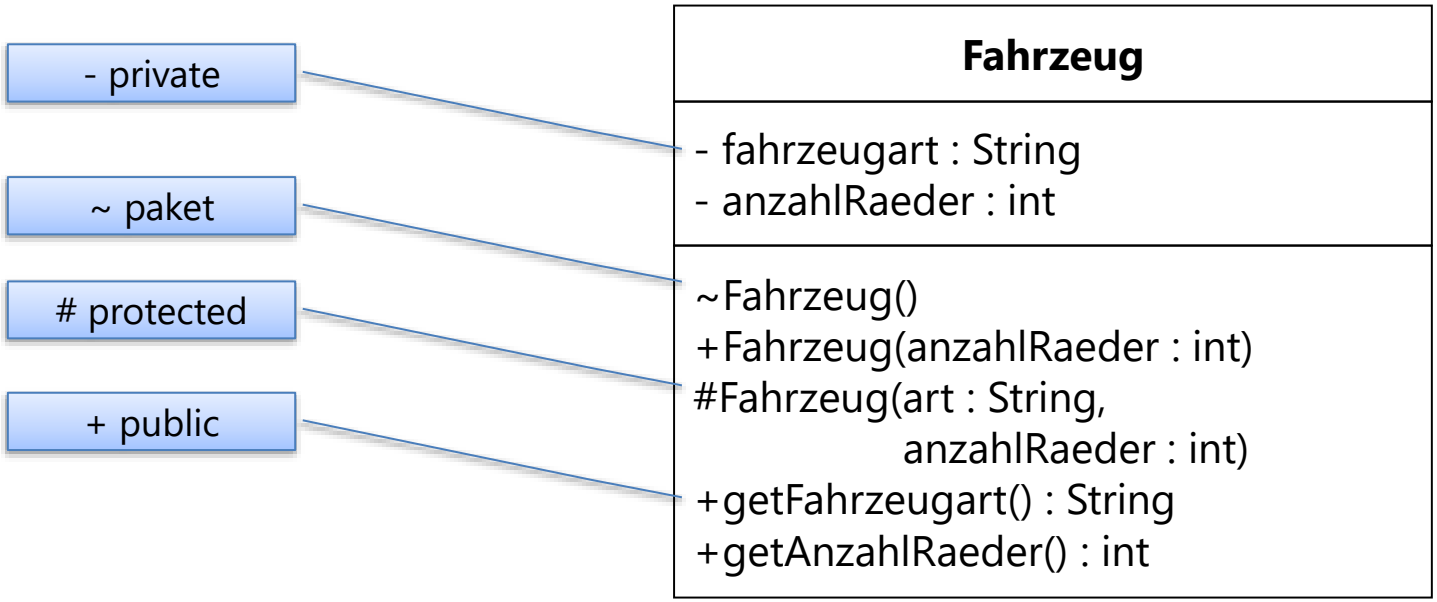
- protected Elemente sind für Code zugreifbar, der sich im selben Paket befindet (!)
- Unterklassen haben Zugriff auf protected Elemente ihrer Oberklassen, egal in welchem Paket sich die Klasse befindet.

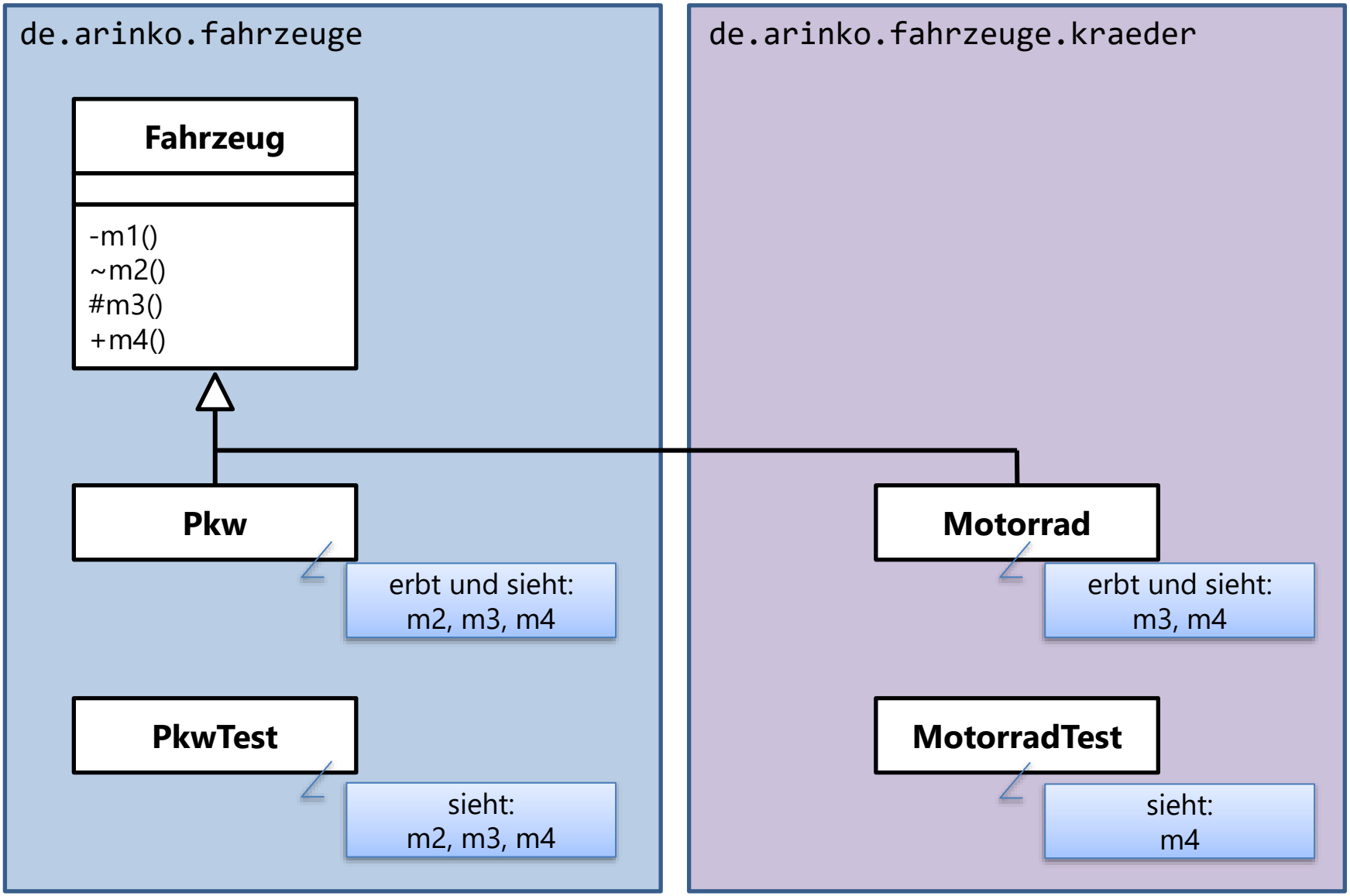
- „default visibility“ wird in der JLS auch „package access“ genannt. Wir sprechen von „Paketsichtbarkeit“.
- Auf Elemente mit Paketsichtbarkeit darf von überall im selben Paket zugegriffen werden.
- Auf Elemente mit Paketsichtbarkeit einer Oberklasse darf dann zugegriffen werden, wenn sich die Oberklasse im selben Paket befindet.

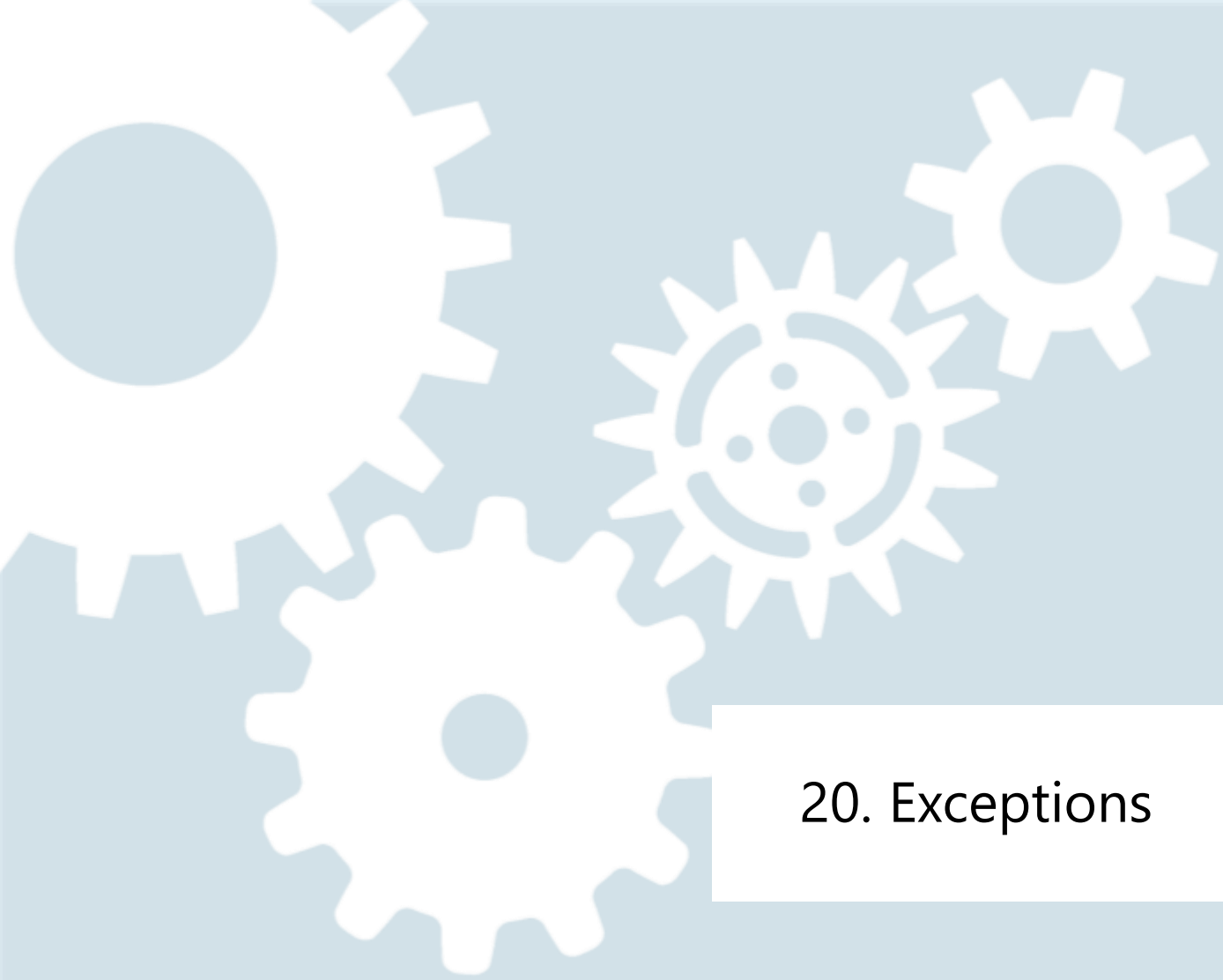
- Man bemerkt:
- protected schliesst die Paketsichtbarkeit irritierenderweise mit ein.
- Beide Vorgaben sind **nicht stabil** gegenüber Refactoring, das Typen in andere Pakete verschiebt.

Benutzer	im selben Paket	außerhalb des Pakets
`private`		
default		
`protected`		
`public`		

Subklasse	im selben Paket	außerhalb des Pakets
`private`		
default		
`protected`		
`public`		


- private
~ paket
# protected
+ public
Fahrzeug
- fahrzeugart : String
- anzahlRaeder : int
~Fahrzeug()
+Fahrzeug(anzahlRaeder : int)
#Fahrzeug(art : String,
anzahlRaeder : int)
+getFahrzeugart() : String
+getAnzahlRaeder() : int


de.arinko.fahrzeuge
Fahrzeug
-m1()
~m2()
#m3()
+m4()
Pkw
erbt und sieht:
m2, m3, m4
PkwTest
sieht:
m2, m3, m4
de.arinko.fahrzeuge.kraeder
Motorrad
erbt und sieht:
m3, m4
MotorradTest
sieht:
m4



20. Exceptions

ARINKO

- Exceptions signalisieren zur Runtime (also erst zur Laufzeit, nicht während der Kompilierung), dass eine ungewöhnliche Situation eine gesonderte Behandlung benötigt.
- Typische Beispiele:
  - Zugriffsversuch auf ein Objekt, obwohl die Referenzvariable nur null enthielt
  - Zugriff auf ein Array außerhalb seiner Grenzen
  - Cast auf einen Typ der nicht korrekt ist
  - Der Speicher ist alle
  - Eine Klasse konnte nicht dynamisch geladen werden
  - Eine Klasse enthält zur Laufzeit nicht mehr die Elemente, gegen die sie noch kompiliert worden war

- Anwendungen können aber auch solche Fehlersituationen kreieren:
  - Eine Warennummer ist ungültig
  - Ein Datum liegt außerhalb eines definierten Bereichs
  - Ein Befehl konnte nicht an ein zuständiges Subsystem abgesendet werden

- Die typische Fehlerbehandlung in Programmiersprachen, die kein Exceptions-Konzept haben, sieht so aus, dass Fehler durch Rückgabecodes geliefert werden.
- Beispiel einer Fehlerbehandlung in C:

```
int a = berechneBetrag();
if (a < 0)
{
  /* Fehler */
  switch (a)
  {
    case -1: /* Fehlerbehandlung für Errorcode -1 */; break;
    case -2: /* ... */
      /* ... */
  }
}
else
{
  /* Alles gut */
}
```

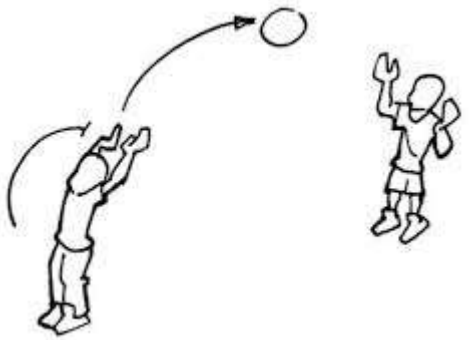
- Frage: Nachteil?

- Kapselung der Fehlerbehandlung
- Trennung von „normal“ ablaufendem Code und Fehlerbehandlungscode
- Code kann durch sog. „checked“ Exceptions auf sich aufmerksam machen und nicht-ignorierbare Fehler deklarieren
- Besser Klassifizierung (im wahrsten Sinne des Wortes: Exceptions sind ebenfalls Klassen)
- Propagation der Fehler, bis ein zuständiger Error-Handler gefunden wird
- Schnellere Abarbeitung (und bessere Lesbarkeit) von Programmteilen ohne Fehler

- Die meisten objektorientierten Sprachen liefern einen ähnlichen Mechanismus für den systematischen Umgang mit Ausnahme-/Fehlersituationen.
- Es wir das „throw-and-catch“-Paradigma verwendet.

- „Throw an exception“
  Es wird eine Fehlersituation signalisiert (es wird ein Fehler „geworfen“).
  Das geschieht an der Stelle im Programm, an dem der Fehler auftritt bzw. festgestellt wird.

- „Catch an exception“
  Die Fehlersituation wird registriert und behandelt (der Fehler wird „gefangen“).
  Das geschieht in der Regel in einer anderen Methode oder einer anderen Klasse.



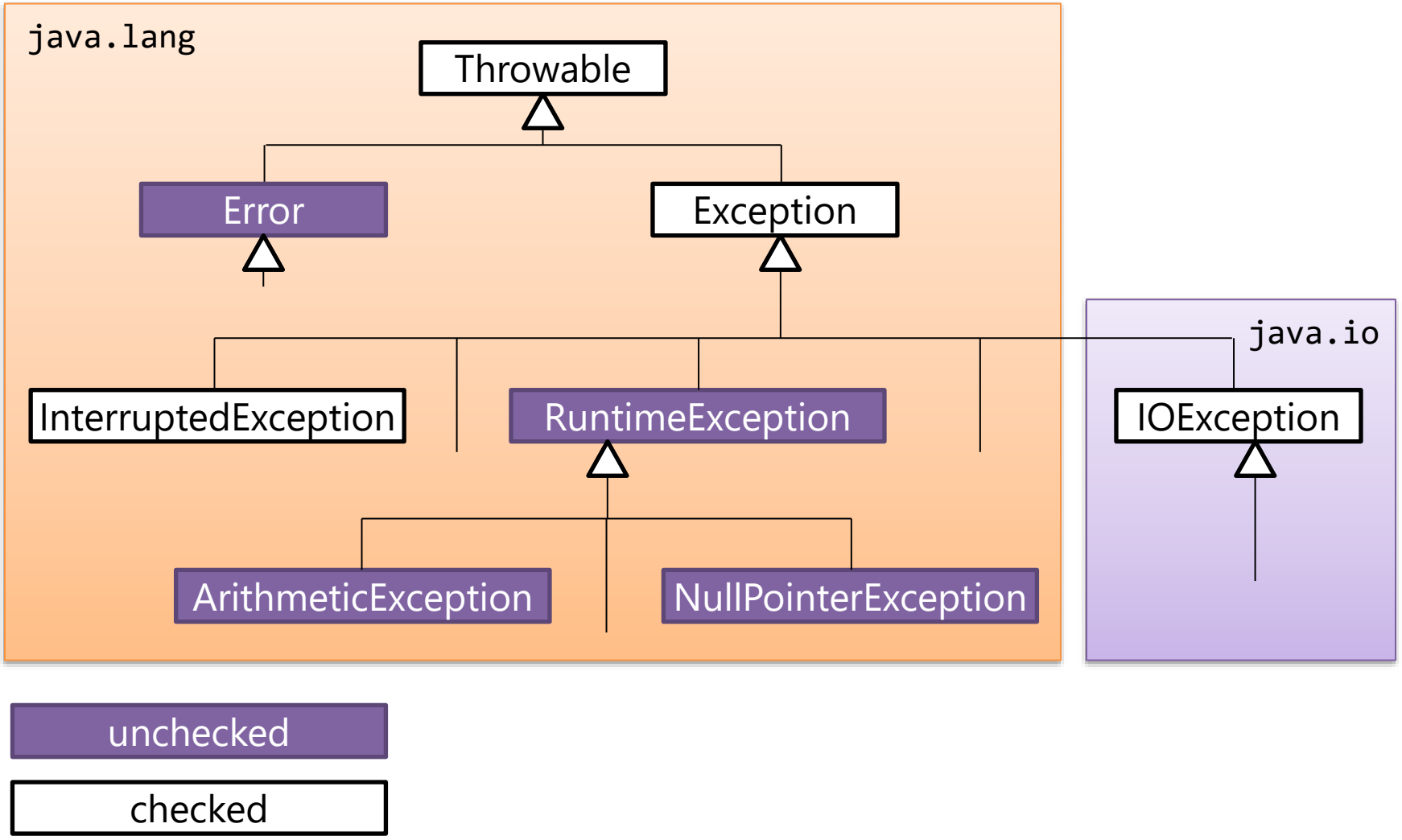
- „Exception werfen“ bedeutet: mit einer Java-Anweisung eine Ausnahmesituation signalisieren, der man ein Exception-Objekt mitgibt, das günstigerweise Informationen über die Ausnahmesituation trägt

  Beispielsweise: der Typname klassifiziert die Art der Ausnahme, Attribute und Methoden konkretisieren den genauen Fehler.

- „Exception fangen“ bedeutet: die Ausnahmesituation wird solange durch das Programm gereicht, bis ein geeigneter Exception-Handler gefunden wird.
- Wenn ein geeigneter Exception-Handler erreicht ist, gilt die Ausnahme als behandelt.
- Was genau vom Exception-Handler unternommen wird, obliegt dem Entwickler. Es kann somit günstige und weniger günstige Exception-Handler geben.

- Wird kein geeigneter Exception-Handler gefunden, erreicht der Fehler irgendwann die virtuelle Maschine. Diese reagiert, indem sie den Fehler protokolliert und den handelnden Thread beendet.

Auszug aus der Exception-Hierarchie



- checked Exceptions: der Compiler stellt sicher, dass Code, der anderen Code aufruft, welcher eine checked Exception auslöst, sich auch direkt oder indirekt mit der Exception befasst.
- unchecked Exceptions: der Compiler prüft nicht, ob eine Reaktion auf die Exception vorhanden ist.

- Vorteile von checked Exceptions:

1. Der Entwickler muss sich sofort auch um die Fehlerbehandlung kümmern.
2. Dadurch kann die Code-Stabilität und –Sicherheit erhöht werden.

- Nachteile von checked Exceptions:

1. Wenn der Entwickler sich sofort um die Fehlerbehandlung kümmern muss, nervt ihn das oft und er schreibt einen leeren Fehlerbehandlungsblock, der die Fehler für immer „verschluckt“.
2. Dadurch kann die Fehlererkennung und Code-Sicherheit drastisch verringert werden.

- Es ist zu beobachten, dass aus den Erfahrungen der vergangenen Jahre als Konsequenz in vielen Frameworks vermehrt auf unchecked Exceptions gesetzt wird.

- **unchecked Exceptions** sind alle Exceptions, die von den Klassen Error und RuntimeException abgeleitet sind.
- `Error`: i.d.R. nicht behandelbarer Fehler (Speichermangel, Code gegen falsche Java-Version compiliert, Code gegen falsche Schnittstellen compiliert…)
- `RuntimeException`: möglicherweise behandelbare Fehler (falscher Cast, durch 0 geteilt, auf nicht vorhandenes Objekt zugegriffen…)
- Die meisten dieser Exceptions werden von der virtuellen Maschine ausgelöst. Man kann sie aber auch im Anwendungscode auslösen.

- **checked Exceptions** sind alle anderen Exceptions, die nicht direkt oder indirekt von `Error` oder `RuntimeException` abgeleitet sind.
- checked Exceptions werden nach dem „throw-OR-catch“-Paradigma behandelt:

  man kann sie entweder fangen (d.h. behandeln) [„catch“]
  oder man wirft sie zur aufrufenden Methode weiter, damit diese die Behandlung übernimmt [„throw“]

- Das Exception-Handling in Java folgt diesem Muster:

```
versuche ob
{
  // code normal abläuft
}
fange Fehler im Fehlerfall für Fehlerart1
{
  // Behandlung für Fehlerart1
}
fange Fehler im Fehlerfall für Fehlerart2
{
  // Behandlung für Fehlerart2
}
schließlich führe immer aus
{
  // abschließende Routine
}
```

- In Java-Syntax sieht das wie folgt aus:

```
try
{
  // code, der normal abläuft
}
catch (Fehlerart1 f)
{
  // Behandlung für Fehlerart1
}
catch (Fehlerart2 f)
{
  // Behandlung für Fehlerart2
}
finally
{
  // immer abschließende Routine
}
```

f ist die Variable, über die das Exception-Objekt jetzt namentlich angesprochen werden kann

- `throw`:
  wirft eine Exception
- `throws`:
  Bestandteil des Methodenkopfs, der die Liste der geworfenen (checked) Exceptions deklariert
- `try-catch-finally`:
  Behandlung der Exceptions

- Exceptions werden im Java-Programm mit throw ausgelöst:

  `throw` *exceptionObjectReference*`;`

- Die Referenz muss auf einen Objekttypen verweisen, der zu `Throwable` kompatibel ist.
- `throw` unterbricht an Ort und Stelle die Ausführung des Programms und propagiert die geworfene Exception solange, bis ein geeigneter Exception-Handler gefunden wird.
- Es existiert neben dem normalen Kontrollfluss somit eine zweite Art Programmfluss.
- Ebenso kann eine Methode, neben dem „normalen“ Weg mit einem Rückgabewert, durch eine Exception verlassen werden.

- Nicht ganz sinnvolles Syntax-Beispiel:

```
if (a < 0)
{
  throw new IOException("a muss > 0 sein.");
}
```

- Die throws-Klausel ist das letzte Element im Methodenkopf.
- Allgemeine Form:

```
throws Exceptionart1, Exceptionart2, …, ExceptionartX
```

- Der Compiler überprüft, ob alle in der Methode geworfenen checked Exceptions, die nicht direkt behandelt werden, auch im Methodenkopf deklariert wurden.
- Unchecked Exceptions können ebenfalls deklariert werden, müssen aber nicht.

- Ebenso nicht ganz sinnvolles Syntax-Beispiel:

```
public static void createsException(int a) throws IOException
{
  if (a < 0)
  {
    throw new IOException("a muss >= 0 sein.");
  }
}
```

- try:
  Alle Anweisungen, die im Idealfall ohne Exception hintereinander ablaufen.
- catch-Klauseln:
  einzelne Exception-Handler, die die unterschiedlichen Ausnahmesituationen behandeln
- finally-Klausel:
  Anweisungen, die in jedem Fall ausgeführt werden

- Jede catch-Klausel steht für einen unterschiedlichen Exception-Typ.
- Jeder Typ darf nur einmal vorkommen.
- Catch-Klauseln funktionieren polymorph, d.h. man kann durch Fangen eines Oberklassentyps eine ganze Teilhierarchie an Fehlern behandeln.
- Speziellere Typen müssen vor allgemeinen gelistet sein.

- Die finally-Klausel wird immer ausgeführt, unabhängig ob ein Fehler vorlag oder nicht.
- Entweder ist die finally-Klausel vorhanden oder mindestens eine catch-Klausel oder beide.
- Nur im Fall der Sonderform des „try-with-resources“ dürfen beide entfallen.

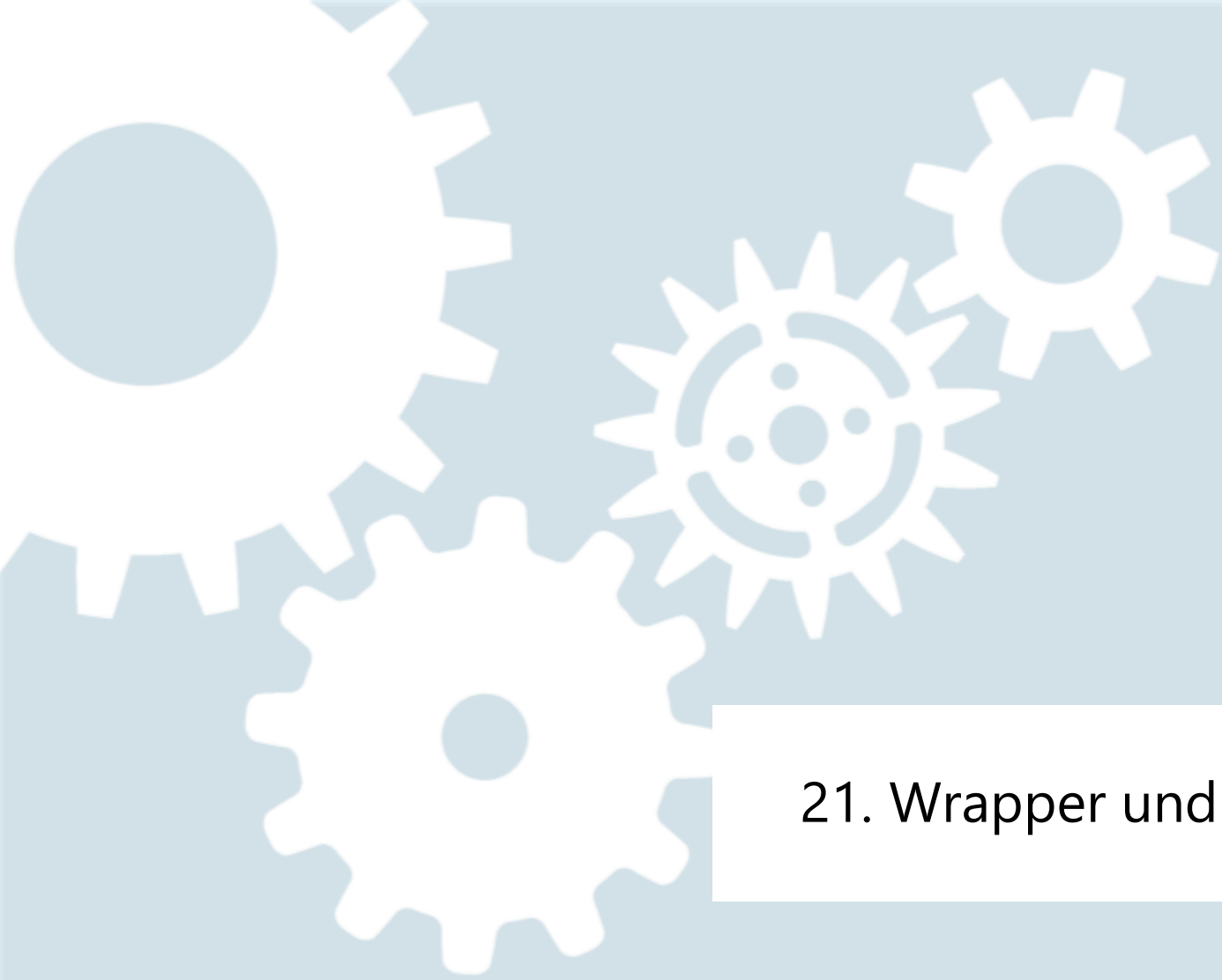
- throw unterbricht sofort die Ausführung des Programms (genauer: des Threads) an der entsprechenden Stelle und macht sich auf die Suche nach einem passenden Exception-Handler.
- Wird in der aktuellen Methoden kein Exception-Handler gefunden, dann wird der Call-Stack nach unten abgearbeitet, bis ein geeigneter Handler gefunden wird.

  Hinweis: bei „checked“ Exceptions wird auf jeden Fall ein geeigneter Handler gefunden, das hat der Compiler bereits sichergestellt.

- Wird so kein geeigneter Exception-Handler gefunden, wird die Exception schließlich zur virtuellen Maschine propagiert. Diese reagiert mit der Ausführung des Default Exception-Handlers.
- Der Default-Exception-Handler (ist i.ü. austauschbar) protokolliert den Fehler und gibt einen sog. Stack-Trace aus.
- Weiterhin beendet der Default-Exception-Handler daraufhin den problematischen Thread.

- Man kann in seinen Programmen eigene Exceptions verwenden.
- Je nachdem man von `Exception` oder `RuntimeException` ableitet, erzeugt man checked oder unchecked Exceptions.
- Ebenso darf man sich bereits vorhandener Exceptionklassen bedienen, sollte aber darauf achten, dass sie sinnvoll verwendet werden.
- Beispiele für Java Exceptions, die gerne wiederverwendet werden

  `Exception` (als allgemeine, aber doch sehr unqualifizierte Fehlersituation)
  `IllegalArgumentException`
  `IllegalStateException`
  `UnsupportedOperationException`
  `NumberFormatException`
  u.v.m.



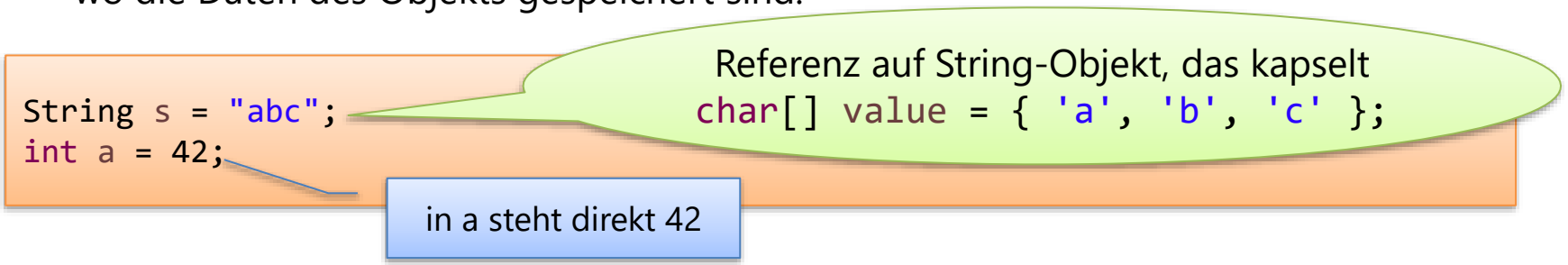
21. Wrapper und Boxing

ARINKO

- Um verallgemeinerte Dienste anzubieten, benützen viele Klassen den allgemeinen Inhaltstyp `Object`.
- Das praktische ist: jedes Objekt passt typmäßig zu `Object`.
- (Das unpraktische ist: jedes Objekt muss man wieder auf einen spezielleren Typ casten, wenn man damit sinnvolle Dinge tun will – anderes Problem, andere Baustelle).
- Noch ein Problem:

nicht alles in Java ist ein Objekt!

- Die elementaren Typen wurden aus Performancegründen in Java nicht prinzipiell objektorientiert gebildet.
- Variablen von elementaren Typen enthalten direkt den betreffenden Wert.
- Variablen von Referenztypen enthalten die Referenz, die auf eine Speicherstelle zeigt, wo die Daten des Objekts gespeichert sind.



- Damit elementare Daten als Objekt verwendet werden können, gibt es zu jedem Typ einen korrespondierenden Wrapper-Typ.
- Ein Wrapper (Umhüller) hüllt den elementaren Wert in einem Objekt ein.

```
Integer a = new Integer(42);
```

Referenz auf Integer-Objekt, das kapselt
`int value = 42;`

elementarer Typ	Wrapper
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

- Die Wrapper-Klassen bieten einige interessante, zumeist statische Methoden, für den Umgang mit elementaren Typen an.

```
int a   = Integer.parseInt("42"); //   a = 42
int max = Integer.max(23, 42);    // max = 42
int min = Integer.min(23, 42);    // min = 23
int sig = Integer.signum(-42);    // sig = -1

boolean c = Character.isAlphabetic('a');  // true
boolean s = Character.isWhitespace('\n'); // true
boolean m = Character.isMirrored('{');    // true { -> }

String  t = Boolean.toString(true);         // "true"
boolean b = Boolean.parseBoolean("True");   // true
boolean f = Boolean.parseBoolean("yes");    // false
```

- Achtung: `Double.parseDouble()` arbeitet nicht internationalisiert.

```
double d = Double.parseDouble("42,3"); // NumberFormatException
```

- Boxing ist das Verpacken eines Wertes in ein Objekt (hier: das Verpacken eines elementaren Wertes in ein Wrapper-Objekt).
- Unboxing ist der gegenteilige Vorgang.
- Herkömmlicher Java-Code, der Boxing und Unboxing dem Programmierer überlässt:

```
Integer boxed = new Integer(42); // Verpacken
int unboxed = boxed.intValue();  // Entpacken
```

- Autoboxing ist nun ein Feature (seit Java 5), das nach gewissen Regeln versucht, obige Konversionen automatisch durchzuführen.
- Dadurch kann der Eindruck entstehen, dass in Java auch elementare Werte Objekte seien. Der Eindruck ist falsch. Am Grundprinzip ändert sich nichts. Es ist lediglich einfacher für den Entwickler.

- Obwohl a vom Typ `int` ist, kann die Methode aufgerufen werden.

```
int a = 42;
benutzeReferenztyp(a);
```

```
public static void benutzeReferenztyp(Integer i)
{
  if (i != null)
  {
    System.out.println(i);
  }
}
```

- Man sieht bereits einen prinzipiellen Unterschied: Wrapper können `null` sein, elementare Werte niemals.
- Mit einem Java-Compiler <V1.5 compiliert o.g. Code nicht.
- Es würde dann ungefähr eine Fehlermeldung folgender Art produziert:

```
benutzeReferenztyp(java.lang.Integer) cannot be applied to (int)
```

- Der Compiler versucht, bevor ein Compilefehler bei elementaren Typen erzeugt wird, den Typ in sein korrespondierendes Wrapper-Objekt einzuhüllen.
- Man kann sich das so vorstellen (nicht ganz wahr):

```
Integer a = 42; // provoziert Fehler also versuche
Integer a = new Integer( 42 );
```

- Die Reihenfolge ist aber:

1. in den korrespondierenden Wrapper einhüllen
2. dann ggf. Typweitung (Typkompatibilität)

- Das erzeugt gelegentlich Überraschungen:

```
Double d = 4; // Fehler, int -> Integer <!> Double
Object o = 4; // Korrekt, int -> Integer -> Object

Integer i = 1;
long x = i + 2; // Korrekt, x = 3, int + int -> long

Long y = 0; // Fehler, int -> Integer <!> Long
long z = new Integer(0); // Korrekt, Integer -> int -> long
```

- Vorsicht beim Vergleich von Wrapperobjekten!

```
int a1 = 42;
int a2 = 42;
System.out.println(a1 == a2); // true, klar

Integer i1 = 42;
Integer i2 = 42;
System.out.println(i1 == i2); // true..., warum?

Integer j1 = 128;
Integer j2 = 128;
System.out.println(j1 == j2); // false..., warum, wenn oben true?
```

- Der Compiler verwendet eben nicht

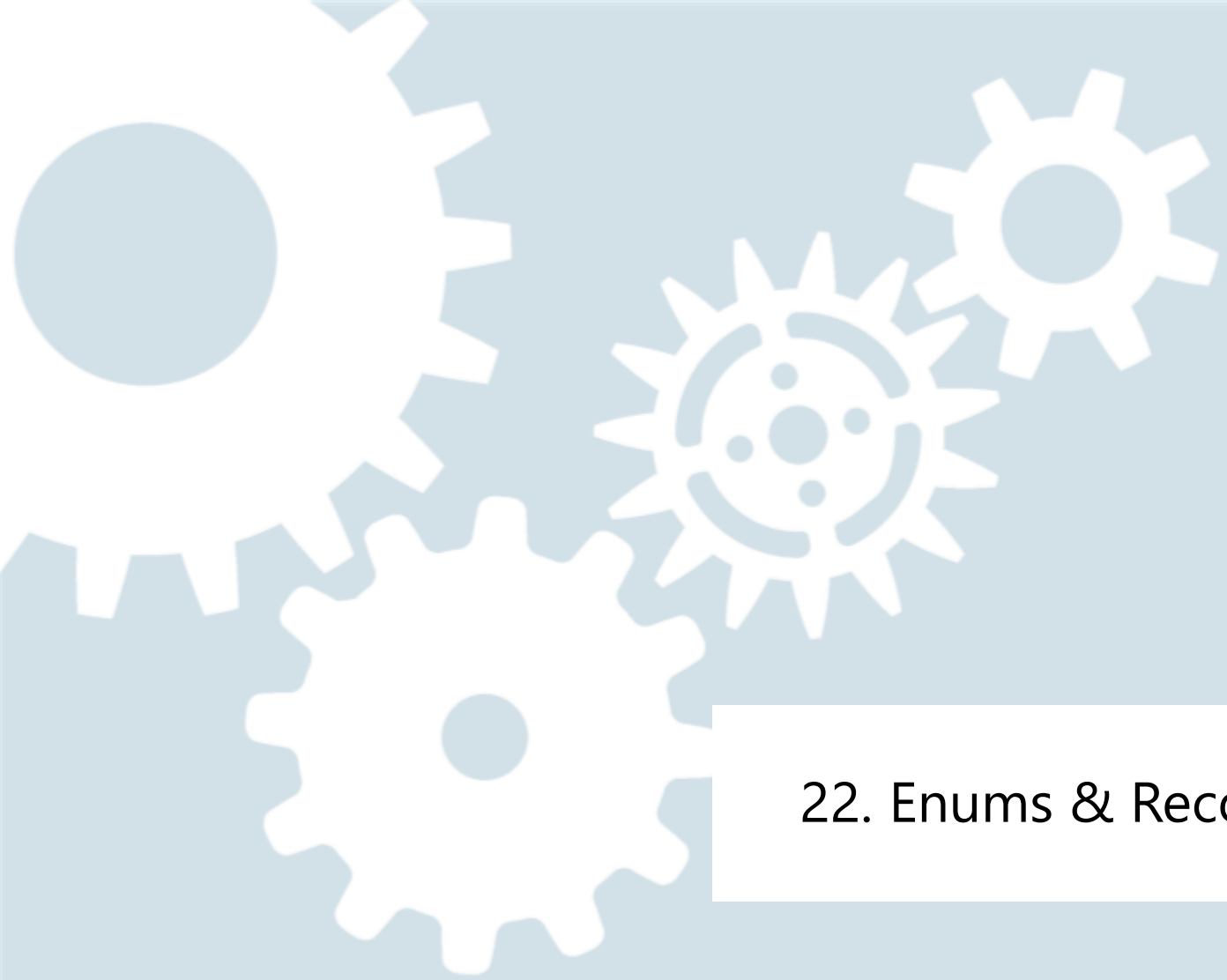
  `new Integer( int )`

  was immer `false` beim Vergleich mit == erzielt hätte.
- Stattdessen wird die statische Methode

  `Integer.valueOf( int )`

  eingesetzt.
- Die `valueOf`-Methoden der Wrapper-Klassen liefern im Wertebereich
  `true…false`
  `-128 – +127`
  `ASCII-Zeichen`
  gecachte identische Objekte zurück.
- Außerhalb des Wertebereichs wird mit `new` gearbeitet. Aus der Klasse `Integer`:

```
public static Integer valueOf(int i) {
  if (i >= IntegerCache.low && i <= IntegerCache.high)
    return IntegerCache.cache[i + (-IntegerCache.low)];
  return new Integer(i);
}
```



22. Enums & Records

ARINKO

- Aufzählungen sind in der Programmierung keine Seltenheit.
- Oft werden disjunkte Systemzustände oder Optionen angeboten.
- In früheren Java-Versionen wurden hierzu gerne symbolische Konstanten verwendet, die mit einem beliebigen Typ assoziiert wurden, z.B. so:

```
public interface Ampelfarben
{
  public static final int ROT = 1;
  public static final int GELB = 2;
  public static final int GRUEN = 3;
}
```

- Unschön ist es, dass a) der Typ an sich nichts aussagt und b) beim Verwenden es Unklarheiten gibt.

```
public class Ampel
{
  public void stelleSignal(int farbe)
  {
    …
  }
}
```

- Viele Designer und Architekten haben sich mit einem Design-Pattern beholfen, das einige Grundprobleme aus der beliebigen Verwendung von symbolischen Konstanten behebt.
- Hier die Variante für die Ampelfarbe:

```
public class Ampelfarbe
{
  public static final Ampelfarbe ROT = new Ampelfarbe("rot");
  public static final Ampelfarbe GELB = new Ampelfarbe("gelb");
  public static final Ampelfarbe GRUEN = new Ampelfarbe("grün");

  private String farbname;

  private Ampelfarbe(String farbname)
  {
    this.farbname = farbname;
  }

  public String getFarbname()
  {
    return this.farbname;
  }
}
```

das sind die einzigen existierenden Instanzen

weitere gibt es nicht, da der Konstruktor private ist

- Im Gegensatz zur Lösung mit der int-Konstante, die folgenden Aufruf ermöglicht

```
Ampel ampel = new Ampel();
ampel.stelleSignal(15); // stelleSignal hat Eingangstyp int…
```

- …ist mit dem Einsatz des Patterns die Typ- und Wertsicherheit wiedergekehrt:

```
Ampel ampel = new Ampel();
ampel.stelleSignal(Ampelfarbe.GRUEN); // stelleSignal ist auf Ampelfarbe
                                      // typisiert
```

- Da man den Entwickler dazu zwingt, die typisierten symbolischen Konstanten zu verwenden, kann man sogar bei Vergleichen das performantere == verwenden.

```
public void stelleSignal(Ampelfarbe farbe)
{
  if (farbe == Ampelfarbe.ROT){…}
}
```

- Leider arbeiten hier so geeignete Sprachkonstrukte wie das `switch` nicht mit dem Design-Pattern zusammen

- Mit Java 5 wurde das Design Pattern als Sprachbestandteil integriert.
- Durch die Integration kann mehr geboten werden.

Enums in Java:

- Neues Schlüsselwort `enum`
- Strenge Typisierung
- Integration in das `switch-Statement`, inkl. separater Namensräume für das `switch-Statement` (die `case-Marken` beziehen sich direkt auf das Enum)
- Default Output bei Ausgaben
- Enums sind Objekte, keine Zahlen oder anderes, sie können daher z.B. auch in Collections verwendet werden (Keys in Maps…)
- Enums eben implizit von `java.lang.Enum` und besitzen 2 statische "convenience" Methoden, `values()` und `valueOf()`.

- Nicht jede Programmiersprache, die Enums umsetzt, arbeitet hier mit einem typisierten Konzept.
- Für C++ sind Enums Zahlenwerte, die noch Typinformation tragen.
- Auch C# arbeitet unterschiedlich (typischerweise auch hier Zahlen, man kann aber auch darüber hinausgehen)

- Hier ein Beispiel für ein Enum, das Euro-Münzen aufzählt und dabei noch ihren Cent-Wert als Daten trägt:

```
public enum Muenze
{
  EIN_CENT(1), ZWEI_CENT(2), FUENF_CENT(5),
  ZEHN_CENT(10), ZWANZIG_CENT(20), FUENFZIG_CENT(50),
  EIN_EURO(100), ZWEI_EURO(200);

  // Wert wird intern in Cent gehalten
  private final int wert;

  private Muenze(int wert)
  {
    this.wert = wert;
  }

  public int getWert()
  {
    return this.wert;
  }
}
```

die benannten Instanzen müssen als erstes im Enum auftauchen

weitere Elemente wie in einer normalen Klasse

Der Konstruktor muss aber analog zum Pattern versteckt sein

```
public static String farbe(Muenze muenze)
{
  switch (muenze)
  {
    case EIN_CENT:
    case ZWEI_CENT:
    case FUENF_CENT:
      return "Kupfer";

    case ZEHN_CENT:
    case ZWANZIG_CENT:
    case FUENFZIG_CENT:
      return "Messing";

    case EIN_EURO:
      return "Silber mit goldenem Rand";

    case ZWEI_EURO:
      return "Gold mit silbernem Rand";

    default:
      throw new IllegalArgumentException("Muenze existiert nicht");
  }
}
```

hier dürfen jetzt nur Münzen rein...

Konstante aus dem Namensraum, man schreibt nicht `Muenze.EIN_CENT`

Kann eigentlich nie passieren... wann aber doch?

- Mit der Methode `values()` kann man alle Enum-Instanzen eines Typs als Array erhalten.
- Das ist beispielsweise ideal für Testläufe wie hier:

```
Muenze[] muenzen = Muenze.values();
for (Muenze muenze : muenzen)
{
  System.out.printf("%s (%d Cent) : %s%n",
                    muenze, muenze.getWert(), farbe(muenze));
}
```

```
EIN_CENT (1 Cent) : Kupfer
ZWEI_CENT (2 Cent) : Kupfer
FUENF_CENT (5 Cent) : Kupfer
ZEHN_CENT (10 Cent) : Messing
ZWANZIG_CENT (20 Cent) : Messing
FUENFZIG_CENT (50 Cent) : Messing
EIN_EURO (100 Cent) : Silber mit goldenem Rand
ZWEI_EURO (200 Cent) : Gold mit silbernem Rand
```

- Mit der Methode `valueOf()` kann man ein Enum auf Grund seiner String-Repräsentation erhalten.
- Kann nützlich sein für Eingabetests oder wenn der Wert als Plaintext in Dateien oder einer Datenbank steht.

```
Muenze muenze = Muenze.valueOf("EIN_EURO");
System.out.println(farbe(muenze));

muenze = Muenze.valueOf("ZEHN_EURO");
System.out.println(farbe(muenze));
```

```
Silber mit goldenem Rand
Exception in thread "main"
java.lang.IllegalArgumentException: No enum constant
Muenze.ZEHN_EURO
```

- Diese `IllegalArgumentException` kommt nicht aus der `farbe()`-Methode, sondern von `valueOf()`.

Enums können relativ ähnlich zu normalen Klassen aufgebaut werden:

- Konstruktoren dürfen überladen und innerhalb des Enums verkettet werden.
- Sie können über Methoden und Attribute verfügen.

Ergänzungen/Zusätze:

- Die Enum-Konstanten müssen als erstes in der Typdeklaration auftauchen.
- Sie können "convenient" abgekürzt werden, wenn es keine Konstruktorargumente gibt, d.h. `GRUEN` ist dasselbe wie `GRUEN()`.
- Alle Enums verfügen über passende `toString()`, `hashCode()`, `equals()`-Methoden und sind `Comparable` und `Serializable`.

Verbote:

- Enums mit `new` anlegen ist logischerweise verboten, auch Klonen ist nicht erlaubt.
- Konstruktorverkettung zur Oberklasse ist nicht erlaubt.
- Konstruktoren dürfen nicht `public` oder `protected` sein. Fehlt der Modifizierer, dann gilt hier `private`!
- Keine abstrakten Methoden und kein `finalize()`.
- Keine Bezugnahme auf statische non-final Elemente während der Objektkonstruktion (die Reihenfolge ist bei Enums etwas anders).

- `public static enumtyp valueOf(String s)`
  Gibt die Enum-Konstante mit dem übereinstimmenden Namen zurück. Wenn eine solche nicht existiert, wird eine `IllegalArgumentException` geworfen.
- `public static enumtyp[] values()`
  Gibt alle Enum-Werte in der Reihenfolge ihrer Deklaration in einem Array zurück.
- `public final int ordinal()`
  Liefert den Aufzählungswert, beginnend mit 0, der Enumkonstante. Das ist das nächste, was an C++-Enums noch erinnert.
- `public final int compareTo()`
  Methode von `Comparable`. Vergleicht Enums auf Basis ihres Ordinals.
- `public final Object clone()`
  Wirft immer eine `CloneNotSupportedException`

- `hashCode()` und `equals()` sind ebenfalls nicht überschreibbar.
- Nur `toString()` ist als implizit implementierte Methode noch veränderbar.

- Java 17 definiert als Neuerung sog. „Records“.
- Aus der Motivation der Spezifikation für die Erweiterung:

> It is a common complaint that "Java is too verbose" or has "too much ceremony". Some of the worst offenders are classes that are nothing more than immutable data carriers for a handful of values. Properly writing such a data-carrier class involves a lot of low-value, repetitive, error-prone code: constructors, accessors, equals, hashCode, toString, etc.
>
> https://openjdk.org/jeps/395

- Statt Konstruktoren und Infrastrukturcode für unveränderbare Objekte zu codieren, kann man nun Records definieren:

```
public record Punkt(int x, int y)
{}
```

- Der gezeigte Record

```
public record Punkt(int x, int y) {}
```

erzeugt im Prinzip dieses Konstrukt:

```
public class Punkt
{
  private final int x;
  private final int y;
  public Punkt(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  public int x() { return this.x; }
  public int y() { return this.y; }
  public String toString() {
    return String.format("Punkt [x=%d, y=%d]", this.x, this.y); }
  public boolean equals(Object o) {
    return this.x == ((Punkt)o).x && this.y == ((Punkt)o).y; }
  public int hashCode() { return Objects.hash(this.x, this.y); }
}
```

Records müssen etwas abweichend zu normalen Klassen aufgebaut werden, aber folgendes ist erlaubt:

- Instanz-Methoden
- Statische Methoden, Attribute und deren Initialisierer
- Generics sind möglich
- Interface-Implementierung

Ergänzungen/Zusätze:

- Alle Records verfügen implizit über passende `toString()`, `hashCode()`, `equals()`-Methoden und sind `Serializable`.
- Records sind implizit `final` und können daher nicht `abstract` sein.
- Die Zugriffsmethoden für die Daten haben direkt den Namen der Daten und sind nicht mit get/set gepräfixt.

Verbote:

- Eigene Instanzattribute sind verboten. Die im Klassenkopf genannten Daten sind die einzigen Objektattribute.
- Kein `extends`. Die implizite Superklasse ist immer `java.lang.Record`.

- Nutzung des eben definierten Records:

```
Punkt p1 = new Punkt(5, 12);
Punkt p2 = new Punkt(5, 12);
System.out.printf("(%d,%d)%n", p1.x(), p1.y());   // (5,12)
System.out.println(p2);                           // Punkt[x=5, y=12]
System.out.println(p1.equals(p2));                // true
System.out.println(p1.hashCode()==p2.hashCode()); // true
```

- Konstruktorcode wie beispielsweise Verifizierungen und Bereichsprüfungen kann vereinfacht angeboten werden („compact canonical constructor“):

```
public record Punkt(int x, int y)
{
  public Punkt
  {
    if (x < 0 || y < 0)
      throw new IllegalArgumentException();
  }
}
```

Implizit bekannte Parameter

Zuweisungen passieren immer noch automatisch und müssen nicht selbst codiert werden


Praxis
Theorie