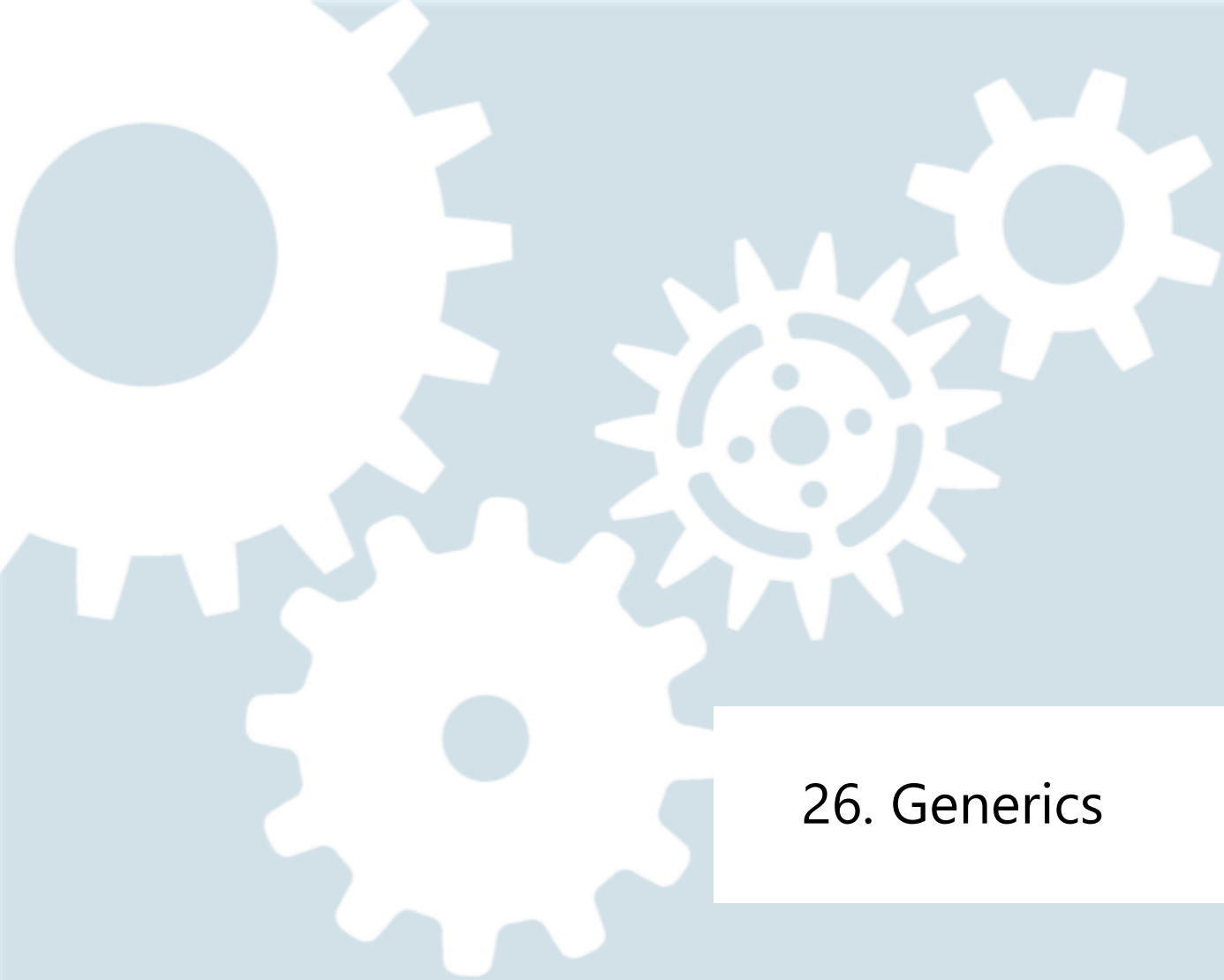


26. Generics

ARINKO

- Die Problematik mit Collections bislang ist, dass sie quasi untypisiert benutzt werden müssen.
- Der Inhaltstyp Object kann alles oder nichts sein. Der Entwickler muss wissen, was er in die Collection gelegt hat und als was er es herausholen darf.

```
HashSet hs = new HashSet();
// Collection füllen... hier ist der Inhaltstyp String bekannt
hs.add("eins");
hs.add("zwei");
hs.add("drei");

// dann iterieren
Iterator iter = hs.iterator();
while (iter.hasNext())
{
  // hier ist der Inhaltstyp Object
  Object e = iter.next();
  // Wenn man mit String arbeiten will, muss man casten
  String s = (String) e;
  System.out.println(s.toUpperCase());
}
```

- Wenn man verallgemeinernde Konzepte in Java bauen will, muss man die Schnittstelle reduzieren (Tradeoff Flexibilität/Wiederverwendung vs. Spezialisierung).
- Collections sind ein solch verallgemeinerndes Konzept. Damit alle Objekte in eine Collection gelegt werden können, wurde die Schnittstelle fast vollständig reduziert.
- Das bedeutet: keine Typprüfung durch den Compiler mehr – die Problematik wird auf die Laufzeit verschoben (ClassCastException). Das nimmt viel von der Programmqualität, Les- und Wartbarkeit und Sicherheit.
- Bis Java 1.4 hat man das zumeist wie folgt gelöst:
  - Collections, da flexibel, wurden als Behälter innerhalb von Klassen gekapselt
  - Arrays, da typsicher, bilden die Außenschnittstelle
- Das bedeutet aber auch, dass ständig umkopiert werden muss und somit Datenstrukturen mehrfach im Speicher vorkommen.

- Unbefriedigend…

- Falls es zukünftig doch mal notwendig sein sollte… Das Umkopieren von Collections in Arrays und vice versa ist recht einfach, mit einer kleinen Hürde.

1. Vom Array zur Collection – ganz einfach über die Klasse Arrays:

```
String[] array = { "a", "b", "c" };
List strings = Arrays.asList(array);
```

2. Von der Collection zum Array – hier existieren an den Collections die Methoden `toArray()` und `toArray(Object[])`.

   `toArray()` liefert immer ein `Object[]`. Hier hat man aus Typsicht nichts gewonnen.

   `toArray(Object[])` nimmt ein bestehendes Array, untersucht dessen Typ (!), füllt das gegebene Array, falls die Größe passt oder baut ein neues Array mit diesem Typ und gibt es unter der Flagge `Object[]` zurück. *Dieses Array kann gecastet werden.*

```
String[] s1 = (String[]) strings.toArray(); // ClassCastException

String[] s2 = new String[0]; // oder schon passende Größe
s2 = (String[]) strings.toArray(s2); // hier klappt der Cast
```

- Java 5 greift die Problematik auf und bietet mit Generics eine generelle, nicht nur auf Collections beschränkte, Lösungsmöglichkeit.
- Diese ist ausschließlich auf den Compiler fokussiert, zur Laufzeit ändert sich nichts.
- Dies soll

  wartbaren
  besser lesbaren
  typsicheren
  robusteren

  Code ermöglichen.
- Außerdem bleibt Java rückwärtskompatibel (das war eine der Grundbedingungen für die Spezifikation).
- Allerdings ist dabei ein recht komplexes Gebäude entstanden…

- Grundgedanke der Generics ist, dass Klassen parametrisiert werden können.
- Diese speziellen Parameter können für Vorkommen eines anderen, beliebigen Typs innerhalb der Klasse stehen.
- Beispiel für die Verwendung von Generics:

```
List<String> strings = new ArrayList<String>();
strings.add("a");
strings.add("b");
strings.add("c");

for (String string : strings) // Inhaltstyp ist String, nicht Object!
{
  System.out.println(string.toUpperCase());
}
```

- Und das hier erzeugt einen Compile-Fehler:

```
strings.add(new Integer(1));
```

```
The method add(String) in the type List<String> is not applicable for
the arguments (Integer)
```

```
public class GenerischerTyp<T1, T2, ...>
{
}
```

- T1, T2… sind **formale Typ-Parameter** oder auch Typ-Variablen.
- Mit … ist angedeutet, dass es eine ganze Liste von diesen Parametern geben kann.
- Jeder formaler Typ-Parameter steht für einen beliebig, aber festen Referenz-Typ, also Klasse, Interface, Enum oder Array.
- Geltungsbereich der o.g. Deklaration ist der gesamte Klassenrumpf, aber ausschließlich aller statischen Elemente.
- Typ-Parameter können noch Einschränkungen für den tatsächlich zu verwendenden Typ haben:

```
public class GenerischerTyp<T extends C & I1 & I2>
```

- Das erste Element kann ein Klassen- oder Interfacetyp sein.
- Die weiteren Elemente müssen Interfaces sein.
- Bei der Verwendung muss der konkret eingesetzte Typ alle Bedingungen erfüllen.

```
public interface Collection<E> extends Iterable<E>
{
  Iterator<E> iterator();
  boolean add(E e);
  // ...
}
```

- Bedeutet: ein einmal gewählter Typ kann dann per add hinzugefügt werden und man bekommt einen Iterator, der genau diesen Typ beinhaltet.

```
public interface Map<K, V>
{
  V put(K key, V value);
  Set<K> keySet();
  Collection<V> values();
}
```

- Schönes Beispiel für 2 Typvariablen.
- `keySet` liefert ein Set mit dem Key-Typ.
- `values` liefert eine Collection mit dem Value-Typ.

```
public class DelayQueue<E extends Delayed>
{
…
}
```

- DelayQueue findet sich im Paket java.util.concurrent.
- Was sagt die Einschränkung aus?

  E definiert einen (nahezu) beliebigen Inhaltstyp der Queue, der aber die Bedingung erfüllen muss, das Interface Delayed zu implementieren.
- Offenbar benötigt die Queue diese Eigenschaft.
- Man kann davon sprechen, dass dies eine Mindestgarantie ist.

- Kurzer Hintergrund: DelayQueues arbeiten ihre Elemente als Queue ab, aber nur, wenn ein gewisse zeitliche Verzögerung ("delay") bereits am Element passiert ist. Ansonsten wird das Element von der Queue noch nicht als entnehmbar angeboten.

- Bei der Verwendung von generischen Typen müssen alle Typ-Parameter mit konkreten Typen belegt werden.
- Das ist dann ein parametrisierter Typ.
- Die Einschränkungen für die Typvariablen müssen voll beachtet werden.
- Beispiele:

```
Collection<String> c1 = new ArrayList<String>();
Map<Integer,String> m1 = new HashMap<Integer,String>();

DelayQueue<String> dqs = new DelayQueue<String>(); // Fehler
```

String ist nicht Delayed

- Links und Rechts stehen derzeit bei der Deklaration und Initialisierung von parametrisierten Typen immer dieselben Typvariablen.

```
Collection<String> c1 = new ArrayList<String>();
```

- Das ist typischer "boilerplate code" (Code, der da sein muss, aber keinen Mehrwert besitzt) und hat viele Entwickler gestört.
- Ab Java 7 kann die rechte Seite auf die Typangabe verzichten:

```
Collection<String> c1 = new ArrayList<>();
Map<Integer,String> m1 = new HashMap<>();
```

- Die leeren Typklammern bilden ein Karo, das im Englischen als "diamond" bezeichnet wird, daher "diamond operator".
- Man beachte: ohne Diamond-Operator ist es ein anderes Phänomen!

```
Collection<String> c1 = new ArrayList(); // Compiler-Warnung
```

- Generics sind prinzipiell als rückwärtskompatibles Phänomen in Java eingeführt worden.
- Somit ist die Verwendung aller Klassen (v.a. von denen, die es bereits seit vielen Jahren in der Java-Plattform gab) auch ohne Generics erlaubt.

```
Collection c1 = new ArrayList(); // erlaubt, erzeugt aber Warnung
```

- Die Verwendung von generischen Typen als nicht-parametrisierte Typen erzeugt allerdings eine Compiler-Warnung (aber nur eine Warnung, es ist kein Fehler!)

```
Collection is a raw type. References to generic type Collection<E>
should be parameterized
```

- Generische Typen, die nicht generisch benutzt werden, nennt man "raw types" ("rohe Typen")
- Generische Varianten eines Typs sind immer zu seinem Raw-Type kompatibel:

```
Collection c1 = new ArrayList<String>();
```

- Das erzeugt immer noch eine Warnung, ist aber unproblematisch.

- Spannend ist der umgekehrte Fall:

```
Collection<String> c1 = new ArrayList(); // uh…
```

- Da der Compiler nicht garantieren kann, dass in der rechts stehenden ArrayList definitiv Strings drin sind, erzeugt er wiederum eine Warnung.
- Diese Problematik hat man oft v.a. bei Verwendung von Legacy-Code:

```
public static List alteMethode() {…}
```

```
List<String> ls = alteMethode();
```

- Man bekommt die Warnung auch mit einem Cast nicht weg:

```
List<String> ls = (List<String>) alteMethode();
```

- Da dies potentiell gefährlich ist, sollte man dokumentieren, dass man die Problematik zur Kenntnis genommen hat (dabei wird gleichzeitig die Warnung abgestellt):

```
@SuppressWarnings("unchecked")
 List<String> ls = alteMethode();
```

- Programmiersprachen können ganz allgemein, sollten sie über generische Typen verfügen, auf zwei Arten damit umgehen:

1. Heterogen
2. Homogen

- Heterogen: für jeden parametrisierten Typ wird individueller Code erzeugt (C++ Templates tun so etwas)
- Homogen: für jeden parametrisierte Klasse wird eine generelle Klasse erzeugt, die als Inhaltstyp an Stelle des generischen Typs lediglich `Object` besitzt und beim Compiliervorgang werden die Vorkommen der Typvariablen entsprechend umgesetzt

- Java verwendet den homogenen Ansatz ("Substitutionsprinzip")
- Das Resultat von C++-Templates und Java Generics ist recht ähnlich, aber die Realisierungen sind extrem wenig verwandt.

- Beim Compiliervorgang geschieht folgendes:

1. Der Compiler prüft, ob eine Typvariable in der Liste überall mit dem selben konkreten Typ belegt wurde.
2. Ist dies nicht der Fall, wird ein Fehler ausgelöst.
3. Ist alles korrekt belegt, wird diese Typvariable "abgehakt" und entfernt.
4. Gehe weiter zu 1. falls es noch weitere Typvariablen gibt, sonst fertig.

- Das bedeutet, dass nach Beendigung des Compiliervorgangs alles auf Korrektheit überprüft wurde, aber im Byte-Code nichts mehr von generischen Typen zu finden ist.
- Die Spezifikation nennt das Ersetzen von parametrisierten Typen durch unparametrisierte Typen "Substitution".
- Das gesamte Phänomen wird "type erasure" (Typauslöschung) genannt.

- Mit gewissen Einschränkungen aus neueren Java-Versionen kann man sagen:

  zur Laufzeit gibt es keine Generics mehr

- Jedes Objekt besitzt Meta-Information über seine Herkunft, das Class-Objekt.

```
List<String> ls = new ArrayList<>();
List<Integer> li = new ArrayList<>();

// das hier geht gar nicht - Compilefehler:
//    if (ls instanceof List<Integer>)

if (ls.getClass() == li.getClass())
{
  System.out.println("Selbe Klasse");
}
```

- Die Ausgabe lautet "Selbe Klasse", weil beide Class-Objekte identisch sind.
- Beide verweisen auf die Klasse `java.util.ArrayList`.
- Es wird mit parametrisierten Typen also kein echter neuer Typ ins Typsystem eingeführt.

- Es ist nicht immer notwendig oder sinnvoll, eine ganze Klasse zu parametrisieren.
- Methoden können unabhängig davon mit eigenen Typvariablen arbeiten.
- Die allgemeine Syntax lautet

```
modifiers <T1, T2, ...> returnType name(param1, param2, ...)
```

- Oder z.B. in der (nicht-typisierten!) Klasse Arrays:

```
public static <T> List<T> asList(T... a) {…}
```

- Aussage hier: die Methode bekommt eine Typvariable. Die Eingangsparameter müssen einen gewissen Typ besitzen und man erhält eine Liste mit demselben Typ.
- Diese Methoden müssen üblicherweise mit Hilfe der Parameterliste einen Typ "fest" machen, wenn das Resultat auch eine Typvariable enthält, ansonsten wird es im Code schwer, sich ein passendes Objekt zu besorgen.

- Ist Y von X abgeleitet, gilt bekanntermaßen folgendes (Typkompatibilität):

```
X x = new Y();
```

- Daraus folgt nicht, dass diese Kompatibilität auch für generische Typen gelten, die mit diesen Klassen parametrisiert werden, also bspw.:

```
ArrayList<X> xs = new ArrayList<Y>();
```

- Allgemein gesprochen:

`Y extends X` $\not\Rightarrow$ `G<Y> extends G<X>`

- Beweisführungen können oft kompliziert sein. Hier ist es relativ einfach: Beweis durch Gegenbeispiel.

```
Set<String> setS = new HashSet<>();
// Annahme, das funktioniert, da String ja kompatibel ist
// zu Object, dann könnten doch auch Set<String> und Set<Object>…
Set<Object> setO = setS;

// in setO ist aber dies erlaubt:
setO.add(new Integer(1));

// und hier käme die Überraschung
for (String string : setS)
{
  System.out.println(string);
}
```

- Daher ist die 2. Codezeile vom Compiler nicht erlaubt. Es gibt einen Fehler:

  `Type mismatch: cannot convert from Set<String> to Set<Object>`

- Die ungültige Implikation hinsichtlich der Typkompatibilität zeitigt überraschende Auswirkungen.
- Beispiel: wir schreiben eine Methode, die mit beliebigen Collections (wir genügen uns mit Inhaltstyp `Object`) umgehen kann.

```
public static void display(Collection<Object> c)
{
  for (Object o : c)
  {
    System.out.println(o);
  }
}
```

- Aus `Y extends X` folgt aber nicht `G<Y> extends G<X>`. Beim Aufruf mit einer beliebigen Collection, die nicht `Object` als Inhaltstyp hat, passiert eine Überraschung:

```
Collection<String> cs = new ArrayList<String>();
display(cs);
```

```
The method display(Collection<Object>) is not applicable for the
arguments (Collection<String>)
```

- Das Problem lässt sich mit Hilfe einer sog. "wildcard" lösen.

```
public static void display(Collection<?> c)
{
  for (Object o : c)
  {
    System.out.println(o);
  }
}
```

- Der Wildcard-Typ nennt sich nun "collection of unknown".
- Nun dürfen alle Collections mit beliebigen Inhaltstypen übergeben werden.
- Aber...

  ...innerhalb der Methode verliert man Typsicht. Hier ist nur der Inhaltstyp `Object` anzutreffen. Will man mehr, muss man casten bzw. mit `instanceof` erfragen.

- Um etwas konkreter als `Object` zu werden, wo nötig, kann man Wildcards ebenfalls mit Einschränkungen versehen:

```
public interface Printable
{
  void doPrint();
}
```

```
public static void display(Collection<? extends Printable> c)
{
  for (Printable o : c)
  {
    o.doPrint();
  }
}
```

- `? extends Printable` bedeutet nun, dass konkret Collections mit Inhaltstypen, die direkt oder indirekt von `Printable` abgeleitet sind, übergeben werden können.

- Die Absicherung bei Wildcards mit Einschränkungen geht in beide Richtungen.
- Wenn man eine gewisse Minimalgarantie angeben will (d.h. der Inhaltstyp darf in einer Klassenhierarchie bis zu einer gewissen "Höhe" stehen, aber nicht "höher"), dann ist das eine hierarchische Obergrenze "upper bound".

```
public static void display(Collection<? extends Printable> c)
{…}
```

- Upper Bound: Printable. Höher (z.B. Object) darf man nicht gehen.

- Wenn man eine gewisse Maximalgarantie verlangt (d.h. der Inhaltstyp darf bis zu einer gewissen "Tiefe" in der Hierarchie stehen, aber auch gerne weiter oben angesiedelt sein), dann ist das eine hierarchische Untergrenze "lower bound":

```
public static void display(Collection<? super Fahrzeug> c)
{…}
```

- Diese Methode würde nun `Fahrzeug` (und evtl. Oberklassen) sowie `Object` akzeptieren, aber nicht Pkw oder Motorrad.

- In der Java-Plattform finden sich Beispiele für beide Varianten:

```
public interface Collection<E> extends Iterable<E>
{
  boolean addAll(Collection<? extends E> c);
  …
}
```

- Bedeutet: die Collection, die mit `addAll` zusätzlich eingefügt wird, muss mindestens den Inhaltstyp `E` der eigentlichen Collection besitzen. Speziellere Objekte sind kein Problem.

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
{
  public TreeSet(Comparator<? super E> comparator) {…}
  …
}
```

- Bedeutet: der `Comparator`, der in den Konstruktor übergeben wird, darf höchstens die Eigenschaften des Typs `E` überprüfen. Für speziellere Eigenschaften möglicher Unterklassen darf er nicht zuständig sein, da die Collection nicht unbedingt diese Unterklassenobjekte auch enthält.

- `Collection<? extends Number>` erlaubt uns, `Number`-Objekte aus der Collection zu nehmen.
- Wir dürfen aber keine beliebigen Unterklassen von `Number` in die Collection reingeben. Die ursprüngliche Collection könnte eine `Collection<Integer>` sein, da darf man kein `Double` hinzugeben.
- Kurz: `? extends X` erlaubt das Lesen.

- `Collection<? super Number>` erlaubt uns, Objekte in die Collection zu schreiben, die hierarchisch geeignet ab `Number` angesiedelt sind.
- Das Lesen fällt hier deutlich schwerer, weil der genaue Inhaltstyp nicht bekannt ist.
- Kurz: `? super Y` erlaubt das Schreiben.

- Die Autoren Naftalin und Wadler haben aus diesen Beobachtungen einen Merksatz geschaffen, das sog. get-put-Prinzip ("get-put-principle")

  siehe auch
  https://doc.lagout.org/programmation/Java/
  darin
  Java Generics and Collections_ Speed Up the Java Development Process

The Get and Put Principle: use an `extends` wildcard when you only get values out of a structure, use a `super` wildcard when you only put values into a structure, and don't use a wildcard when you *both* get and put.

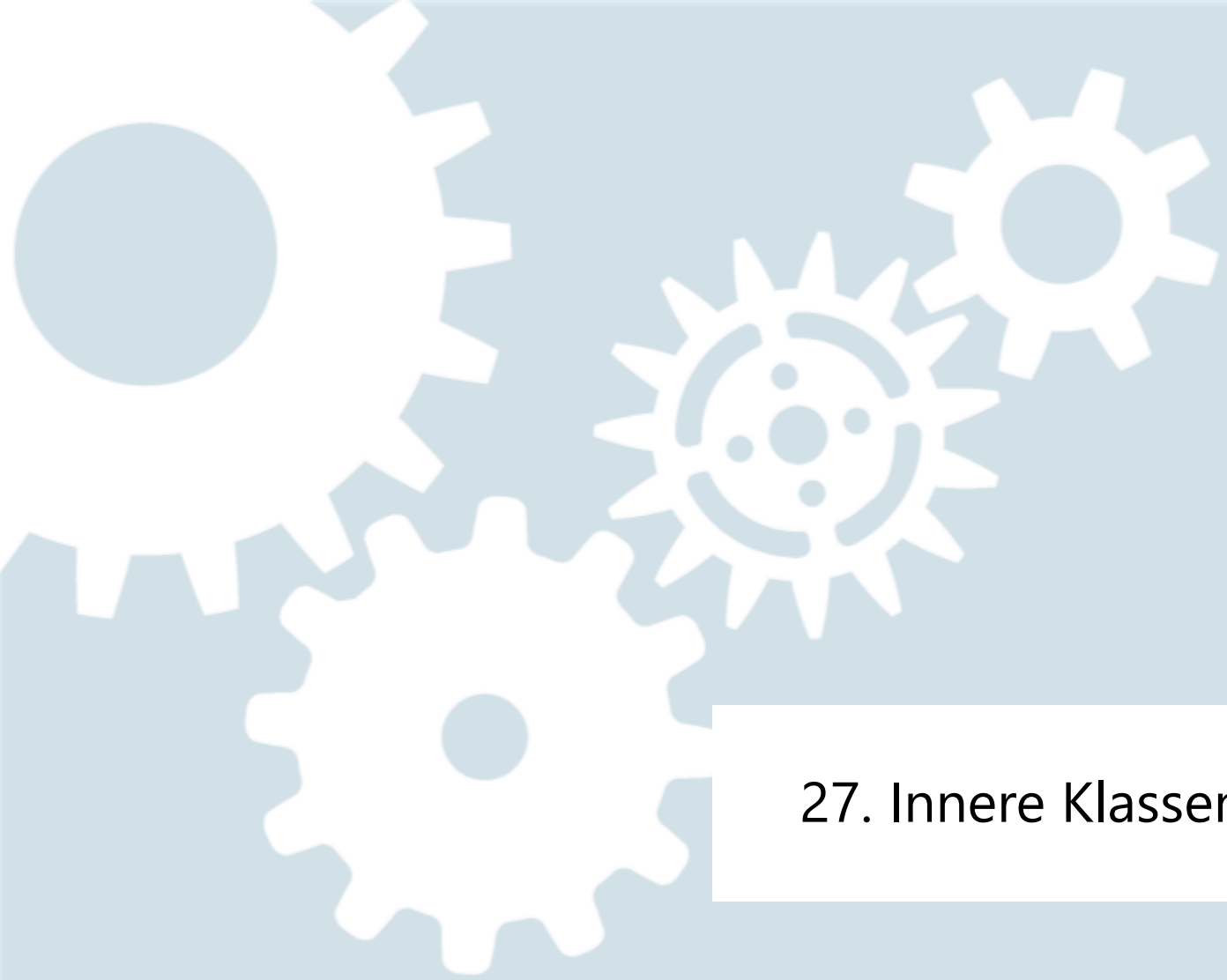
- Gutes Beispiel für eine Anwendung: die Methode `copy` in `Collections`.

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
{…}
```

- Generics sind ein wirkungsvolles und dennoch rückwärtskompatibles Instrument, um Code typsicherer, oft les- und wartbarer zu machen.
- Generics hängen nicht ausschließlich mit Collections zusammen. Eigene Klassen dürfen sie genauso benützen.
- Allerdings kann ein intensiver Einsatz von Generics Programme auch schwerer verständlich machen…

```
public static <T extends Object & Comparable<? super T>>
  T min(Collection<? extends T> coll)
```

- Generics sind hauptsächlich auf Verwendung getrimmt. Die Deklaration (s.o.) ist oft schwierig zu verstehen und zu erstellen.
- Verwenden Sie Generics hauptsächlich im Zusammenhang mit Collections, aber dafür dort durchgängig.



27. Innere Klassen

ARINKO

- Die meisten Klassen sind direkter Bestandteil ihres Paketes. Die Sprachspezifikation nennt diese Klassen "top level classes".
- Klassen, deren Deklaration sich im Rumpf eines anderen Typs befinden, nennt man "nested classes", gerne auch mal "embedded classes".
- Man findet ebenso die Bezeichnung "Innere Klassen" ("inner classes"), wobei diese Bezeichnung nicht ganz korrekt ist.

- Es gibt genau 4 Varianten von eingebetteten Klassen:

1. statisch eingebettete Klassen
2. Member-Klassen
3. lokale Klassen
4. anonyme Klassen

- Varianten 2-4 sind die eigentlichen "inneren Klassen".

- Bitte beachten: Entwickler sprechen gerne lax von inneren Klassen für alle 4 Fälle, was aber von der Sprachspezifikation so nicht gedeckt ist.

- Alle 4 Varianten werden hintereinander an einem durchgängigen Beispiel gezeigt.
- Es handelt sich um ein Konto, bei dem gewisse Personen Berechtigungen besitzen, z.B. darf eine Person auf das Konto einzahlen, eine andere darf etwas abheben, eine dritte darf das Konto auflösen.
- Die Berechtigungen sind in einem Interface zusammengezogen. Ein Objekt dieses Interfaces bezieht sich immer auf ein Konto und eine Person und stellt deren Berechtigung hinsichtlich auf das Konto dar:

```
public interface Kontoberechtigung
{
  public boolean darfEinzahlen();
  public boolean darfAbheben();
  public boolean darfAufloesen();
}
```

- Der Testcode außerhalb ändert sich ebenfalls nicht:

```
Konto konto = new Konto("0815", 0);
Person dagobert = new Person("Dagobert","Duck");
Person donald = new Person("Donald", "Duck");

konto.addAbheber(dagobert);
konto.addEinzahler(donald);
konto.addAufloeser(dagobert);

Kontoberechtigung b = konto.getBerechtigungFuer(donald);

if (b.darfAbheben())
{
  System.out.println("Juhuuuuuu!");
}
```

- Stören wir uns mal nicht am Design, dass ein Konto keinen Inhaber o.ä. hat…

- Eine statisch eingebettete Klasse wird als Klassenelement innerhalb einer anderen Klasse definiert und trägt den Modifizierer `static`.
- Statisch eingebettete Klassen haben gegenüber top-level Klassen erhöhte Sicht auf die sie umgebende Klasse:
  - sie sehen *alle* Klassenelemente (statische Attribute und Methoden) der umgebenden Klasse
  - sie sehen *alle* Attribute und Methoden von Instanzen der umgebenden Klasse, sofern sie eine Referenz auf diese Instanz besitzen
- Sie können, da sie als Klassenelemente auftauchen, alle 4 Zugriffsmodifizierer besitzen: `private`, paket-sichtbar, `protected` und `public`.

```
public class Konto
{
  private Set<Person> abheber;
  private Set<Person> einzahler;
  private Set<Person> aufloeser;
  …
  public Konto(String kontonummer, int kontostand)
  {
    …
    this.abheber   = new HashSet<>();
    this.einzahler = new HashSet<>();
    this.aufloeser = new HashSet<>();
  }

  public void addEinzahler(Person p)
  {
    this.einzahler.add(p);
  }

  public void addAbheber(Person p)
  {
    this.abheber.add(p);
  }

  … analog…
```

```
private static class Berechtigung implements Kontoberechtigung
{
  private Konto konto;
  private Person person;

  public Berechtigung(Konto k, Person p)
  {
    this.konto = k;
    this.person = p;
  }

  public boolean darfAbheben()
  {
    return this.konto.abheber.contains(this.person);
  }

  public boolean darfAufloesen()
  {
    return this.konto.aufloeser.contains(this.person);
  }

  public boolean darfEinzahlen()
  {
    return this.konto.einzahler.contains(this.person);
  }
}
```

Falls jemand gesucht hat: HIER ist der Mehrwert der eingebetteten Klasse – die Collection soll außerhalb nicht sichtbar sein

```
  public Kontoberechtigung getBerechtigungFuer(Person p)
  {
    return new Berechtigung(this, p);
  }
}
```

- Der Aufruf getBerechtigungFuer assoziiert also das Berechtigungsobjekt mit dem aktuellen Kontoobjekt (this!) und der gegebenen Person p.
- Wir erinnern uns:

```
Kontoberechtigung b = konto.getBerechtigungFuer(donald);

if (b.darfAbheben())
{
  System.out.println("Juhuuuuuu!");
}
```

- Eingebettete Klassen ohne den Modifizierer `static` sind "inner classes" (Innere Klassen).
- Innere Klassen können an verschiedenen Lokationen auftauchen.

1. Member-Klassen (als Klassenelement)
2. lokale Klassen (in einem Code-Block)
3. anonyme Klassen (in einem Code-Block)

- Alle inneren Klassen haben die Rechte einer statisch eingebetteten Klasse plus noch zusätzlichen.

- Innere Klassen kommen nur in Verbindung mit einer Instanz der äußeren Klasse vor, die man "enclosing instance" (dt. "umgebende Instanz") nennt.
- Man kann sagen, dass eine Instanz der inneren Klasse eine Referenz auf die umgebende Instanz in sich trägt.
- In der inneren Klasse kann man implizit auf die Attribute und Methoden dieser umgebenden Instanz zugreifen.
- Sollte man die umgebende Instanz syntaktisch benennen müssen/wollen, wird sie mit `NameDerUmgebendenKlasse.this` angesprochen, wohingegen nach wie vor `this` das aktuelle Objekt der inneren Klasse bezeichnet.

- Member-Klassen kommen direkt im Rumpf einer anderen Klasse vor, dürfen aber *nicht* den Modifizierer `static` besitzen.
- Sie können analog zu den statisch eingebetteten Klassen, da sie als Klassenelemente auftauchen, alle 4 Zugriffsmodifizierer besitzen: `private`, paket-sichtbar, `protected` und `public`.
- Ansonsten gilt alles für innere Klassen gesagte.

- Im Beispiel ändern wir nur die Klasse `Berechtigung` in eine Member-Klasse um. Alles andere ändert sich nicht.

```
private class Berechtigung implements Kontoberechtigung
{
  private Person person;

  public Berechtigung(Person p)
  {
    this.person = p;
  }

  public boolean darfAbheben()
  {
    return abheber.contains(this.person);
  }

  public boolean darfAufloesen()
  {
    return aufloeser.contains(this.person);
  }

  public boolean darfEinzahlen()
  {
    return einzahler.contains(this.person);
  }
}
```

Bezug auf Konto entfällt, da dies ja die umgebende Instanz ist

Impliziter Zugriff auf Elemente der umgebenden Instanz

```
  public Kontoberechtigung getBerechtigungFuer(Person p)
  {
    return new Berechtigung(p);
  }
}
```

- Der Bezug zu Konto wird nun implizit hergestellt.
- Bei der Erzeugung des inneren Objektes bindet Java sozusagen die aktuelle Instanz des äußeren Objektes daran.

- Lokale Klassen erscheinen innerhalb eines Code-Blocks, z.B. in einer Methode, einem Intialisierungsblock, in einer Schleife etc…
- Der Geltungsbereich ist der Block, in dem die Klasse deklariert ist.
- Lokale Klassen haben dieselben Möglichkeiten wir Member-Klassen, aber keine Sichtbarkeitsmodifizierer, da sie sich in einem Block befinden.
- Zusätzlich sehen sie noch `final` deklarierte Variable der umgebenden Methode (also `final` Parameter und `final` lokale Variable).

- Von final-Elementen kann die virtuelle Maschine Kopien anlegen, so dass sie der lokalen Klasse noch zur Verfügung stehen, auch wenn die Variablen bereits durch den Kontrollfluss aus ihrem Geltungsbereich und somit vom Stack eliminiert sein sollten.

- Im Beispiel verschwindet die Member-Klasse und wir verschieben die Klasse in die Methode `getBerechtigungFuer`, da diese Methode die einzige ist, die überhaupt Objekte dieser Klasse erzeugt.

```
public Kontoberechtigung getBerechtigungFuer(final Person p)
{
  class Berechtigung implements Kontoberechtigung
  {
    public boolean darfAbheben()
    {
      return abheber.contains(p);
    }

    public boolean darfAufloesen()
    {
      return aufloeser.contains(p);
    }

    public boolean darfEinzahlen()
    {
      return einzahler.contains(p);
    }
  }

  return new Berechtigung();
}
```

Kniff hier ist, die Person final zu machen. So steht sie implizit ebenfalls zur Verfügung.

Impliziter Zugriff auf Element der umgebenden Instanz und final Parameter

Als Konstruktor können wir uns jetzt den default Konstruktor gefallen lassen…

- Anonyme Klassen sind lokale Klassen, die sofort an Ort und Stelle instanziiert werden und man daher auch keinen Namen für die Klasse braucht.
- Anonyme Klassen werden immer von einer Klasse oder einem Interface, dessen Implementierung sie stellen müssen, abgeleitet.
- Sie können überall dort vorkommen, wo ein Referenz-Ausdruck stehen muss.
- Und sie kommen immer nur direkt mit dem Operator `new` vor (was an obigen Eigenschaften liegt).

- Anonyme Klassen können für prototypische Ansätze und Tests recht nützlich sein.
- Sie werden bemerken, wenn Sie zu viel davon einsetzen, dass der Code aber wenig nachvollziehbar wird.
- Empfehlung: benutzen Sie dieses Sprachmittel sehr dosiert in Ihren Projekten!

- Das Beispiel zeigt wieder die Methode `getBerechtigungFuer`, erspart sich aber, einen Klassennamen für die nur lokal vorhandene und sofort instanziierte Klasse zu deklarieren…

```
public Kontoberechtigung getBerechtigungFuer(final Person p)
{
    return new Kontoberechtigung()
    {
      public boolean darfAbheben()
      {
        return abheber.contains(p);
      }

      public boolean darfAufloesen()
      {
        return aufloeser.contains(p);
      }

      public boolean darfEinzahlen()
      {
        return einzahler.contains(p);
      }
    };
}
```

Sofort wird eine Klasse, die das Interface ableitet, instanziiert

Hier müssen natürlich alle Methoden aus dem Interface implementiert werden


Praxis
Theorie