

Lektion 21

Builder Pattern

Method Chaining

Fluent Interface

Geschachtelte Klassen (statisch geschachtelte, innere)

Builder Pattern

Wir wollen eine Klasse Person mit Pflichtattributen und optionalen Attributen entwickeln.

```
public class Person
{
    String vorname;
    String nachname;
    String geburtsdatum;
    String strasse;
    int hausnummer;
    int plz;
    String wohnort;
    int koerpergroesse;
    int gewicht;
}
//getter und setter
```

The code is annotated with two curly braces on the right side. The first brace groups the first five attributes (vorname, nachname, geburtsdatum, strasse, hausnummer) under the label 'verpflichtend' (mandatory). The second brace groups the last four attributes (plz, wohnort, koerpergroesse, gewicht) under the label 'optional'.

Welche Konstruktoren stellen wir zur Verfügung?

```
public class Person
{
    String vorname;
    String nachname;
    String geburtsdatum;
    String strasse;
    int hausnummer;
    int plz;
    String wohnort;
    int koerpergroesse;
    int gewicht;

    //getter und setter
```

```
    public Person(String vorname, String nachname, String geburtsdatum)
    {
        this.vorname = vorname;
        this.nachname = nachname;
        this.geburtsdatum = geburtsdatum;
    }

    public Person(String vorname, String nachname, String geburtsdatum,
                  String strasse, int hausnummer, int plz,
                  String wohnort, int koerpergroesse, int gewicht)
    {
        this.vorname = vorname;
        this.nachname = nachname;
        this.geburtsdatum = geburtsdatum;
        this.strasse = strasse;
        this.hausnummer = hausnummer;
        this.plz = plz;
        this_wohnort = wohnort;
        this.koerpergroesse = koerpergroesse;
        this.gewicht = gewicht;
    }
}
```

Beim langen Konstruktor ist der Aufruf fehleranfällig.

```
Person p = new Person("John", "Doe", null, "01.01.1900", -1, -1, "Berlin", 188, -1);
```

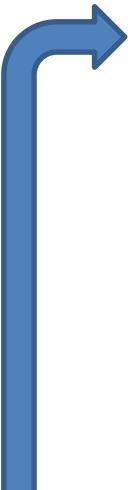
Person wird

- mit Geburtsdatum **null** angelegt
- mit Strasse **01.01.1900**

```
Person p = new Person("John", "Doe", "01.01.1900", null, -1, -1, "Berlin", 188, -1);
```

(In jedem Fall fehlen Validierungen, die wir aus Anschaulichkeitsgründen weglassen)

Beim kurzen Konstruktor (mit den Pflichtfeldern), müssen die restlichen Attribute über setter gesetzt werden.

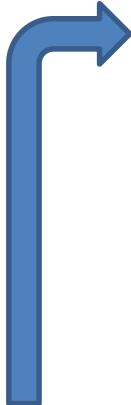


```
Person person = new Person("John", "Doe", "01.01.1900");
...
person.setKoerpergroesse(188);
person.setWohnort("Berlin");
```

Zwischendurch ist das Objekt möglicherweise in einem **inkonsistenten** Zustand.

Beim kurzen Konstruktor (mit den Pflichtfeldern), müssen die restlichen Attribute über setter gesetzt werden.

```
Person person = new Person("John", "Doe", "01.01.1900");
...
person.setKoerpergroesse(188);
person.setWohnort("Berlin");
```



In jeder Zeile muss die gleiche Referenz erneut getippt werden.

Um anzuzeigen, dass optionale Attribute bei der Objekterzeugung gesetzt werden sollen, können die Aufrufe per **Method Chaining** aneinander gereiht werden.

```
public class Person
{
    ...
    public Person withKoerpergroesse(int koerpergroesse)
    {
        this.koerpergroesse = koerpergroesse;
        return this;
    }

    public Person withWohnort(String wohnort)
    {
        this_wohnort = wohnort;
        return this;
    }
    ...
}
```

```
Person p = new Person("John", "Doe", "01.01.1900")
        .withKoerpergroesse(188)
        .withWohnort("Berlin");
```

Man könnte jetzt immer noch das Objekt erzeugen und mit den `with`-Methoden später die Attribute setzen:

```
Person p = new Person("John", "Doe", "01.01.1900");
...
p.withKoerpergroesse(188);
p.withWohnort("Berlin");
```

Bei der Objekterstellung kann man den Entwickler durch das **Builder-Pattern** zwingen, dass er erst alle gewollten optionalen Attribute setzt, bevor er mit dem Objekt arbeitet.

```
public class PersonBuilder {  
    ...  
    String wohnort;  
    int koerpergroesse;  
  
    public PersonBuilder(String vorname, String nachname, String geburtsdatum) {  
        this.vorname = vorname;  
        this.nachname = nachname;  
        this.geburtsdatum = geburtsdatum;  
    }  
    ...  
    public PersonBuilder withWohnort(String wohnort) {  
        this_wohnort = wohnort;  
        return this;  
    }  
    public PersonBuilder withKoerpergroesse(int koerpergroesse) {  
        this_koerpergroesse = koerpergroesse;  
        return this;  
    }  
    public Person build() {  
        return new Person(vorname, nachname, geburtsdatum, strasse, hausnummer, plz, wohnort,  
            koerpergroesse, gewicht);  
    }  
}
```

```
Person p = new PersonBuilder("John", "Doe", "01.01.1900")  
    .withKoerpergroesse(188)  
    .withWohnort("Berlin")  
    .build();
```

Ein Person-Objekt kann theoretisch vom Entwickler über den Person-Konstruktor angelegt werden.

Um das zu unterbinden, kann man den Konstruktor *private* machen und die Klassen ineinander schachteln:

```

public class Person
{
    String vorname;
    String nachname;
    ...
    String wohnort;
    int koerpergroesse;

    private Person(String vorname,
        String nachname, String geburtsdatum,
        String strasse, int hausnummer, int plz,
        String wohnort, int koerpergroesse,
        int gewicht) {
        this.vorname = vorname;
        this.nachname = nachname;
        ...
        this.wohnort = wohnort;
        this.koerpergroesse = koerpergroesse;
        this.gewicht = gewicht;
    }
    //getter
    ...
    public static class PersonBuilder
    {
        String vorname;
        String nachname;
        ...
        String wohnort;
        int koerpergroesse;
    }
}

```

```

public PersonBuilder(String vorname, String nachname,
    String geburtsdatum) {
    this.vorname = vorname;
    this.nachname = nachname;
    this.geburtsdatum = geburtsdatum;
}
public PersonBuilder withWohnort(String wohnort) {
    this.wohnort = wohnort;
    return this;
}
public PersonBuilder withKoerpergroesse(int koerpergroesse) {
    this.koerpergroesse = koerpergroesse;
    return this;
}
public Person build() {
    return new Person(vorname, nachname, geburtsdatum,
        strasse, hausnummer, plz, wohnort, koerpergroesse, gewicht);
}
} //end of class PersonBuilder
} //end of class Person

```

```

Person p = new Person.PersonBuilder("John", "Doe", "01.01.1900")
    .withKoerpergroesse(188)
    .withWohnort("Berlin")
    .build();

```

Builder Pattern

```
Person p = new Person.PersonBuilder("John", "Doe", "01.01.1900")
    .koerpergroesse(188)
    .wohnort("Berlin")
    .build();
```

Der Entwickler, der unsere Klasse nutzt, erzeugt nicht direkt ein Objekt der Klasse, sondern ein Builder-Objekt mit den verpflichtenden Parametern.

Dann ruft er **setter-ähnliche** Methoden auf, um die optionalen Parameter zu setzen.

Am Ende erzeugt er das Objekt mit Hilfe einer `build()`-Methode.
Das Objekt ist i.d.R. unveränderlich (`immutable`).

Welche Techniken haben wir beim
Builder Pattern verwendet?

Method Chaining
Fluent Interface
Geschachtelte Klasse

Method Chaining

Woher kommt der Name Method Chaining?

```
RunInstancesRequest runInstancesRequest = new RunInstancesRequest().withInstanceTy
```

Method Chaining ist die Technik Methodenaufrufe aneinanderzureihen, ohne die Rückgabewerte jedes Zwischenschritts in einer Variablen zwischen zu speichern.

Method Chaining - Beispiel

- Um bei Amazon Web Services eine virtuelle Maschine hochzufahren:

```
RunInstancesRequest runInstancesRequest = new RunInstancesRequest()  
    .withInstanceType("t1.micro")  
    .withImageId("ami-d2929fa6")  
    .withMinCount(1)  
    .withMaxCount(1)  
    .withSecurityGroupIds("quick-start-1")  
    .withKeyName("MyKey")  
;  
RunInstancesResult runInstancesResult = ec2.runInstances(runInstancesRequest);
```

Es gibt ca. 30 Attribute, aus denen man wählen kann.

schickt den Wunsch, eine virtuelle Maschine mit obigen Eigenschaften hochzufahren, über das Netzwerk an Amazon.

Fluent Interface

Ein Fluent Interface...

- ist ein objektorientiertes API (d.h. eine Sammlung von Klassen/Interfaces, die Entwicklern zur Nutzung zur Verfügung gestellt werden, um eine bestimmte Funktionalität zu verwenden), das so designed wurde, dass es bei jedem Methodenaufruf immer einen Rückgabetypen zurückgibt, der festlegt, welches die sinnvollen nächstmöglichen Methodenaufrufe sind.
- wird spätestens durch den Rückgabetypen void terminiert.
- ermöglicht es ggf. mit beinahe natürlicher Sprache zu programmieren

Der Rückgabetyp bestimmt die nächsten Methodenaufrufe des Fluent Interface.

Mit Method Chaining sieht man den Rückgabetyp im Code nicht.

Ohne Method Chaining:

```
Client newClient = ClientBuilder.newClient();
WebTarget target = newClient.target("https://en.wikipedia.org/");
target = target.path("wiki/Fluent_interface");
Builder builder = target.request(MediaType.TEXT_HTML);
Response response = builder.get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Mit Method Chaining:

```
Response response = ClientBuilder.newClient()
    .target("https://en.wikipedia.org/")
    .path("wiki/Fluent_interface")
    .request(MediaType.TEXT_HTML)
    .get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Ohne Method Chaining:

```
Client newClient = ClientBuilder.newClient();
WebTarget target = newClient.target("https://en.wikipedia.org/");
target = target.path("wiki/Fluent_interface");
Builder builder = target.request(MediaType.TEXT_HTML);
Response response = builder.get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Häufig heißen die
Methodennamen gleich wie
die Attribute, die gesetzt
werden.

```
public interface WebTarget
    extends Configurable<WebTarget>
{
    ...
    public WebTarget path(String path);
}
```

Mit Method Chaining:

```
Response response = ClientBuilder.newClient()
    .target("https://en.wikipedia.org/")
    .path("wiki/Fluent_interface")
    .request(MediaType.TEXT_HTML)
    .get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Eine Referenz auf das gleiche Objekt wird zurückgegeben.

Ohne Method Chaining:

```
Client newClient = ClientBuilder.newClient();
WebTarget target = newClient.target("https://en.wikipedia.org/");
target = target.path("wiki/Fluent_interface");
Builder builder = target.request(MediaType.TEXT_HTML);
Response response = builder.get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Mit Method Chaining:

```
Response response = ClientBuilder.newClient()
    .target("https://en.wikipedia.org/")
    .path("wiki/Fluent_interface")
    .request(MediaType.TEXT_HTML)
    .get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Ohne Method Chaining:

```
Client newClient = ClientBuilder.newClient();
WebTarget target = newClient.target("https://en.wikipedia.org/");
target = target.path("wiki/Fluent_interface");
Builder builder = target.request(MediaType.TEXT_HTML);
Response response = builder.get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Die Terminierung erfolgt hier **nicht** über den Rückgabewert void.

Eine sinnvolle Grenze des Fluent Interface ist hier die `get()`-Methode, die einen Netzwerkrequest startet, da hier Fehler auftreten können.

Mit Method Chaining:

```
Response response = ClientBuilder.newClient()
    .target("https://en.wikipedia.org/")
    .path("wiki/Fluent_interface")
    .request(MediaType.TEXT_HTML)
    .get();
String s = response.readEntity(String.class);
System.out.println(s);
```

Geschachtelte Klassen

Hundert-Türen-Problem

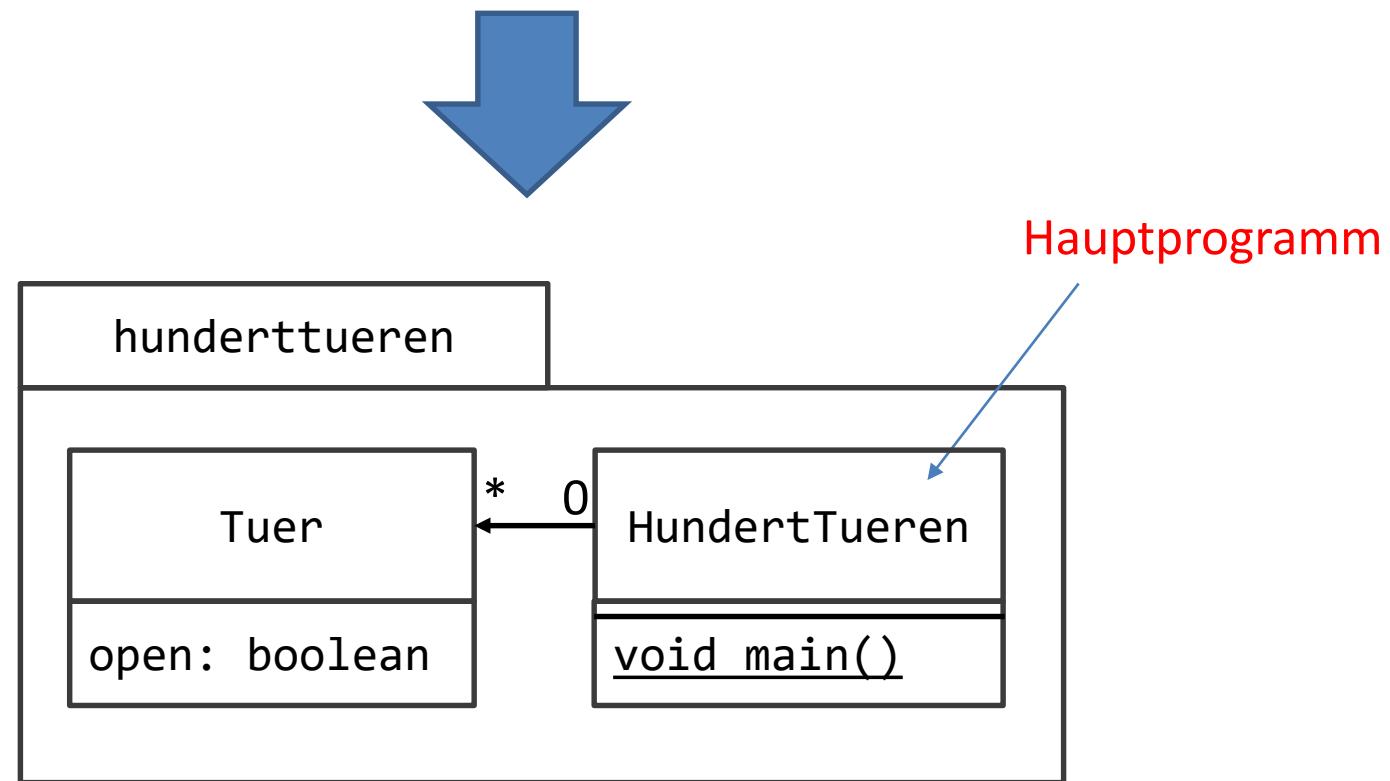
Wir stehen vor 100 nebeneinander angeordneten Schließfächern, die sämtlich geschlossen sind.

Ein Mann hat einen Schlüsselbund mit allen 100 Schlüsseln und wird genau hundertmal an den Schließfächern vorbeigehen und dabei manche öffnen oder schließen.

- Beim ersten Durchgang öffnet er alle Schließfachtüren.
- Beim zweiten Durchgang geht der Mann zu jedem zweiten Fach und wechselt dessen Zustand.
 - Das heißt: Ist es geschlossen, wird es geöffnet.
 - Ist es bereits offen, wird es geschlossen.
- Im konkreten Fall schließt er die Fächer 2, 4, 6, ... 98 und 100, weil vorher alle Türen offen standen.
- Beim dritten Durchgang ändert er den Zustand jedes dritten Faches - also 3, 6, 9, ... 96, 99.
- Beim vierten Durchgang geht es um jedes vierte Fach, beim fünften um jedes fünfte - und so weiter.
- Beim letzten, dem 100. Durchgang ändert der Mann schließlich nur den Zustand der Tür Nummer 100.

Die Frage lautet: Wie viele der 100 Fächer stehen nach dem 100. Durchgang offen?

Normale Vorgehensweise:
Es soll 100 Schließfachtüren geben, die entweder offen oder geschlossen sind.



```
package hunderttueren;

public class Tuer
{
    boolean open = false;

    @Override
    public String toString()
    {
        return open ? "x" : "o";
    }
}
```

```
package hunderttueren;

public class HundertTueren
{
    Tuer[] tueren = new Tuer[100];

    public HundertTueren()
    {
        for (int i = 0; i < tueren.length; i++)
            tueren[i] = new Tuer();
    }

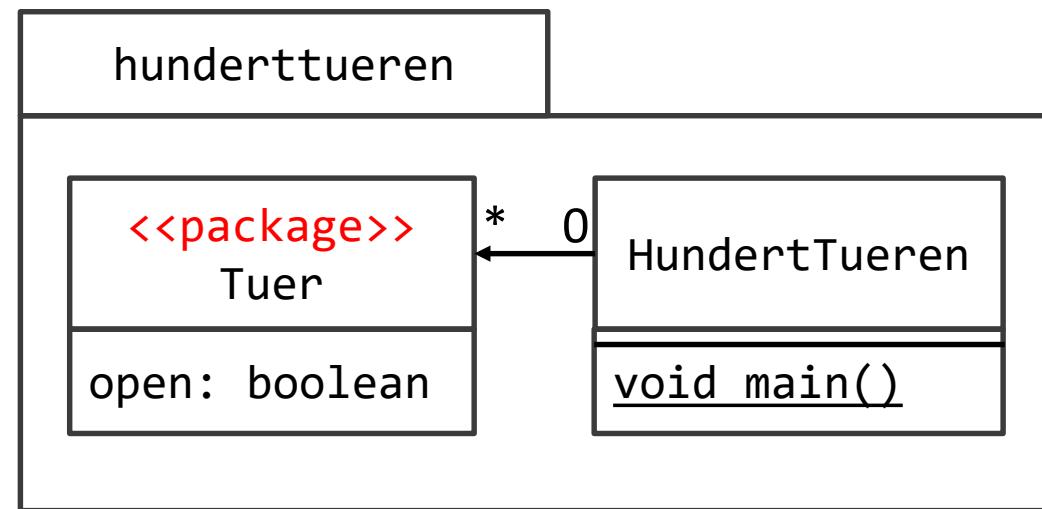
    public void starteDurchgaenge()
    {
        for (int i = 1; i <= 100; i++)
        {
            for (int j = i - 1; j < tueren.length; j = j + i)
                tueren[j].open = !tueren[j].open;
        }
    }

    public int zaehleOffeneTueren()
    {
        int offen = 0;
        for (Tuer t : tueren)
            if (t.open) offen++;
        return offen;
    }
}

//main...
```

Nehmen wir an, wir wollen den Zugriff auf die Klasse Tuer für andere verbieten, da die Tuer nur im Kontext von HundertTueren verwendet werden soll.

Modifikation von
public auf package
Sichtbarkeit



```
package hunderttueren;

public class Tuer
{
    boolean open = false;

    @Override
    public String toString()
    {
        return open ? "x" : "o";
    }
}
```

```
package hunderttueren;

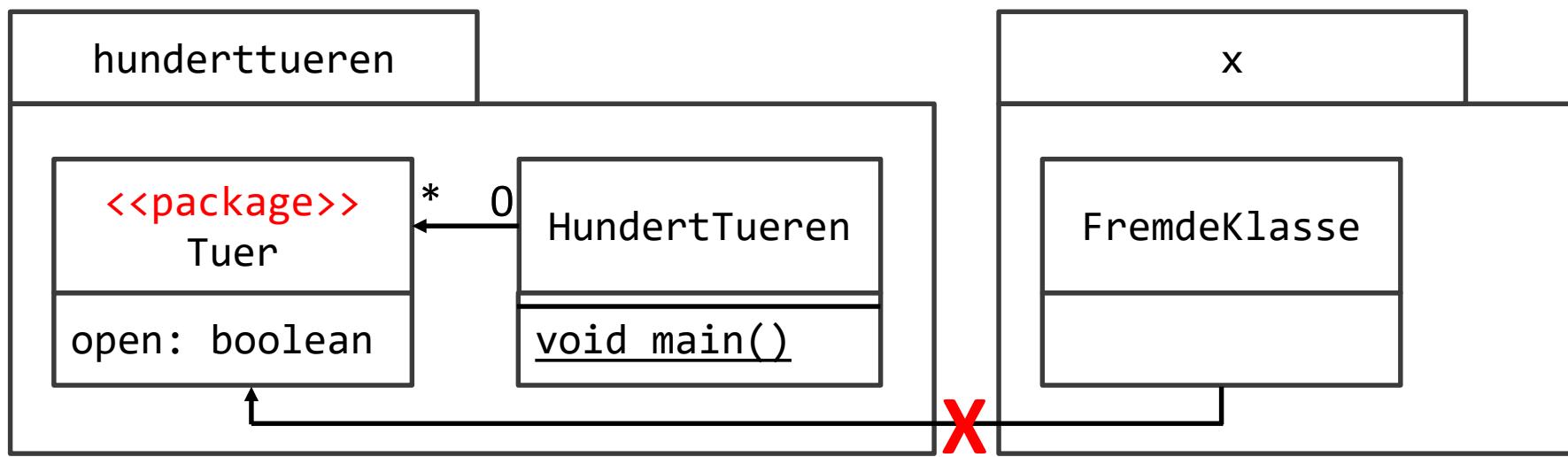
public class HundertTueren
{
    Tuer[] tueren = new Tuer[100];

    public HundertTueren()
    {
        for (int i = 0; i < tueren.length; i++)
            tueren[i] = new Tuer();
    }

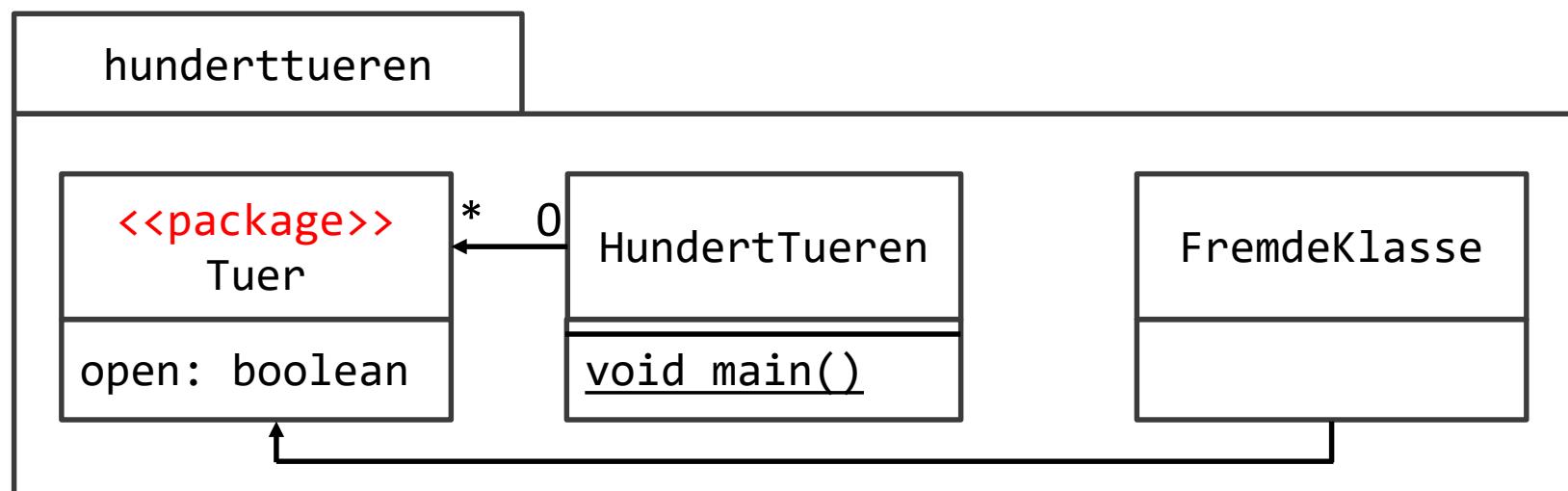
    public void starteDurchgaenge()
    {
        for (int i = 1; i <= 100; i++)
        {
            for (int j = i - 1; j < tueren.length; j = j + i)
                tueren[j].open = !tueren[j].open;
        }
    }

    public int zaehleOffeneTueren()
    {
        int offen = 0;
        for (Tuer t : tueren)
            if (t.open) offen++;
        return offen;
    }

    //main...
}
```



funktioniert nicht, da die Klasse Tuer nicht sichtbar ist.



funktioniert, wenn der Programmierer der fremden Klasse einfach das gleiche Package verwendet

Wir müssten die Klasse private machen.
Das ist aber wenig sinnvoll, weil die Klasse niemand benutzen könnte. (Daher ist das nicht erlaubt.)

```
package hunderttueren;  
  
private class Tuer {  
    boolean open = false;  
  
    @Override  
    public String toString()  
    {  
        return open ? "x" : "o";  
    }  
}
```



```
package hunderttueren;  
  
public class HundertTueren {  
    Tuer[] tueren = new Tuer[100];  
  
    public HundertTueren() {  
        for (int i = 0; i < tueren.length; i++)  
            tueren[i] = new Tuer();  
    }  
  
    public void starteDurchgaenge() {  
        for (int i = 1; i <= 100; i++) {  
            for (int j = i - 1; j < tueren.length; j = j + i)  
                tueren[j].open = !tueren[j].open;  
        }  
    }  
  
    public int zaehleOffeneTueren() {  
        int offen = 0;  
        for (Tuer t : tueren)
```

würde nicht funktionieren

Wir können die Klasse Tuer in die Klasse HundertTueren schieben,
als eine sogenannte **geschachtelte Klasse**.

```
package hunderttueren;  
  
class Tuer  
{  
    boolean open = false;  
  
    @Override  
    public String toString()  
    {  
        return open ? "x" : "o";  
    }  
}
```

```
package hunderttueren;  
  
public class HundertTueren  
{  
    Tuer[] tueren = new Tuer[100];  
  
    public HundertTueren() {  
        for (int i = 0; i < tueren.length; i++)  
            tueren[i] = new Tuer();  
    }  
  
    public void starteDurchgaenge() {  
        for (int i = 1; i <= 100; i++) {  
            for (int j = i - 1; j < tueren.length; j = j + i)  
                tueren[j].open = !tueren[j].open;  
        }  
    }  
  
    public int zaehleOffeneTueren() {  
        int offen = 0;  
        for (Tuer t : tueren)
```

Diese Art von geschachtelter Klasse wird auch als
innere Klasse bezeichnet.

Die geschachtelte Klasse
können wir **private** machen.

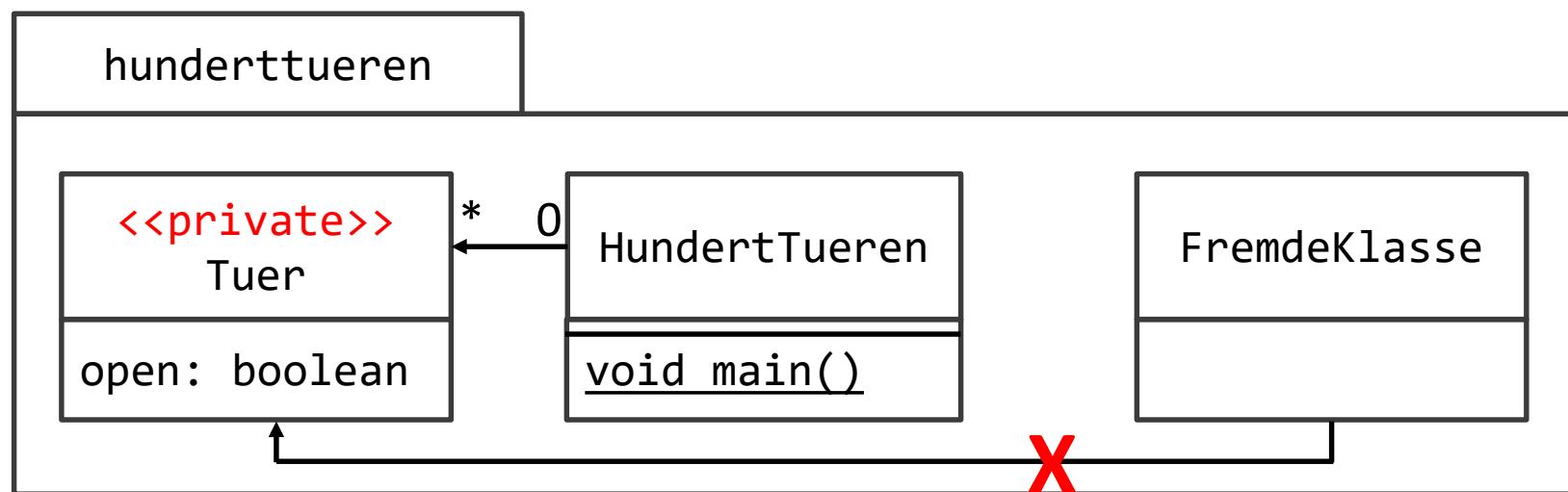
```
public class HundertTueren
{
    private class Tuer
    {
        boolean open = false;

        @Override
        public String toString()
        {
            return open ? "x" : "o";
        }
    }

    Tuer[] tueren = new Tuer[100];

    public HundertTueren()
    {
        for (int i = 0; i < tueren.length; i++)
            tueren[i] = new Tuer();
    }

    public void starteDurchgaenge()
    {
```



Der Zugriff der fremden Klasse ist nicht mehr möglich.

Wann verwendet man geschachtelte Klassen noch?

Als Helfer- oder Adapter-Klassen

Eine Helfer-Klasse könnte eine Klasse Operation innerhalb einer Taschenrechner-Klasse sein, die die Operationen eines Taschenrechners beschreibt.

Eine Adapter-Klasse lässt eine Instanz einer Klasse wie eine Instanz einer anderen Klasse aussehen.

Eine Adapter-Klasse lässt eine Instanz einer Klasse wie eine Instanz einer anderen Klasse aussehen.

Wo wird sowas gebraucht?

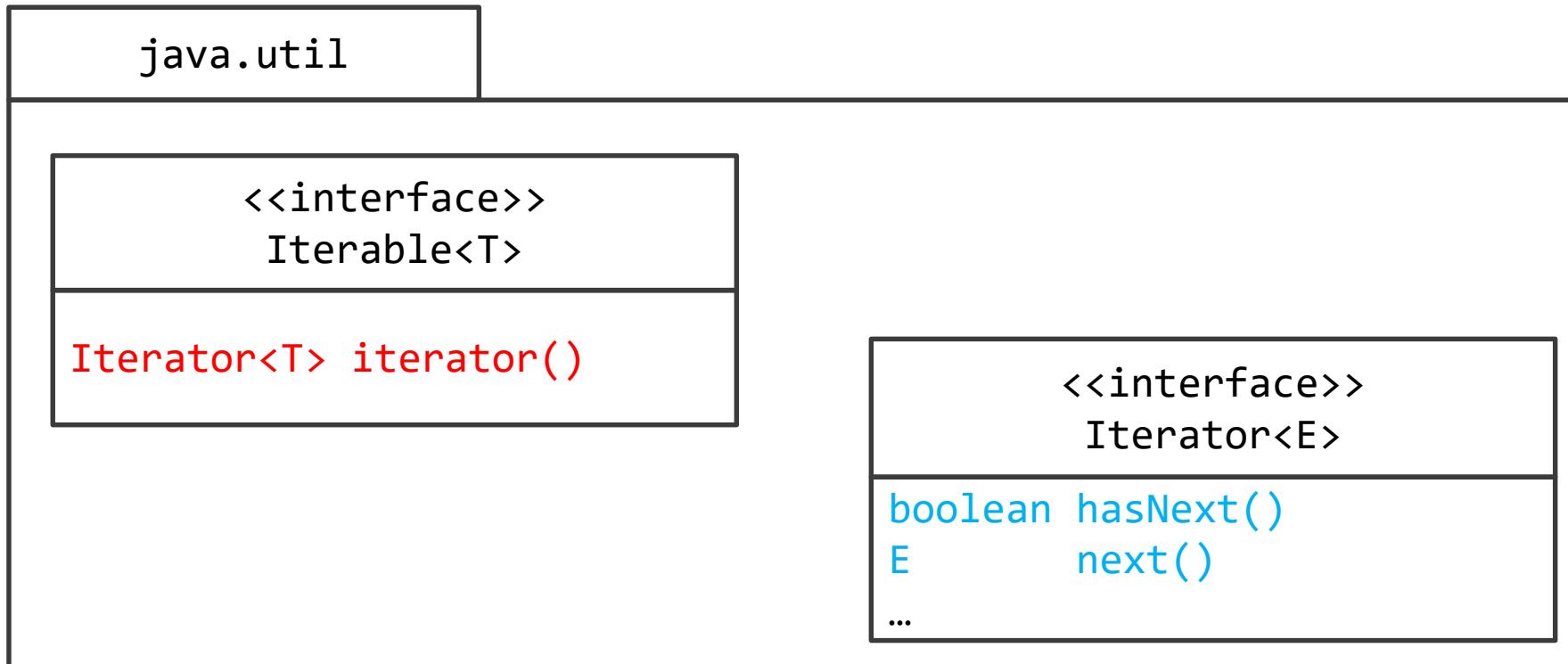
Ein typisches Beispiel sind Iteratoren.

Einfaches Beispiel:
Wir wollen über die Characters eines Strings iterieren.

```
String s = "Hallo Welt";
for (char c : s)
{
    System.out.print(c);
}
```

Nicht möglich, da String das Interface Iterable nicht implementiert.

Zur Erinnerung:



```
public class IterableString
    implements Iterable<Character>
{
    private String string;

    public IterableString(String string)
    {
        this.string = string;
    }

    @Override
    public Iterator<Character> iterator()
    {
        return new StringIterator(string);
    }
}

public class Main {
    public static void main(String[] args) {
        IterableString s
            = new IterableString("Hallo Welt");
        for (char c : s)
            System.out.print(c);
    }
}
```

```
class StringIterator implements Iterator<Character>
{
    private String string;

    public StringIterator(String string)
    {
        this.string = string;
    }

    private int currentPosition = 0;

    @Override
    public boolean hasNext()
    {
        if (currentPosition < string.length()) return true;
        else return false;
    }

    @Override
    public Character next()
    {
    }
}
```

```
public class IterableString
    implements Iterable<Character>
{
    private String string;

    public IterableString(String string)
    {
        this.string = string;
    }

    @Override
    public Iterator<Character> iterator()
    {
        return new StringIterator(string);
    }
}

public class Main {
    public static void main(String[] args) {
        IterableString s
            = new IterableString("Hallo Welt");
        for (char c : s)
            System.out.print(c);
    }
}
```

```
class StringIterator implements Iterator<Character>
{
    private String string;

    public StringIterator(String string)
    {
        this.string = string;
    }

    private int currentPosition = 0;

    @Override
    public boolean hasNext()
    {
        return currentPosition < string.length();
    }

    @Override
    public Character next()
    {
        char c = string.charAt(currentPosition);
        currentPosition++;
        return c;
    }
}
```

Beide Klassen arbeiten auf dem gleichen String.

Vlt ist es übersichtlicher, wenn eine Klasse in die andere
geschachtelt wird.

Der Konstruktor und die Variabledeklaration von string können in der geschachtelten Klasse wegfallen.

```
public class IterableString implements Iterable<Character> {  
    private String string;  
  
    public IterableString(String string) {  
        this.string = string;  
    }  
  
    @Override  
    public Iterator<Character> iterator() {  
        return new StringIterator(string);  
    }  
  
    class StringIterator implements Iterator<Character> {  
        private String string;  
  
        public StringIterator(String string) {  
            this.string = string;  
        }  
  
        private int currentPosition = 0;  
  
        @Override  
        public boolean hasNext() {  
            return currentPosition < string.length();  
        }  
  
        @Override  
        public Character next() {
```

Die innere Klasse hat Zugriff auf die
(privaten) Variablen der äußeren Klasse.

```
public class IterableString implements Iterable<Character> {
    private String string;

    public IterableString(String string) {
        this.string = string;
    }

    @Override
    public Iterator<Character> iterator() {
        return new StringIterator();
    }

    class StringIterator implements Iterator<Character> {
        private int currentPosition = 0;

        @Override
        public boolean hasNext() {
            return currentPosition < string.length();
        }

        @Override
        public Character next() {
            char c = string.charAt(currentPosition);
            currentPosition++;
            return c;
        }
    }
}
```

```
public class IterableString implements Iterable<Character> {
    private String string;

    public IterableString(String string) {
        this.string = string;
    }

    @Override
    public Iterator<Character> iterator() {
        return new StringIterator();
    }

    class StringIterator implements Iterator<Character> {
        private int currentPosition = 0;

        @Override
        public boolean hasNext() {
            return currentPosition < string.length();
        }

        @Override
        public Character next() {
            char c = string.charAt(currentPosition);
            currentPosition++;
            return c;
        }
    }
}
```

Die Main-Klasse bleibt unverändert.

```
public class Main {
    public static void main(String[] args) {
        IterableString s
            = new IterableString("Hallo Welt");
        for (char c : s)
            System.out.print(c);
    }
}
```

Im übrigen lässt sich unser einfaches Beispiel viel einfacher realisieren:

```
String s = "Hallo Welt";
for (char c : s.toCharArray())
{
    System.out.print(c);
}
```

Das vorherige Beispiel war nur zur Illustration einer Adapter-Klasse.

Im ersten Beispiel hat die innere Klasse nicht auf den Zustand der äußeren Klasse zugegriffen.

Wenn ein solcher Zugriff nicht erfolgt, sollten wir die Klasse **static** machen.

innere Klasse:

```
public class HundertTueren
{
    private class Tuer
    {
        boolean open = false;

        @Override
        public String toString()
        {
            return open ? "x" : "o";
        }
    }

    Tuer[] tueren = new Tuer[100];

    public HundertTueren()
    {
        for (int i = 0; i < tueren.length; i++)
            tueren[i] = new Tuer();
    }
}
```

statische geschachtelte Klasse:

```
public class HundertTueren
{
    private static class Tuer
    {
        boolean open = false;

        @Override
        public String toString()
        {
            return open ? "x" : "o";
        }
    }

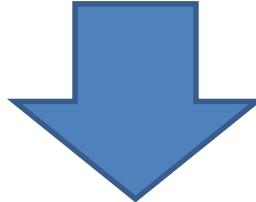
    Tuer[] tueren = new Tuer[100];

    public HundertTueren()
    {
        for (int i = 0; i < tueren.length; i++)
            tueren[i] = new Tuer();
    }
}
```

innere Klasse:

Ein Objekt der inneren Klasse hält immer eine Referenz auf das Objekt der äußeren Klasse.

```
class Outer {  
    class Inner { }  
}
```



Der Compiler ergänzt für die innere Klasse sinngemäß **folgenden** Code:

```
class Inner {  
    private final Outer myOuter;  
    Inner(Outer myOuter) {  
        this.myOuter = myOuter;  
    }  
}
```

Eine Instanz der inneren Klasse ist auf die Existenz einer Instanz der äußeren Klasse angewiesen.

statische geschachtelte Klasse:

Eine statisch geschachtelte Klasse kann unabhängig von der äußeren Klasse genutzt werden.

```
class Outer {  
    static class Inner { }  
}
```

Eine Instanz einer statischen geschachtelten Klasse ist nicht auf die Existenz einer Instanz der äußeren Klasse angewiesen.

innere Klasse:

```
public class HundertTueren
{
    class Tuer
    {
        ...
    }
    ...
}

public class Main
{
    public static void main(String[] args)
    {
        HundertTueren h = new HundertTueren();
        HundertTueren.Tuer t = h.new Tuer();
    }
}
```

`new` kann nur in Abhängigkeit eines Objekts der äußeren Klasse aufgerufen werden, da die innere Klasse die Referenz zur äußeren Klasse speichern muss.

statische geschachtelte Klasse:

```
public class HundertTueren
{
    static class Tuer
    {
        ...
    }
    ...
}

public class Main
{
    public static void main(String[] args)
    {
        HundertTueren.Tuer t
            = new HundertTueren.Tuer();
    }
}
```

Eine statische geschachtelte Klasse kann auch ohne ein Objekt der äußeren Klasse instantiiert werden.

Es gibt verschiedene Kategorien von
Geschachtelten Klassen
(Nested Classes):

- static nested classes**
- inner classes**
- local classes
- anonymous classes
- lambda expressions