Innere Klassen

Es ist möglich, Klassen in Klassen zu programmieren! Wenn du das bisher gemacht hast, war es wahrscheinlich ein Fehler, weil die geschweiften Klammern falsch gesetzt waren. Es gibt aber Fälle, wo es gewollt ist, dass eine Klasse in eine andere geschachtelt ist.

Beispielsweise dann, wenn wir einen Datentypen erstellen möchten, der nur für eine einzige Klasse existieren soll. Vielleicht möchten wir das, weil wir nicht möchten, dass von außen sichtbar ist, wie der Datentyp aussieht. Beispiel:

```
class Buchverwaltung{
	private class Buch{
		private String autor;
		private String titel;

		public Buch(String autor, String titel){
			this.autor = autor;
			this.titel = titel;
		}
	}
}
```

Die Klasse Buch kann nur innerhalb von Buchverwaltung verwendet werden. Dabei speichert die innere Klasse immer eine Referenz auf die äußere Klasse. Es wird hierfür automatisch Code ergänzt (nicht so wichtig).

Möglicherweise interessiert sich die innere Klasse überhaupt nicht dafür, welchen Zustand die äußere Klasse hat (Instanzattribute). Dann können wir sie auch static machen:

```
class Buchverwaltung{
	private static class Buch{
		private String autor;
		private String titel;

		public Buch(String autor, String titel){
			this.autor = autor;
			this.titel = titel;
		}
	}
}
```

Dann ist es möglich, eine Instanz der inneren Klasse zu erzeugen, ohne dass es eine Instanz der äußeren Klasse gibt.

```
// Innerhalb von Buchverwaltung
public static String gibNeuesBuchAus(){
	Buch b = new Buch("Holzmann", "Guter Titel");

	// System.out …
}
```

Außerhalb der Klassen

Dann sieht das ganze ein bisschen komisch aus, aber daran muss man sich gewöhnen:

```
class Holz{
	public class Block{
	}
}

class Test{
	public static void main(String[] args){
		Holz h = new Holz();
		Holz.Block hb = h.new Block();
	}
}
```

Und wenn die innere Klasse **static** ist? Dann sieht das leider auch nicht viel besser aus. Der Unterschied ist, dass wir keine Instanz von Holz brauchen.

```
class Holz{
	public static class Block{
	}
}

public class Test{
	public static void main(String[] args){
		Holz.Block b = new Holz.Block();
	}
}
```

Anonyme innere Klassen

Es ist möglich bei Instanziierung einer Klasse einen neuen Datentyp direkt anzugeben. Es wird dabei nicht angegeben, wie dieser Datentyp heißt (deswegen anonym).
Zunächst ein Beispiel, wie wir es kennen:

```
public class Test{
	public static void main(String[] args){
		// Instanziierung
		Student s = new Student();
		InformatikStudent is = new InformatikStudent();

		System.out.println(is.getMatrikelnummer());
	}
}

class Student{
	private int matrikelnummer = 1001;
	public int getMatrikelnummer(){
		return matrikelnummer;
	}
}

class InformatikStudent extends Student{
	public int getMatrikelnummer(){
		return super.getMatrikelnummer() + 90000;
	}
}
```

Und hier mit anonymer innerer Klasse:

```
public class Test{
	public static void main(String[] args){
		// in folgender Zeile wird ein neuer Datentyp erzeugt, der
		// aber nur für is verfügbar ist
		Student is = new Student(){
		    // das ist das neue daran, weil wir jetzt Methoden
		    programmieren können
		    // siehe Gemeinsamkeiten mit dem vorherigen Beispiel
		    public int getMatrikelnummer(){
				return super.getMatrikelnummer() + 90000;
		    }
		};

		System.out.println(is.getMatrikelnummer());
	}
}

class Student{
	private int matrikelnummer = 1001;
	public int getMatrikelnummer(){
		return matrikelnummer;
	}
}
```

Der gerade neu erzeugte Datentyp (in der Zeile *Student is = new Student…*) agiert wie eine Unterklasse des ursprünglichen Datentypen. Die Instanz is hat als Datentypen also eine Unterklasse von Student, aber der genaue Datentyp ist anonym, weil wir hier keinen Namen hingeschrieben haben.

Warum ist das jetzt eine innere Klasse? Das liegt daran, dass wir nach der Zeile *Student is = …* neue Methoden schreiben. Wir befinden uns also innerhalb einer neuen Klasse (nicht mehr direkt in Test, sondern innerhalb einer Klasse, die in Test programmiert wird).

Hier ist noch ein Beispiel:

```
public class AnonymousTest {
	public static void main(String[] args){
		Reifen r = new Reifen(2.0){
			public void drehen(){
				System.out.println("Ich drehe mich schneller! Mit " +
					druck + " Bar");
			}
		};

		Reifen r2 = new Reifen(1.8);

		r.drehen();

		r2.drehen();
	}
}
```

```
class Reifen{
	protected double druck;

	public Reifen(double druck){
		this.druck = druck;
	}

	public void drehen(){
		System.out.println("Ich drehe mich");
	}
}

Ausgabe der main-Methode:
Ich drehe mich schneller! Mit 2.0 Bar
Ich drehe mich
```

In der Klasse Reifen wird für das Attribut druck der Zugriffsmodifizierer protected verwendet, weil die anonyme innere Klasse wie eine Unterklasse behandelt wird. Wenn druck private wäre, könnte in der anonymen Klasse nicht darauf zugegriffen werden.

(für ein weiteres Beispiel siehe Skript 20 Folie 34 f.)

Wir haben also die Möglichkeit, mal eben mitten im Code eine neue Unterklasse anzulegen, die wir dann aber nur für eine einzelne Instanz verwenden können. Wofür braucht man das? Zum Beispiel, wenn man GUIs programmiert und einen Button programmieren möchte. Dann ist es üblich, eine Unterklasse vom Standardbutton (JButton) zu erzeugen. Wenn man nun aber für jeden der zehn Buttons in der Oberfläche eine einzelne Klasse bzw. einzelne Datei anlegt, wird es schnell unübersichtlich. Wenn man die Buttons direkt bei der Instaziierung definieren kann, kann das so praktischer sein. Siehe hier:

(Beispielcode zu JButton)

Einschränkungen für anonyme innere Klassen

Wenn Java eine Instanz der anonymen inneren Klasse erzeugt, sucht es in der normalen Klasse nach den Attributen und Methoden. Attribute und Methoden, die dort nicht existieren, können nicht von außerhalb verwendet werden:

```
class Holz{
}

class Test{
	public static void main(String[] args){
		Holz h = new Holz(){
			public void method(){
				System.out.println("yea");
			}
		};

		h.method();
	}
}
```

```
Compilefehler:
Test.java:12: error: cannot find symbol
            h.method();
             ^
  symbol:   method method()
  location: variable h of type Holz
1 error
```

Es ist also nötig, dass die Methoden in der Klasse Holz bereits definiert sind. method kann aber auch keine abstrake Methode sein (wer das jetzt ausprobieren wollte). Das folgende Beispiel funktioniert allerdings, da es nicht notwendig ist, dass die Klasse Test vom Attribut secret etwas weiß.

```
class Holz{
      public void method(){
            System.out.println("2");
      }
}

class Test{
      public static void main(String[] args){
            Holz h = new Holz(){
                  public int secret = 8;

                  public void method(){
                        System.out.println("yea " + secret);
                  }
            };

            h.method();
      }
}
```

Das funktioniert, weil der Zugriff auf das Attribut secret nur innerhalb der anonymen Klasse erfolgt.

Ein bessere Verständnis der Thematik sollte kommen, wenn man sich mit Threads beschäftigt hat.