

# Klassen & Objekte in Java

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller

Universität Innsbruck

# Klassen

- Klassendefinitionen beginnen mit dem Schlüsselwort `class` und dem Namen der Klasse.
- Anschließend folgen in geschweiften Klammern eine beliebige Anzahl von:
  - Feldern (Objekt- und Klassenvariablen)
  - Methoden
  - Konstruktoren
  - Klassen- sowie Exemplarinitialisierer
  - Geschachtelte Klassen, Schnittstellen und Aufzählungen

```
public class Rectangle {  
    ...  
}
```

← Klassendeklaration

# Felder

- Felder werden auch als Attribute oder Membervariablen bezeichnet.
- Statische Felder werden auch Klassenvariablen genannt.
- Objektbezogene Felder werden auch Objektvariablen genannt.
- Objekt- und Klassenvariablen können bei der Deklaration initialisiert werden.

```
public class Rectangle {  
    private int width;  
    private int length;  
    private static int instanceCounter;  
    ...  
}
```

← Objektvariablen

← Klassenvariable

# Methoden

```
public class Rectangle {  
    ...  
    public int getWidth() {  
        return width;  
    }  
  
    public void setWidth(int width) {  
        this.width = width;  
    }  
  
    public int getLength() {  
        return length;  
    }  
  
    public void setLength(int length) {  
        this.length = length;  
    }  
  
    public int getArea() {  
        return width * length;  
    }  
  
    public void printRectangle() {  
        System.out.println("Rectangle width: " + width + ", length: " + length);  
    }  
}
```



# this-Referenz

- Jedes Objekt hat eine this-Referenz.
- this ist in jeder nicht-statischen Methode automatisch definiert.
- Mit this referenziert ein Objekt auf sich selbst.
- Mithilfe von this können Objektvariablen von lokalen Variablen unterschieden werden.
  - Objektvariablen und lokale Variablen können den gleichen Bezeichner haben.
- this kann als Rückgabewert oder Parameter verwendet werden.

```
public class Rectangle {  
  
    ...  
    private int width;  
    ...  
  
    public void setWidth(int width) {  
        this.width = width;  
    }  
    ...  
}
```



# Von der Klasse zum Objekt

- Typ Rechteck ist durch die Klasse Rectangle definiert.
- Nächster Schritt: Erzeugung eines Rectangle-Exemplars

```
Rectangle r1 = new Rectangle(10, 3);
```

↑            ↑            ↑  
Datentyp    Bezeichner    neues Rectangle-Objekt

```
...  
public Rectangle(int width, int length) {  
    this.width = width;  
    this.length = length;  
}  
  
public Rectangle() {  
  
}  
...
```

} Konstruktoren

# Konstruktoren

- Konstruktoren werden bei der Erzeugung von Exemplaren einer Klasse verwendet.
  - Objektvariablen können damit mit sinnvollen Werten belegt oder initialisiert werden.
- In Kombination mit dem Schlüsselwort `new` wird durch einen Konstruktor ein Exemplar erzeugt und eine Referenz darauf zurückgegeben.
- Konstruktor
  - Hat den gleichen Namen wie die Klasse.
  - Wird wie eine Methode ohne die Angabe eines Rückgabetyps deklariert.
- Eine Klasse kann überladene Konstruktoren haben.

# Konstruktoren und Parameter

- Konstruktoren mit Parametern

- Übergabe von Werten (für die Initialisierung)

```
public Rectangle(int width, int length) {...}
```

```
public Rectangle(int sideLength) {...}
```

- Parameterlose Konstruktoren

```
public Rectangle() {...}
```

- Können beliebigen Code enthalten.



# Default-Konstruktor

- Java erzeugt einen Default-Konstruktor automatisch, falls in einer Klasse keine Konstruktoren deklariert sind.
- Sobald explizit ein Konstruktor implementiert wurde, wird der Default-Konstruktor nicht erzeugt.
- Soll eine Klasse zusätzlich einen parameterlosen-Konstruktor haben, muss dieser explizit ausprogrammiert werden.
- Beispiel: Beide Implementierungen bieten dieselbe Funktionalität.

```
public class Point {  
    int x;  
    int y;  
}
```

```
public class Point {  
    int x;  
    int y;  
  
    public Point() {} ← Parameterloser Konstruktor  
}
```

# Konstruktorenverkettung mit this()

- Ein Konstruktor kann mit `this()` (bzw. `this(par1, ...)`) einen anderen Konstruktor derselben Klasse aufrufen.
- Folgende Einschränkungen existieren:
  - Der `this`-Aufruf darf nur einmal vorkommen.
  - Der `this`-Aufruf muss als erste Anweisung auftreten.

```
public class Rectangle {  
  
    ...  
  
    public Rectangle(int width, int length) {  
        this.width = width;  
        this.length = length;  
    }  
  
    public Rectangle(int sideLength) {  
        this(sideLength, sideLength);  
    }  
  
    ...  
  
}
```

# Exemplarinitialisierer

- Die Exemplarinitialisierer einer Klasse werden ausgeführt, wenn ein Exemplar erzeugt wird.
- Eine Klasse kann mehrere Exemplarinitialisierer haben.
- Exemplarinitialisierer eignen sich um Code, welcher am Beginn jedes Konstruktors stehen müsste zu bündeln und so Code-Duplikate zu vermeiden.
- Exemplarinitialisierer werden leicht übersehen, da sie nicht explizit in den Konstruktoren aufgerufen werden und auch nicht explizit in der API-Dokumentation angeführt werden.
- Beide Nachteile können durch den Einsatz einer Initialisierungsmethode, welche in allen Konstruktoren aufgerufen wird, umgangen werden.

# Initialisieren des Objektzustandes

- Feldinitialisierung

```
public class Point {  
    int x = 1;  
    int y = 1;  
}
```

- Initialisierung durch einen Exemplarinitialisierer

```
public class Point {  
    int x;  
    int y;  
    {  
        x = 1;  
        y = 1;  
    }  
}
```

- Initialisierung im Konstruktor
- Automatische Initialisierung

# Automatische Initialisierung

- Klassenvariablen und Objektvariablen, welche nicht `final` sind, werden **automatisch** initialisiert:

Datentyp	Initialisierung
<code>boolean</code>	<code>false</code>
<code>byte</code>	<code>(byte) 0</code>
<code>short</code>	<code>(short) 0</code>
<code>int</code>	<code>0</code>
<code>long</code>	<code>0L</code>
<code>float</code>	<code>0.0f</code>
<code>double</code>	<code>0.0d</code>
<code>char</code>	<code>'\u0000'</code>
Referenztypen (z.B. <code>String</code> )	<code>null</code>

- Lokale Variablen werden nicht automatisch initialisiert.

# Verwendung von Objekten (1)

- Nach der Erzeugung eines Objekts kann dieses verwendet werden.
- Auf die Objektvariablen und die Methoden kann durch Qualifizierung zugegriffen werden:  
    `objekt.Objektvariablenname`  
    `objekt.Methodenname(...)`
- Es wird zwischen einfachen und qualifizierten Bezeichnern unterschieden.
  - Einfache Bezeichner bestehen nur aus einem Namen.
  - Qualifizierte Bezeichner bestehen aus einer Folge von Namen, die jeweils durch einen Punkt getrennt sind.

# Verwendung von Objekten (2)

```
public class RectangleApplication {  
  
    public static void main(String[] args) {  
  
        Rectangle rectangle1 = new Rectangle(20, 3);  
        Rectangle rectangle2 = new Rectangle(10, 5);  
  
        rectangle1.printRectangle();  
        rectangle2.printRectangle();  
  
        System.out.println("Area rectangle 1: " + rectangle1.getArea());  
        System.out.println("Area rectangle 2: " + rectangle2.getArea());  
  
    }  
}
```

Ausgabe:

Rectangle width: 20, length: 3

Rectangle width: 10, length: 5

Area rectangle 1: 60

Area rectangle 2: 50



# Zugriffsmodifikatoren in Java

- **public**
  - Zugriff aus **beliebigen Klassen** erlaubt.
- **private**
  - Zugriff nur aus **derselben Klasse** erlaubt.
- **protected**
  - Zugriff aus **beliebigen Klassen desselben Pakets** und aus **Unterklassen** erlaubt.
- Kein Attribut (Default)
  - Zugriff aus **beliebigen Klassen desselben Pakets** erlaubt.



# private (Unterscheidung)

## Klassenbasierte Sichtbarkeit

- Auf private Daten und Methoden eines Objekts kann nur aus Methoden der Klasse zugegriffen werden, in der diese privaten Elemente deklariert wurden.
- Auf private Elemente anderer Exemplare derselben Klasse kann zugegriffen werden (Klasse bestimmt Sichtbarkeit).
- Wird in Java verwendet.

## Objektbasierte Sichtbarkeit

- Auf private Daten und Methoden eines Objekts kann nur innerhalb von Methoden zugegriffen werden, die auf dem Objekt selbst ausgeführt werden.
- Es kann nicht auf private Elemente anderer Exemplare der Klasse zugegriffen werden.

# Getter- und Setter-Methoden (1)

- Direkter Zugriff auf Objektvariablen sollte nur in seltenen Fällen ermöglicht werden → Kapselung!
- Eine private Objektvariable ist von außen nicht erreichbar.
- Sollen Werte gelesen bzw. geändert werden können:
  - Getter- bzw. Setter-Methode (Accessors, Mutators)
  - Beispiel

```
private type identifier;  
public type getIdentifier() {...}  
public void setIdentifier(type identifier) {...}
```
- Vorteile durch Methoden
  - Kontrolle der übergebenen Werte
  - Hilfsmittel zur Fehlersuche
  - Setter/Getter für scheinbare Datenelemente
  - Implementierungsdetails verbergen

# Getter- und Setter-Methoden (2)

- Es muss nicht für jedes Feld einer Klasse ein Getter und Setter angeboten werden.
- Ein intensiver Gebrauch von Getter- und Setter-Methoden ist kein Zeichen von guter Objektorientierung.
  - Deutet eher auf einen fragwürdigen OO-Entwurf hin.
    - Objekt verkommt zu einem Datencontainer – das ist nicht Objektorientierung!
    - Das Verhalten des Objekts kann zu sehr von anderen Klassen gesteuert werden.

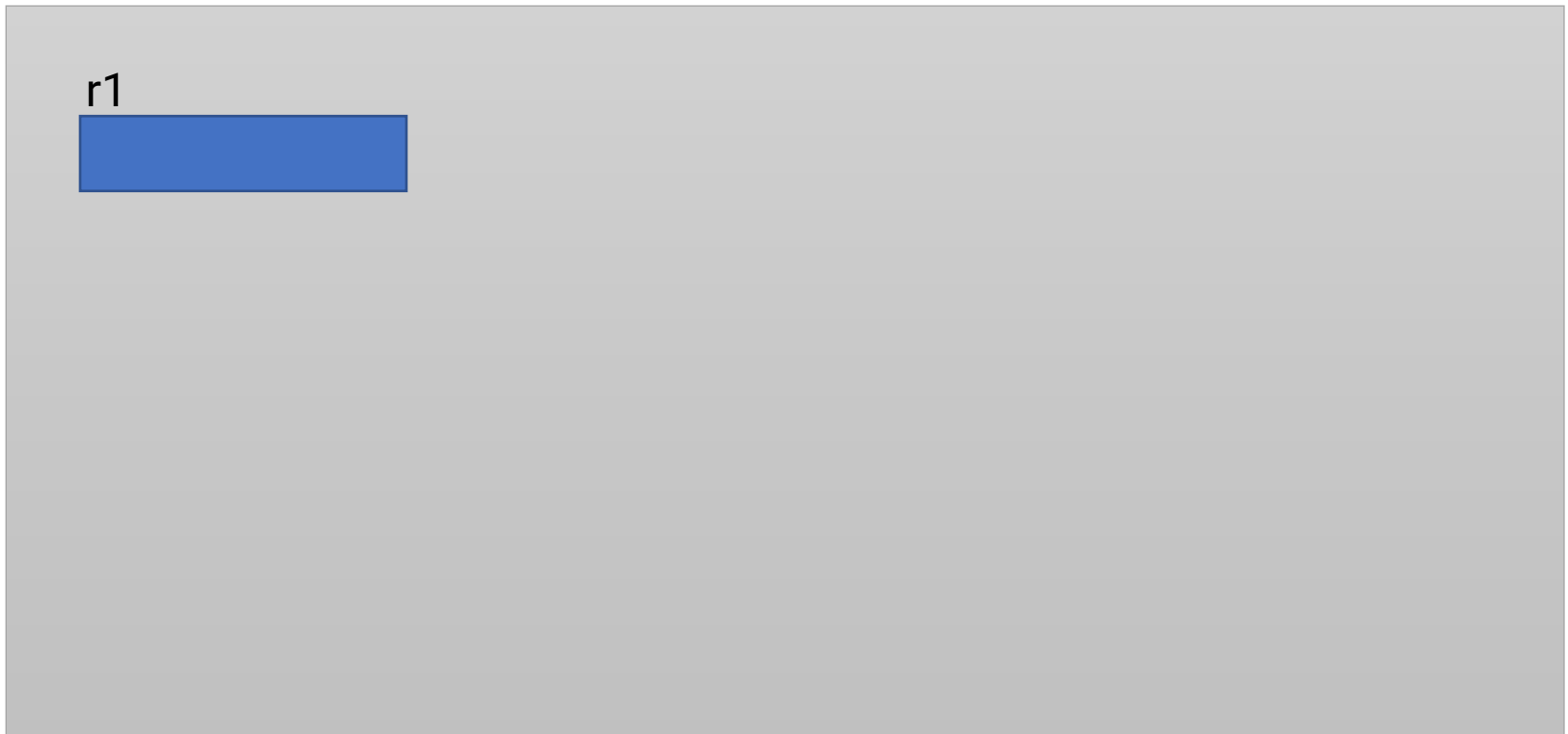
```
public class Rectangle {  
    private int width;  
    ...  
    public int getWidth() {  
        return width;  
    }  
  
    public void setWidth(int width) {  
        this.width = width;  
    }  
    ...  
}
```



# Objekte und Referenzen (1)

Rectangle **r1**;

Speicherbereich

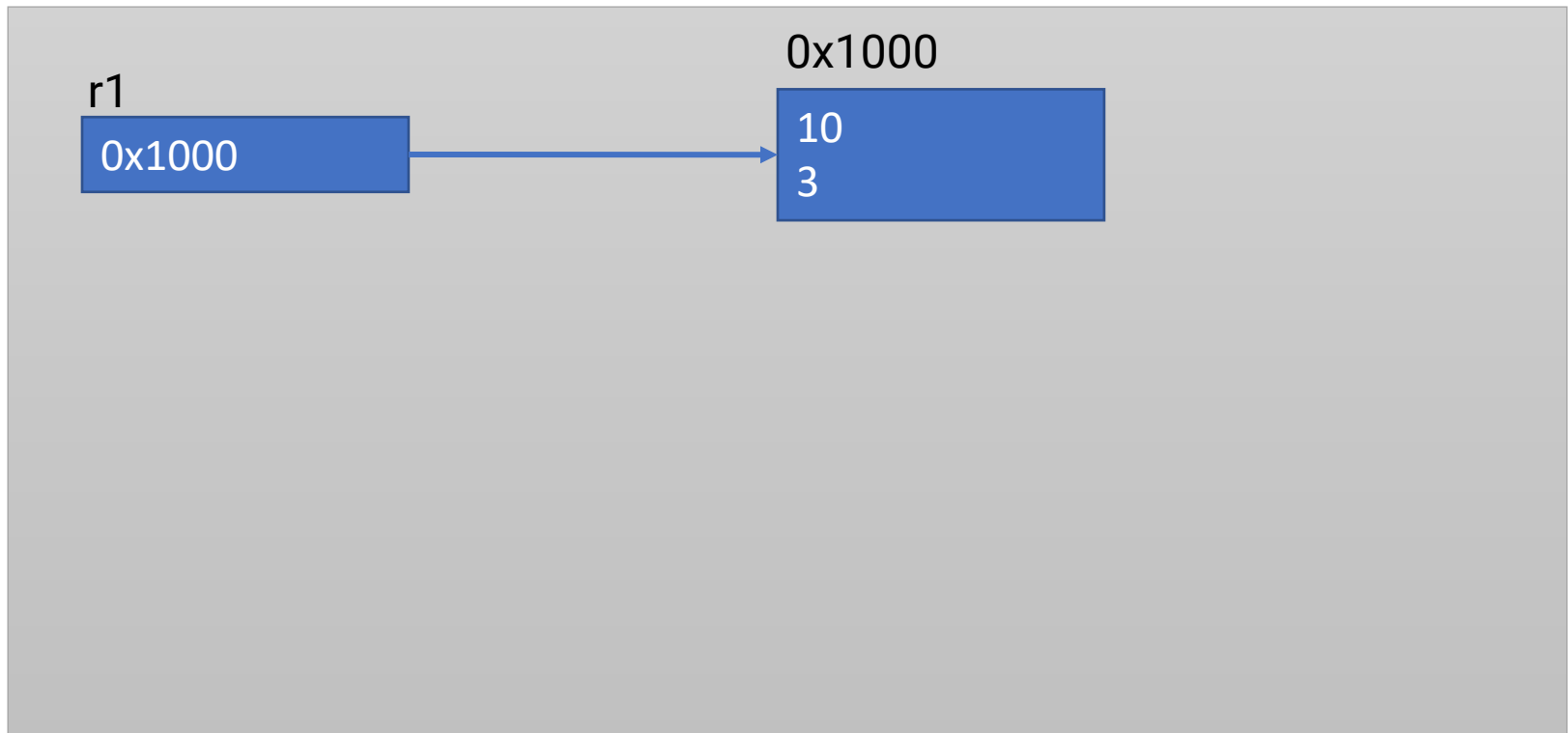


# Objekte und Referenzen (2)

Rectangle **r1**;

**r1** = **new** Rectangle(10, 3);

Speicherbereich



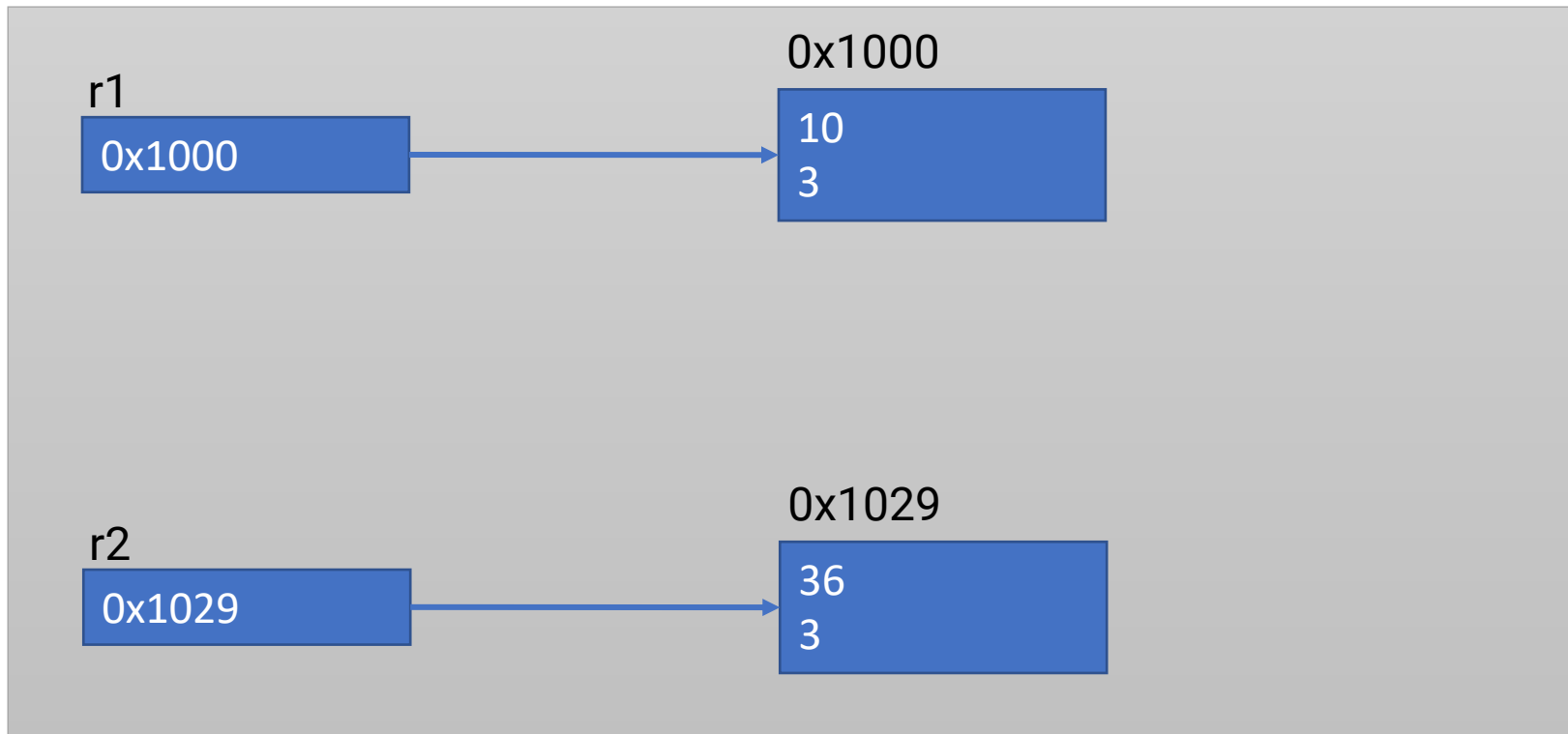
# Objekte und Referenzen (3)

Rectangle **r1**;

**r1** = **new** Rectangle(10, 3);

Rectangle **r2** = **new** Rectangle(36, 3);

Speicherbereich



# Objekte und Referenzen (4)

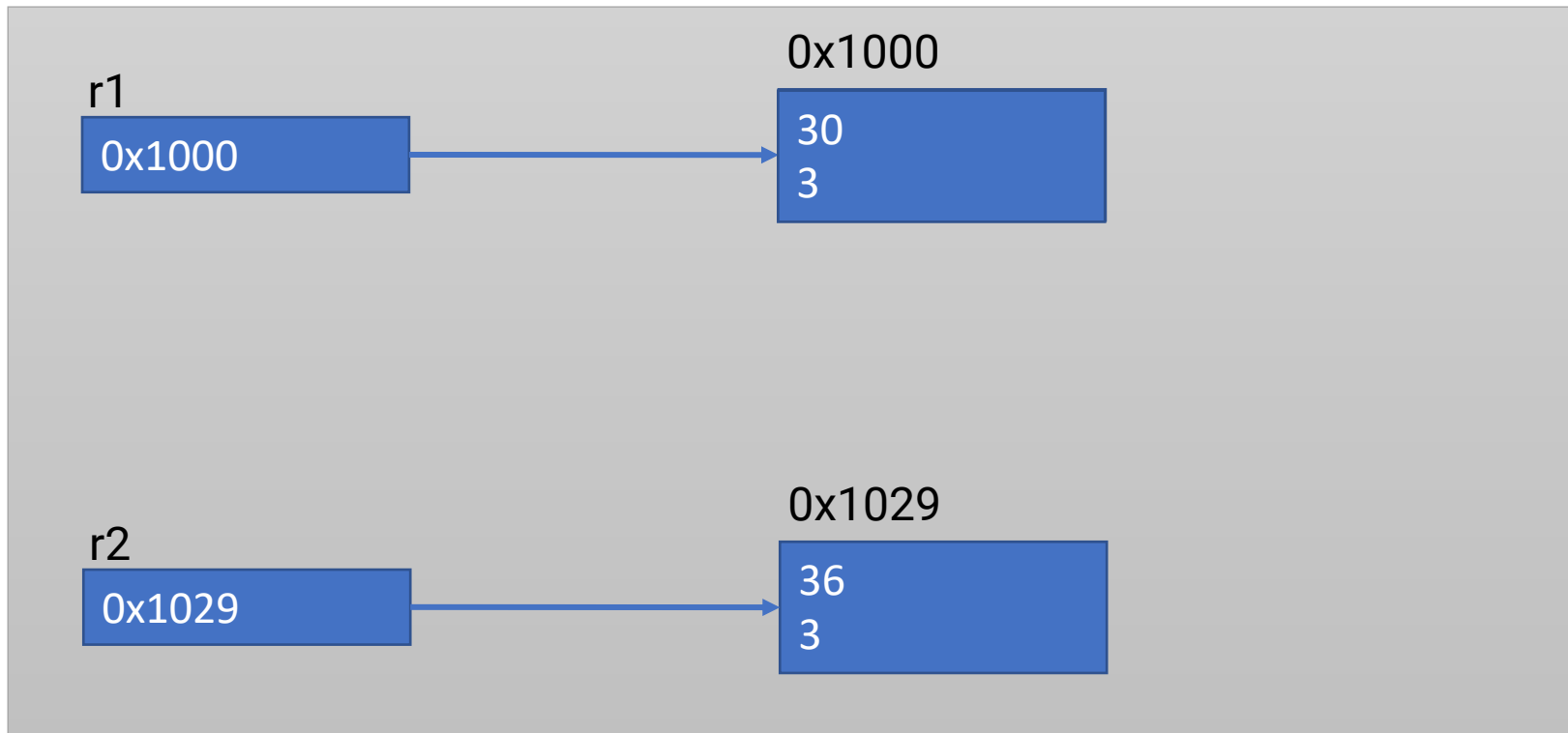
Rectangle **r1**;

**r1** = **new** Rectangle(10, 3);

Rectangle **r2** = **new** Rectangle(36, 3);

**r1**.setWidth(30);

Speicherbereich

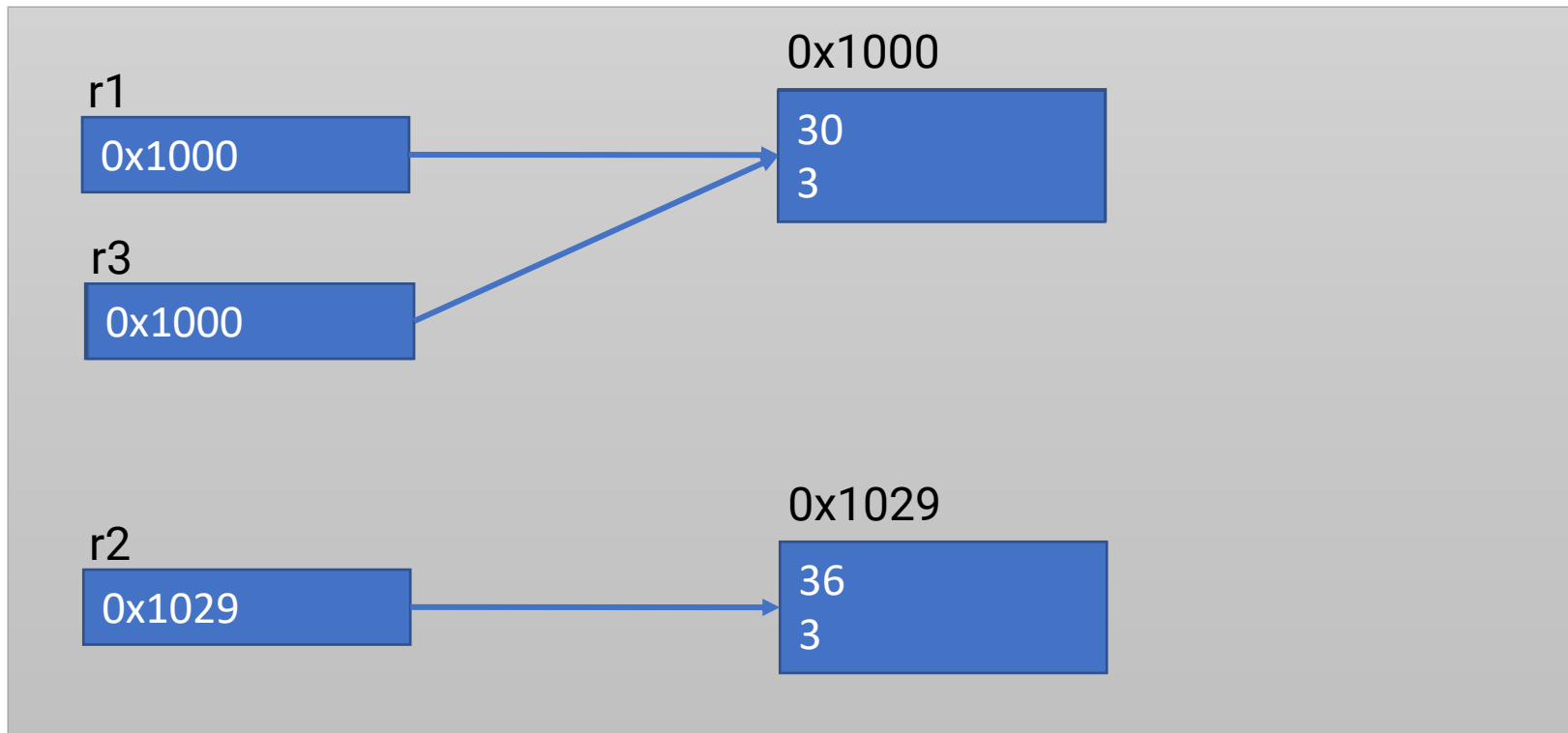


# Objekte und Referenzen (5)

...

Rectangle **r3** = **r1**;

Speicherbereich



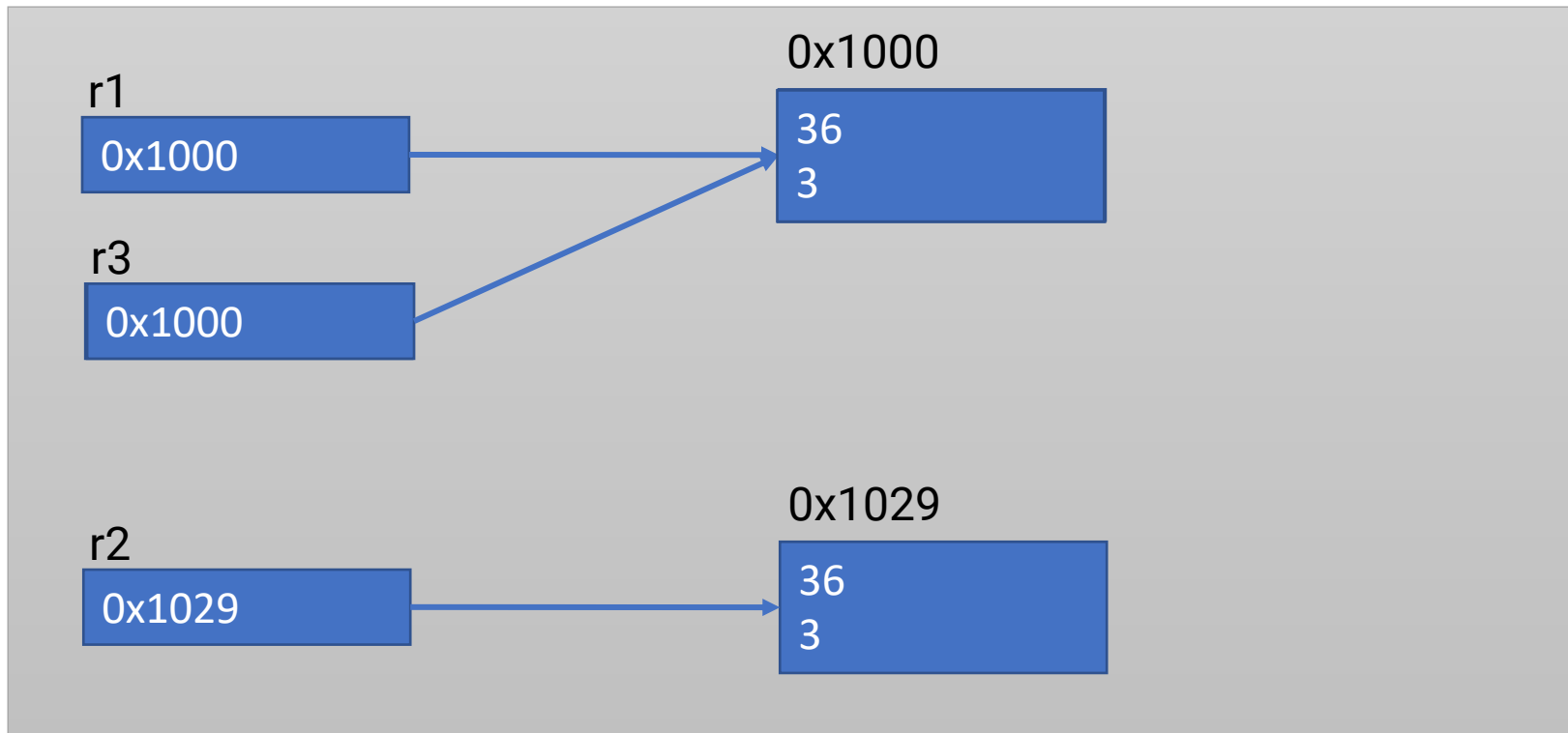


# Objekte und Referenzen (6)

...

```
Rectangle r3 = r1;  
r1.setWidth(36);
```

Speicherbereich



# Vergleiche

- Objekte können mit `==` und `!=` verglichen werden.
  - **Achtung:** Es werden nur die Referenzen und nicht die Werte verglichen.
- Eine Überprüfung, ob Objekte denselben Wert haben, geschieht durch:
  - Eine eigene Vergleichsmethode
  - Den Vergleich der einzelnen Objektvariablen

```
...  
System.out.println(r1 == r2); // false  
System.out.println(r1 == r3); // true  
System.out.println(r2 == r3); // false  
...
```

# Kopieren von Objekten

- Bei der Zuweisung von Referenzvariablen wird nur die Referenz kopiert. Es wird keine Kopie des Objekts erzeugt.
- Ein Objekt kann beispielsweise durch einen Kopier-Konstruktor (Copy-Constructor) oder die `clone()`-Methode kopiert werden.
- Beim Kopieren von Objekten wird zwischen flacher und tiefer Kopie unterschieden.
- Flache Kopie (shallow copy)
  - Es wird nur das Objekt selbst und die darin enthaltenen Werte kopiert.
  - Bei Referenzvariablen wird im Original und der Kopie auf dasselbe Objekt verwiesen.
- Tiefe Kopie (deep copy)
  - Enthält ein Objekt weitere Objekte, so wird das Kopieren rekursiv fortgesetzt.
  - Original und Kopie enthalten logisch gleiche aber getrennte Datenelemente.

# Beispiel Kopier-Konstruktor

```
public class Rectangle {  
  
    private int width;  
    private int length;  
  
    ...  
    public Rectangle(int width, int length) {  
        this.width = width;  
        this.length = length;  
    }  
  
    public Rectangle(Rectangle toCopy) {  
        this.width = toCopy.width;  
        this.length = toCopy.length;  
    }  
    ...  
}
```

„herkömmlicher“ Konstruktor

Copy-Constructor

```
...  
// usage  
Rectangle r4 = new Rectangle(20, 5);  
Rectangle r5 = new Rectangle(r4);  
...
```





# Statisch vs. Objektbezogen

# Statische Methoden und Felder

- Bisher in diesem Foliensatz: alle Methoden sind an Objekte gebunden.
- Methoden und auch Felder sind nicht immer direkt von Objekten abhängig
  - `Math.max()` – ermittelt das Maximum zweier Zahlen
  - `Math.PI` – Annäherung der Kreiszahl Pi ( $\pi$ )
  - `Integer.parseInt()` – wandelt String in Integer um
- Derartige Methoden sollten nicht dem Objekt, sondern der Klasse zugeordnet sein (unabhängig vom Objekt-Zustand).
- Zugriff auf statische Felder und Methoden
  - `Classname.field` bzw. `Classname.method(...)`
  - Innerhalb der Klasse kann die Qualifizierung (`Classname.`) weggelassen werden

# Statische Methoden

- Statische Methoden werden mit dem Schlüsselwort `static` deklariert.
- Sie werden auch als Klassenmethoden bezeichnet.
- Eine statische Methode wird immer ohne Bezug auf ein bestimmtes Objekt bearbeitet.
- Bei der Deklaration von Klassenmethoden wird ein statischer Kontext eingeführt.
- In einem statischen Kontext kann weder explizit noch implizit auf das aktuelle Exemplar der Klasse verwiesen werden.
- Es gibt beispielsweise folgende Einschränkungen:
  - Die Verwendung von `this` und `super` ist nicht möglich.
  - Unqualifizierte Referenzen auf Objektvariablen und objektbezogene Methoden sind nicht möglich.

# Statische Felder

- Statische Felder werden mit dem Schlüsselwort `static` deklariert.
- Sie werden auch als Klassevariablen oder statische Variablen bezeichnet.
- Ein statisches Feld existiert für eine Klasse immer genau einmal unabhängig davon wie viele Exemplare der Klasse existieren.
- Bei der Deklaration eines statischen Feldes wird ein statischer Kontext eingeführt.



# Rectangle-Beispiel mit statischem Zähler (1)

```
public class Rectangle {  
    private int width;  
    private int length;  
    private static int instanceCounter; // bound to class  
  
    Rectangle(int width, int length) {  
        this.width = width;  
        this.length = length;  
        ++instanceCounter; // bound to class  
    }  
    ...  
    public static int getInstanceCounter() { // bound to class  
        return instanceCounter;  
    }  
    ...  
}
```



# Rectangle-Beispiel mit statischem Zähler (2)

```
public class RectangleApplication {  
  
    public static void main(String[] args) {  
        Rectangle rectangle1 = new Rectangle(20, 3);  
        System.out.println("Instances: " + Rectangle.getInstanceCounter());  
        Rectangle rectangle2 = new Rectangle(10, 5);  
        System.out.println("Instances: " + Rectangle.getInstanceCounter());  
  
        rectangle1.printRectangle();  
        rectangle2.printRectangle();  
  
        System.out.println("Area rectangle 1: " + rectangle1.getArea());  
        System.out.println("Area rectangle 2: " + rectangle2.getArea());  
    }  
}
```

Ausgabe:

Instances: 1

Instances: 2

Rectangle width: 20, length: 3

Rectangle width: 10, length: 5

Area rectangle 1: 60

Area rectangle 2: 50



# Statischer Initialisierer

- Für die Initialisierung statischer Variablen kann auch ein statischer Initialisierer verwendet werden.
  - Hat keinen Namen und keine Parameter

```
static {  
    instanceCounter = 0;  
}
```

- Wird aufgerufen, wenn die Klasse geladen wird.
- Auch mehrere statische Blöcke sind möglich.

# Objektbezogen oder statisch

	Objektbezogen	Statisch
<b>Deklarationen</b>	ohne <code>static</code>	mit <code>static</code>
<b>Existieren</b>	in jedem Objekt	nur einmal pro Klasse
<b>Variablen werden angelegt</b>	wenn das Objekt erzeugt wird	wenn die Klasse geladen wird
<b>Variablen werden freigegeben</b>	vom Garbage-Collector, wenn keine Referenz mehr auf das Objekt existiert	wenn die Klasse entladen wird
<b>Konstruktor/Initialisierer wird aufgerufen</b>	wenn das Objekt erzeugt wird	wenn die Klasse geladen wird
<b>Qualifizierung</b>	über das Objekt	über den Klassennamen (oder das Objekt)



# Beziehungen zwischen Klassen

# Beziehungen zwischen Klassen

- In einem objektorientierten Programm stehen Klassen (und damit Objekte) in Beziehung. Es gibt selten isolierte Objekte.
- Beispiel Verwaltung von Studierenden und Kursen

- Klasse Address

 <src/at/ac/uibk/pm/objectorientation/coursemanagement/Address.java>

- Klasse ContactInformation

- Beziehung: Kontaktdaten enthalten eine Adresse

 <src/at/ac/uibk/pm/objectorientation/coursemanagement/ContactInformation.java>


- Klasse Person

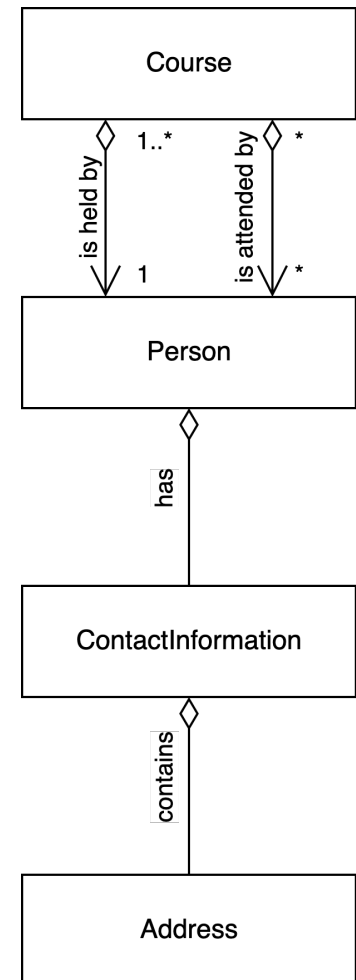
- Beziehung: Jede Person hat Kontaktdaten.

 <src/at/ac/uibk/pm/objectorientation/coursemanagement/Person.java>

- Klasse Course

- Beziehung: zu einem Kurs können sich mehrere Personen anmelden.
- Beziehung: eine Person leitet den Kurs.

 <src/at/ac/uibk/pm/objectorientation/coursemanagement/Course.java>



# Kohäsion und Kopplung

- Kohäsion

- Kohäsion beschreibt wie gut eine Methode tatsächlich genau eine Aufgabe erfüllt oder wie genau abgegrenzt die Funktionalität einer Klasse ist.
- Eine **hohe Kohäsion** deutet auf eine gute Trennung der Zuständigkeiten hin.
  - Das bedeutet eine Methode oder Klasse soll nur eine bestimmte Aufgabe erfüllen.
  - Das ist eine wünschenswerte Eigenschaft von objektorientiertem Code.
- Hohe Kohäsion hilft bei der Wiederverwendung.

- Kopplung

- Darunter versteht man, wie stark Klassen miteinander in Verbindung stehen.
- Ziel einer guten Modellierung ist eine möglichst **lose Kopplung**!
  - Geringe Abhängigkeiten von Klassen untereinander.
  - Eine gute Datenkapselung und sinnvolle Zugriffsmethoden ermöglichen dies!

# Quellen

- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Guido Krüger, Heiko Hansen: **Handbuch der Java-Programmierung**, Addison Wesley, 7. Auflage, 2011
- Christian Silberbauer: **Einstieg in Java und OOP**, Springer Vieweg, 2. Auflage, 2020