

# Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

# Kapselung

- Modellierung und Programmierung
- Verschachtelte Klassen
  - Innere Klassen
  - Statische Klassen
- Finale Klassen, Methoden und Platzhalter
- Patterns
- Packages
- Module

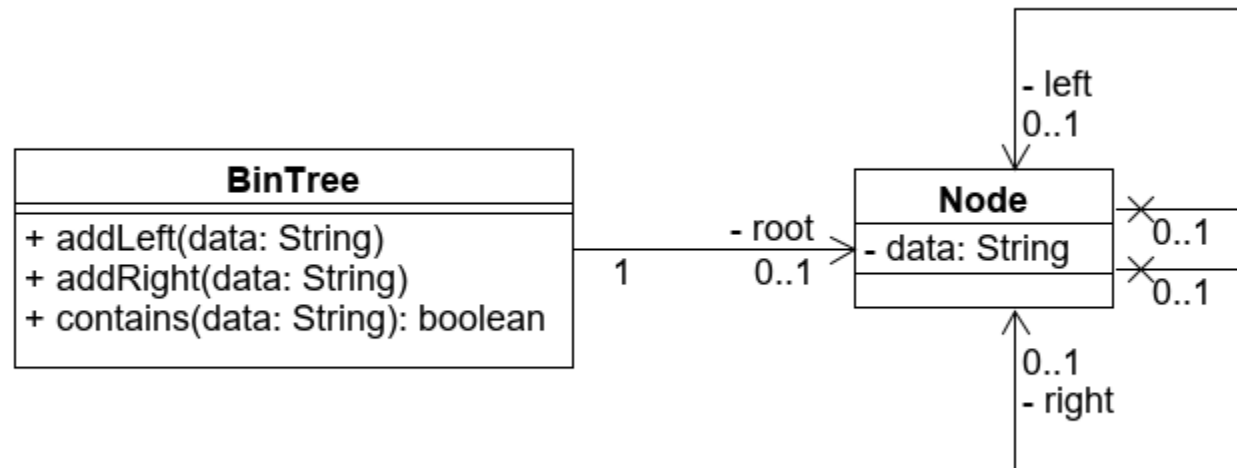
# Modellierung und Programmierung

- Was kommt zuerst, das Klassendiagramm oder der Code?
- Entwurf eines neuen Systems
  - erst ein oder mehrere grobe Klassendiagramme
  - dann mehrere verfeinerte Klassendiagramme
  - dann der Code
  - dann immer wieder
    - Anpassungen am Diagramm, die im Code reflektiert werden
    - Anpassungen am Code, die in den Diagrammen reflektiert werden

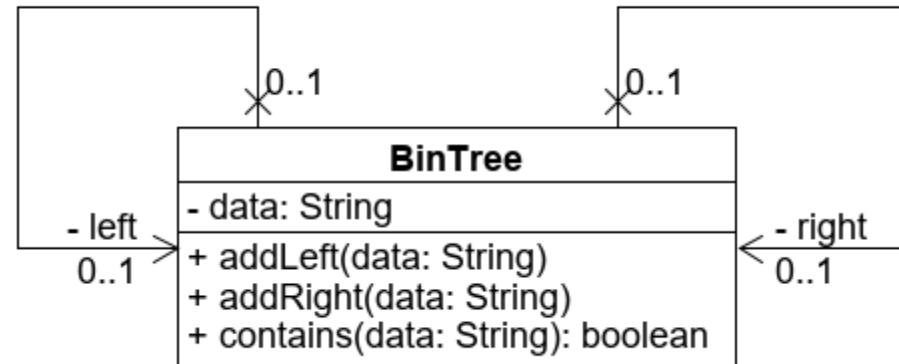
## Henne-Ei-Problem (2)

- Was kommt zuerst, das Klassendiagramm oder der Code?
- **Analyse eines bestehenden Systems**
  - Code existiert bereits
  - dazu wird ein großes umfassendes Klassendiagramm erstellt
  - dieses wird in verschiedene Bereiche unterteilt
  - dann werden ein oder mehrere grobe Übersichts-Klassendiagramme erstellt
  - wenn neue Aspekte betrachtet werden müssen, werden neue Klassendiagramme mit besonderem Fokus erstellt

# Modellierung von Binärbäumen

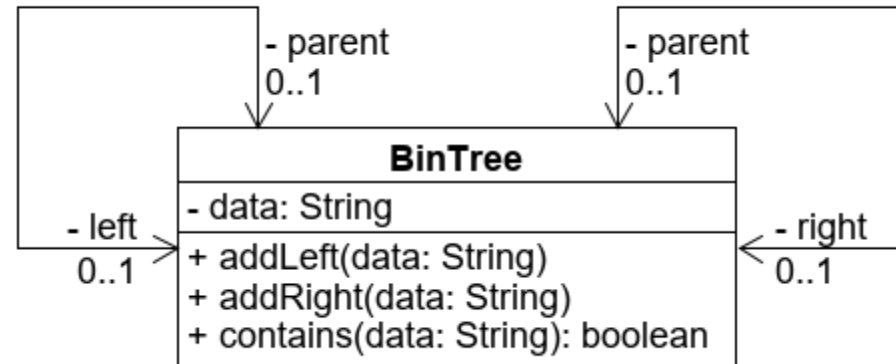


# Modellierung von Binärbäumen (2)

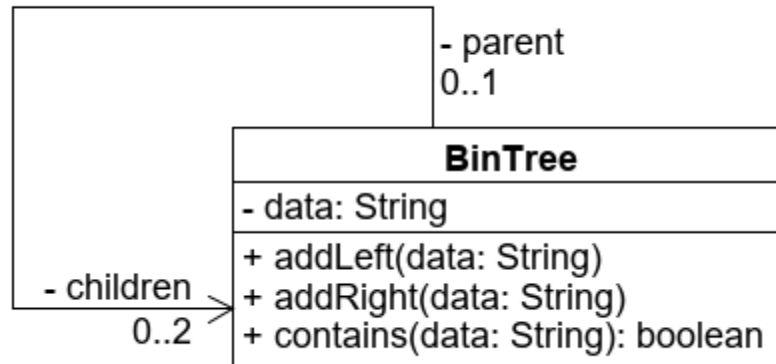




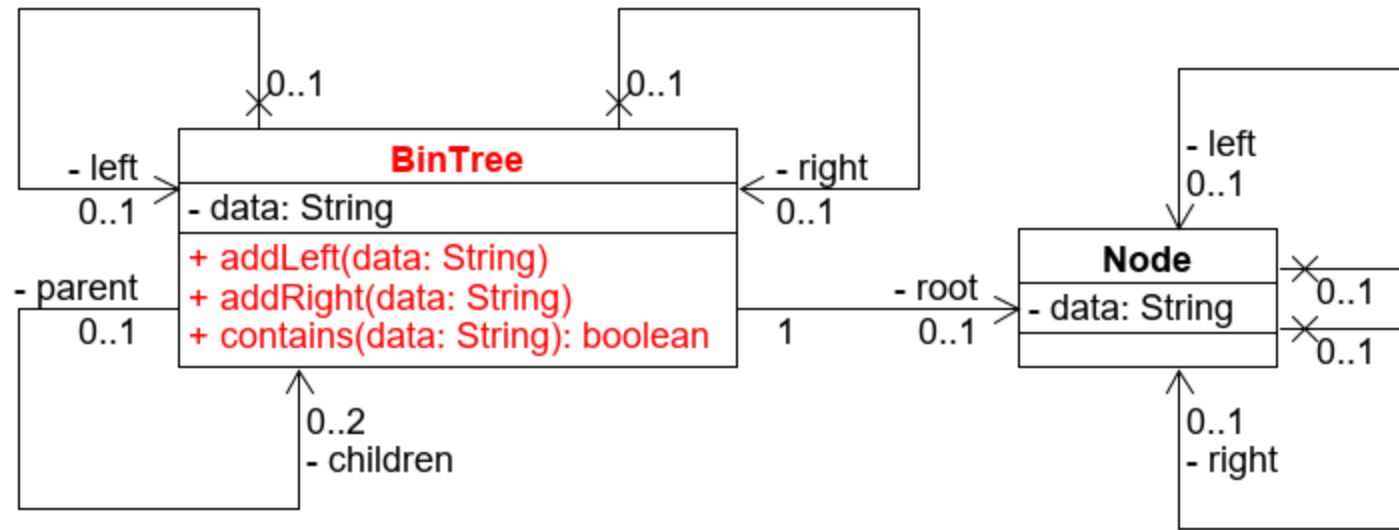
# Modellierung von Binärbäumen (3)



# Modellierung von Binärbäumen (4)

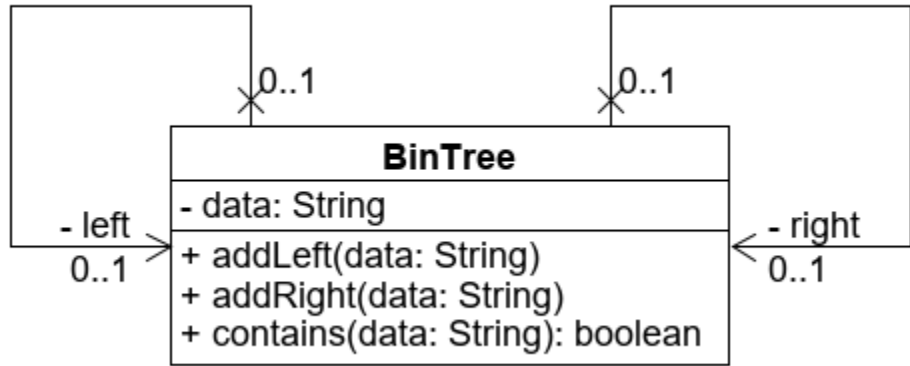


# Modellierung von Binärbäumen (Vergleich)



Die öffentlichen Methoden der Klasse **BinTree** sind die **Schnittstelle nach außen**.  
Die **Details** der Modellierung und der Implementierung sind **versteckt** und **austauschbar**.

# Implementierung eines UML-Modells

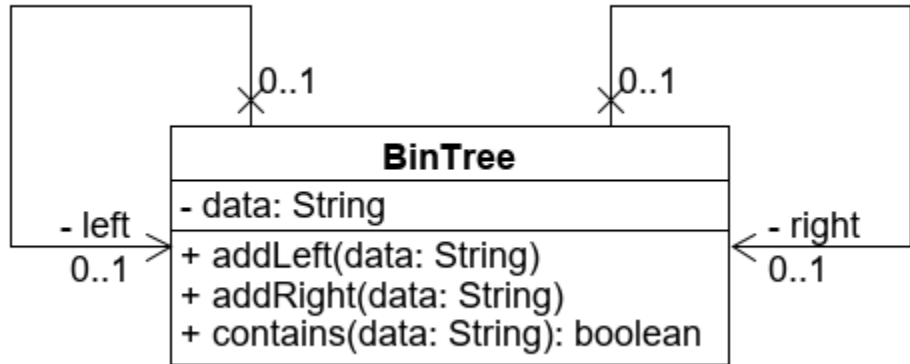


```
public class BinTree {
```

```
}
```



# Implementierung eines UML-Modells



```
public class BinTree {
```

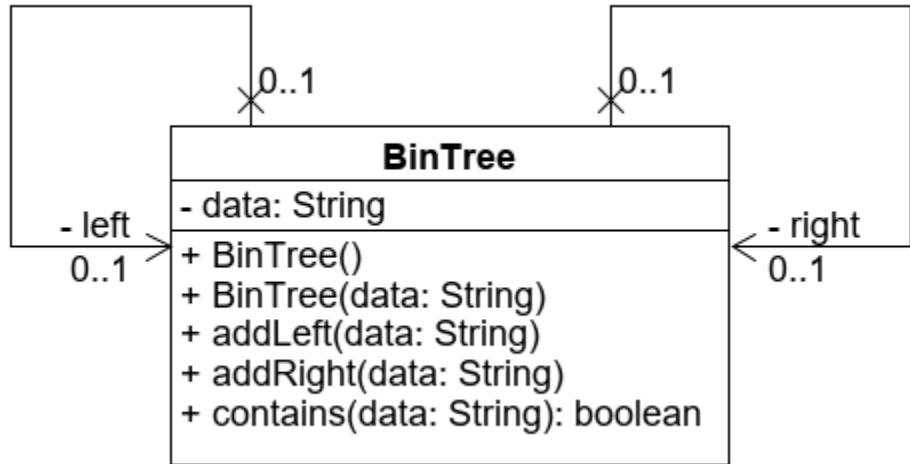
```
    private String data;
    private BinTree left;
    private BinTree right;
```

```
    public void addLeft(String data) {
        left = new BinTree(data);
    }
```

```
    public void addRight(String data) {
        right = new BinTree(data);
    }
```

```
}
```

# Implementierung eines UML-Modells



```

public class BinTree {

    private String data;
    private BinTree left;
    private BinTree right;

    public BinTree() { }

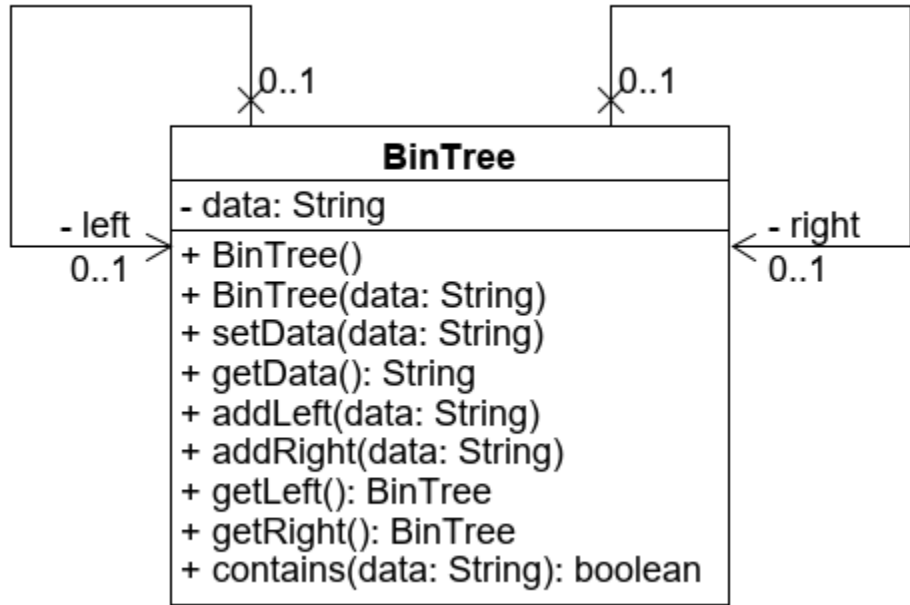
    public BinTree(String data) {
        this();
        this.data = data;
    }

    public void addLeft(String data) {
        left = new BinTree(data);
    }

    public void addRight(String data) {
        right = new BinTree(data);
    }

}
  
```

# Implementierung eines UML-Modells



```
public class BinTree {

    ...

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }

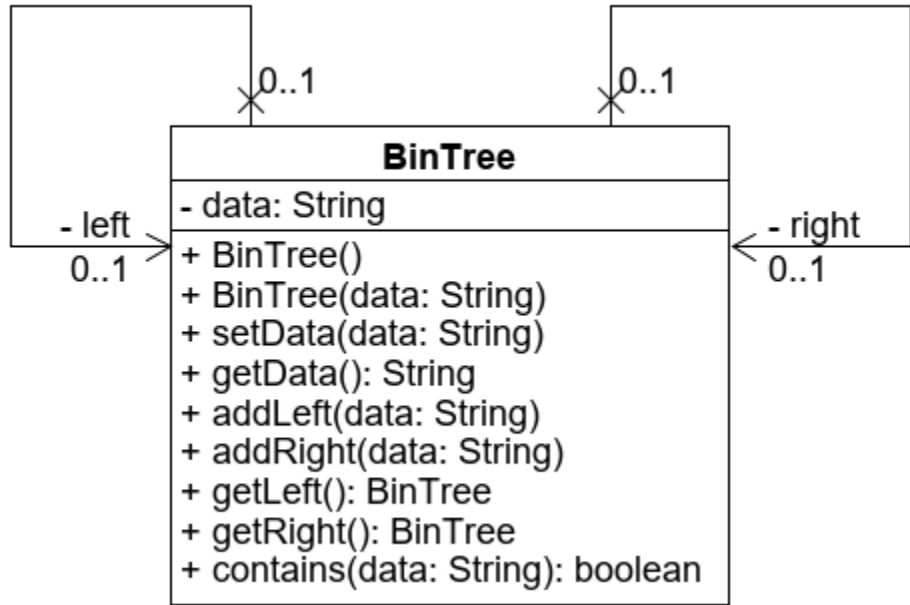
    public BinTree getLeft() {
        return left;
    }

    public BinTree getRight() {
        return right;
    }

}
```



# Implementierung eines UML-Modells



```

public class BinTree {

    ...

    public boolean contains(String data) {
        if (this.data != null
            &&
            this.data.equals(data)) {
            return true;
        }
        return (getLeft() != null
            &&
            getLeft().contains(data))
            ||
            (getRight() != null
            &&
            getRight().contains(data));
    }

}

```

# Verschachtelte Klassen

(Nested Classes)

- Das Konzept der Kapselung ist bereits bekannt von...
  - Abstrakten Datentypen
  - Internem Objektzustand
  - Interfaces
  - ...

- Es gibt Klassen, die eng zusammengehören
  - einige davon sind nur für den „internen Gebrauch“
  - z.B. **LinkedList** und **LinkedListElement**, letztere wird nur intern verwendet
- Zwischen diesen Klassen besteht ein logischer Zusammenhang
- Zusätzlich ist es sinnvoll, einige dieser Klassen vor der Außenwelt zu verstecken: **Kapselung**
- Ein Mechanismus dafür ist das **Verschachteln von Klassen**
  - dabei werden eine oder mehrere Klassen **innerhalb** einer anderen Klasse definiert

# Verschachtelte Klassen

- Bekannte Elemente von Klassen
  - Attribute (statisch und nicht-statisch)
  - Konstanten
  - Konstruktoren
  - Methoden (statisch und nicht-statisch)
- Zusätzliche mögliche Elemente von Klassen
  - Klassen

# Verschachtelte Klassen

- Nested Class (Verschachtelte Klasse)
  - Inner Class
    - Member Class
    - Local Class → später
      - Anonymous Class → später
      - Lambda Expression → später
  - Static Class

# Beispiel: Inner Class

```
public class LinkedList {  
  
    private LinkedListElement head;  
  
    public Object get(int index) { ... }  
  
    public void add(Object value) { ... }  
  
    public void remove(Object value) { ... }  
  
    public boolean contains(Object value) { ... }  
  
    private class LinkedListElement {  
  
        private LinkedListElement next;  
        private Object value;  
  
        // getter and setter methods  
    }  
}
```

# Inner Class

- Sichtbarkeit der Inner Class kann wie für andere Klassen-Elemente festgelegt werden: **public**, **package**, **protected**, **private**
- Die Inner Class kann auf alle Elemente der Outer Class zugreifen und umgekehrt (inkl. **private** Elemente!)
- Die Inner Class befindet sich daher immer im Kontext der Outer Class
  - eine Instanz der Inner Class hängt immer von einer Instanz der Outer Class ab
  - bei der Instanziierung einer Inner Class muss eine Instanz der Outer Class vorliegen



# Beispiel: Instanziierung einer Inner Class

```
public class LinkedList {  
  
    private LinkedListElement head;  
  
    public void add(String value) {  
        LinkedListElement newElement = new LinkedListElement(value);  
        if (head == null) {  
            head = newElement;  
        }  
        // ...  
    }  
  
    private class LinkedListElement {  
  
        public LinkedListElement(String value) {  
            // ...  
        }  
  
    }  
  
}
```

# Instanziierung einer Inner Class

```
public class OuterClass {  
  
    public void test() {  
        InnerClass inner = new InnerClass();  
    }  
  
    public class InnerClass {  
  
    }  
  
}
```

Instanziierung der Inner Class aus der Outer Class heraus

```
public class OtherClass {  
  
    public static void main(String[] args) {  
        OuterClass outerInstance = new OuterClass();  
        OuterClass.InnerClass innerInstance = outerInstance.new InnerClass();  
    }  
  
}
```

Instanziierung der Inner Class von außerhalb benötigt eine Instanz der Outer Class

# Static (Nested) Class

- Statische Klassen können instanziiert werden
  - sie sind aber unabhängig von einer konkreten Instanz der Outer Class
  - für die Instanziierung ist **keine** Instanz der Outer Class nötig
- Sie hat nur Zugriff auf die statischen Elemente der Outer Class
- Elemente der Static Class müssen nicht statisch sein

# Beispiel: Static Class

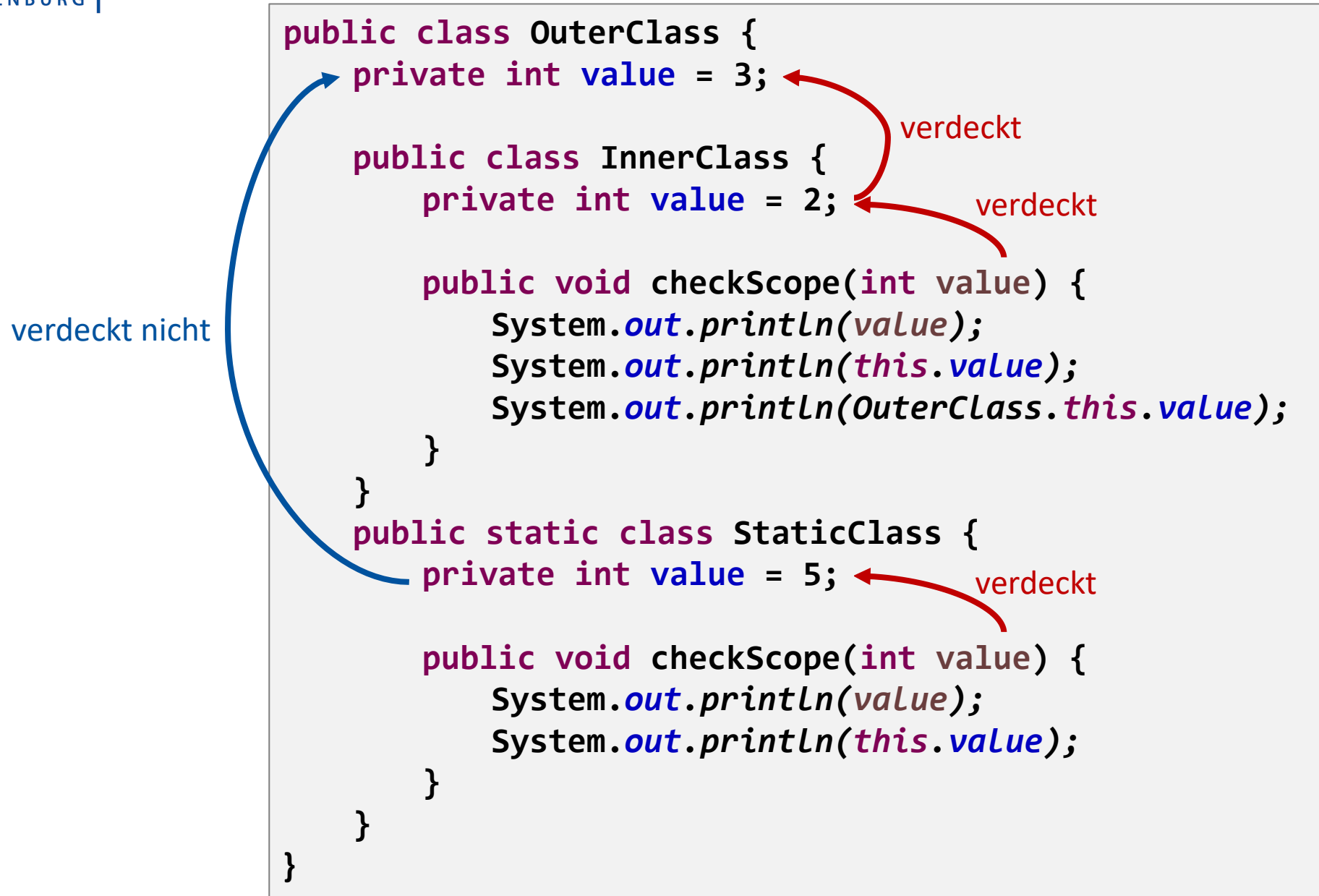
```
public class Geometry {  
  
    public static class Vector {  
        private int x;  
        private int y;  
        // constructors, getter and setter methods  
    }  
  
    public static Vector add(Vector a, Vector b) {  
        Vector result = new Vector();  
        result.x = a.x + b.x;  
        result.y = a.y + b.y;  
        return result;  
    }  
}
```

Instanziierung der Static Class von außerhalb  
benötigt **keine** Instanz der Outer Class

```
public class OtherClass {  
  
    public static void main(String[] args) {  
        Geometry.Vector a = new Geometry.Vector(5, 3);  
        Geometry.Vector b = new Geometry.Vector(2, 8);  
        Geometry.Vector c = Geometry.add(a, b);  
        System.out.println(c.getX() + "/" + c.getY());  
    }  
}
```

- Parameter, Variablen und Attribute mit gleichem Namen **verdecken** (**shadow**) andere Platzhalter von innen nach außen
- Der Zugriff auf einen verdeckten Platzhalter **data** ist weiterhin über **this.data** oder über **ClassName.this.data** möglich
  - **this** verweist auf die Instanz der aktuellen Klasse, egal ob es eine Outer Class, Inner Class oder Static Class ist
  - **ClassName.this** verweist auf die Instanz der Outer Class **ClassName**

# Beispiel: Shadowing



# Beispiel: Shadowing

```
public class OuterClass {  
    private int value = 3;  
  
    public class InnerClass {  
        private int value = 2;  
  
        public void checkScope(int value) {  
            System.out.println(value);  
            System.out.println(this.value);  
            System.out.println(OuterClass.this.value);  
        }  
    }  
  
    public static class StaticClass {  
        private int value = 5;  
  
        public void checkScope(int value) {  
            System.out.println(value);  
            System.out.println(this.value);  
        }  
    }  
}
```

```
public class OtherClass {  
  
    public static void main(String[] args) {  
        OuterClass outerInstance = new OuterClass();  
        OuterClass.InnerClass innerInstance =  
            outerInstance.new InnerClass();  
        innerInstance.checkScope(1);  
  
        OuterClass.StaticClass staticInstance =  
            new OuterClass.StaticClass();  
        staticInstance.checkScope(4);  
    }  
}
```

1  
2  
3  
4  
5

# Finale Klassen, Methoden und Attribute



- Die Erweiterung von Klassen kann zu Problemen führen, wenn bei der Konzeption der Oberklasse eine mögliche Erweiterung nicht berücksichtigt wurde
  - es können Methoden überschrieben werden, die von anderen Methoden der Oberklasse verwendet werden
  - dadurch kann es zu unerwartetem Verhalten der gesamten Klasse kommen, auch in durch die Unterklasse unveränderten Bereichen
- Mögliche Folgerungen
  - Wer eine Klasse überschreibt, sollte wissen, was er/sie/es tut
  - Wer eine Klasse schreibt, sollte mögliche Unterklassen berücksichtigen
  - Wer eine Klasse schreibt, sollte Unterklassen verbieten oder Teile der Klasse gegen Überschreibungen sichern
    - finale Klassen und Methoden

# Beispiel: Fehler durch Überschreiben

```
public class LinkedList {  
  
    private LinkedListElement start;  
  
    public Object get(int index) { ... }  
  
    /** Add the given value to the list */  
    public void add(Object value) { ... }  
  
    /** Add the given value to the list or replace it  
     * if it is already in the list and if replace==true */  
    public void add(Object value, boolean replace) { ... }  
  
    public void remove(Object value) { ... }  
  
    public boolean contains(Object value) { ... }  
  
    public void print() { ... }  
  
    private class LinkedListElement { ... }  
}
```

# Beispiel: Fehler durch Überschreiben

```
public class LinkedList {  
  
    private LinkedListElement start;  
  
    public Object get(int index) { ... }  
  
    /** Add the given value to the list */  
    public void add(Object value) { ... }  
  
    /** Add the given value to the list or replace it  
     * if it is already in the list and if replace==true */  
    public void add(Object value, boolean replace) { ... }  
  
    public void remove(Object value) { ... }  
  
    public boolean contains(Object value) { ... }  
  
    public void print() { ... }  
  
    private class LinkedListElement { ... }  
}
```

# Beispiel: Fehler durch Überschreiben

```
public void add(Object value) {
    add(value, false);
}

public void add(Object value, boolean replace) {
    if (start == null) { // list is empty
        start = new LinkedListElement(value);
    } else {
        // check first element
        if (replace && start.getValue().equals(value)) {
            start.setValue(value);
            return;
        }
        // move to the end of the list
        LinkedListElement current = start;
        while (current.getNext() != null) {
            current = current.getNext();
            if (replace && current.getValue().equals(value)) {
                current.setValue(value);
                return;
            }
        }
        current.setNext(new LinkedListElement(value));
    }
}
```

# Beispiel: Fehler durch Überschreiben

```
public void add(Object value) {
    add(value, false);
}

public void add(Object value, boolean replace) {
    if (start == null) { // list is empty
        start = new LinkedListElement(value);
    } else {
        // check first element
        if (replace && start.getValue().equals(value)) {
            start.setValue(value);
            return;
        }
        // move to the end of the list
        LinkedListElement current = start;
        while (current.getNext() != null) {
            current = current.getNext();
            if (replace && current.getValue().equals(value)) {
                current.setValue(value);
                return;
            }
        }
        current.setNext(new LinkedListElement(value));
    }
}
```

# Beispiel: Fehler durch Überschreiben

```
public class InformativeLinkedList extends LinkedList {  
  
    @Override  
    public void add(Object value, boolean replace) {  
        if (replace && contains(value)) {  
            remove(value);  
            System.out.println("Value '" + value.toString() + "' was replaced!");  
        }  
        add(value);  
    }  
}
```

# Beispiel: Fehler durch Überschreiben

```
LinkedList list = new LinkedList();  
list.add(1);  
list.add(2);  
list.add(1, true);  
list.add(1, false);  
list.print(); // [ 1, 2, 1 ]  
  
list = new InformativeLinkedList();  
list.add(1); // java.lang.StackOverflowError  
list.add(2);  
list.add(1, true);  
list.add(1, false);  
list.print();
```

- Eine Finale Klasse (**final class**) kann nicht erweitert werden
- Das Schlüsselwort **final**, angewandt auf eine Klasse, verhindert also die Spezialisierung dieser Klasse



- Analog dazu können Finale Methoden (Methoden mit dem Schlüsselwort **final** in der Signatur) in Unterklassen nicht überschrieben werden
- Damit kann die Spezialisierung einer Klasse als ganzes weiterhin erlaubt werden, aber bestimmte Bereiche können explizit geschützt werden
- Alle Methoden einer **final class** sind effektiv auch **final**
- Methoden, die von einem Konstruktor aufgerufen werden, sollten final sein, sonst kann es zu unerwartetem Verhalten kommen

# Beispiel: Finale Klasse

```
public final class LinkedList {  
  
    private LinkedListElement start;  
  
    public Object get(int index) { ... }  
  
    /** Add the given value to the list */  
    public void add(Object value) { ... }  
  
    /** Add the given value to the list or replace it  
     * if it is already in the list and if replace==true */  
    public void add(Object value, boolean replace) { ... }  
  
    public void remove(Object value) { ... }  
  
    public boolean contains(Object value) { ... }  
  
    public void print() { ... }  
  
    private class LinkedListElement { ... }  
}
```

# Beispiel: Finale Methoden

```
public class LinkedList {  
  
    private LinkedListElement start;  
  
    public Object get(int index) { ... }  
  
    /** Add the given value to the list */  
    public final void add(Object value) { ... }  
  
    /** Add the given value to the list or replace it  
     * if it is already in the list and if replace==true */  
    public final void add(Object value, boolean replace) { ... }  
  
    public void remove(Object value) { ... }  
  
    public boolean contains(Object value) { ... }  
  
    public void print() { ... }  
  
    private class LinkedListElement { ... }  
}
```

## Beispiel: Fehler durch Überschreiben (2)

```
public class IntArray {  
  
    protected int[] data;  
  
    public IntArray() {  
        init();  
    }  
  
    public IntArray(int value) {  
        init();  
        data[0] = value;  
    }  
  
    protected void init() {  
        data = new int[1];  
        data[0] = 1;  
    }  
  
}
```

```
public class EmptyIntArray extends IntArray {  
  
    public EmptyIntArray() {  
        super();  
    }  
  
    public EmptyIntArray(int value) {  
        super(value);  
    }  
  
    @Override  
    protected void init() {  
        data = new int[0];  
    }  
  
}
```

## Beispiel: Fehler durch Überschreiben (2)

```
public class IntArray {  
    public IntArray(int value) {  
        init();  
        data[0] = value;  
    }  
    protected void init() {  
        data = new int[1];  
        data[0] = 1;  
    }  
}
```

```
public class EmptyIntArray extends IntArray {  
    @Override  
    protected void init() {  
        data = new int[0];  
    }  
}
```

```
IntArray arr;  
arr = new IntArray();  
arr = new EmptyIntArray();  
arr = new IntArray(1);  
arr = new EmptyIntArray(1); // Exception in thread "main"  
                             // java.lang.ArrayIndexOutOfBoundsException: 0
```

## Beispiel: Fehler durch Überschreiben (2)

```
public class IntArray {  
    public IntArray(int value) {  
        init();  
        data[0] = value;  
    }  
    protected final void init() {  
        data = new int[1];  
        data[0] = 1;  
    }  
}
```

```
public class EmptyIntArray extends IntArray {  
    @Override  
    protected void init() {  
        }  
}
```

Cannot override the final method from **IntArray**

- Alle Platzhalter (Parameter, Variablen, Attribute) können als **final** deklariert werden
- Ihnen kann dann nur einmal ein Wert/eine Referenz zugewiesen werden

```
final int x;  
x = 1;  
x = 2;
```

The final local variable **x** may already have been assigned

```
final int x;  
if (condition) {  
    x = 1;  
} else {  
    x = 0;  
}
```

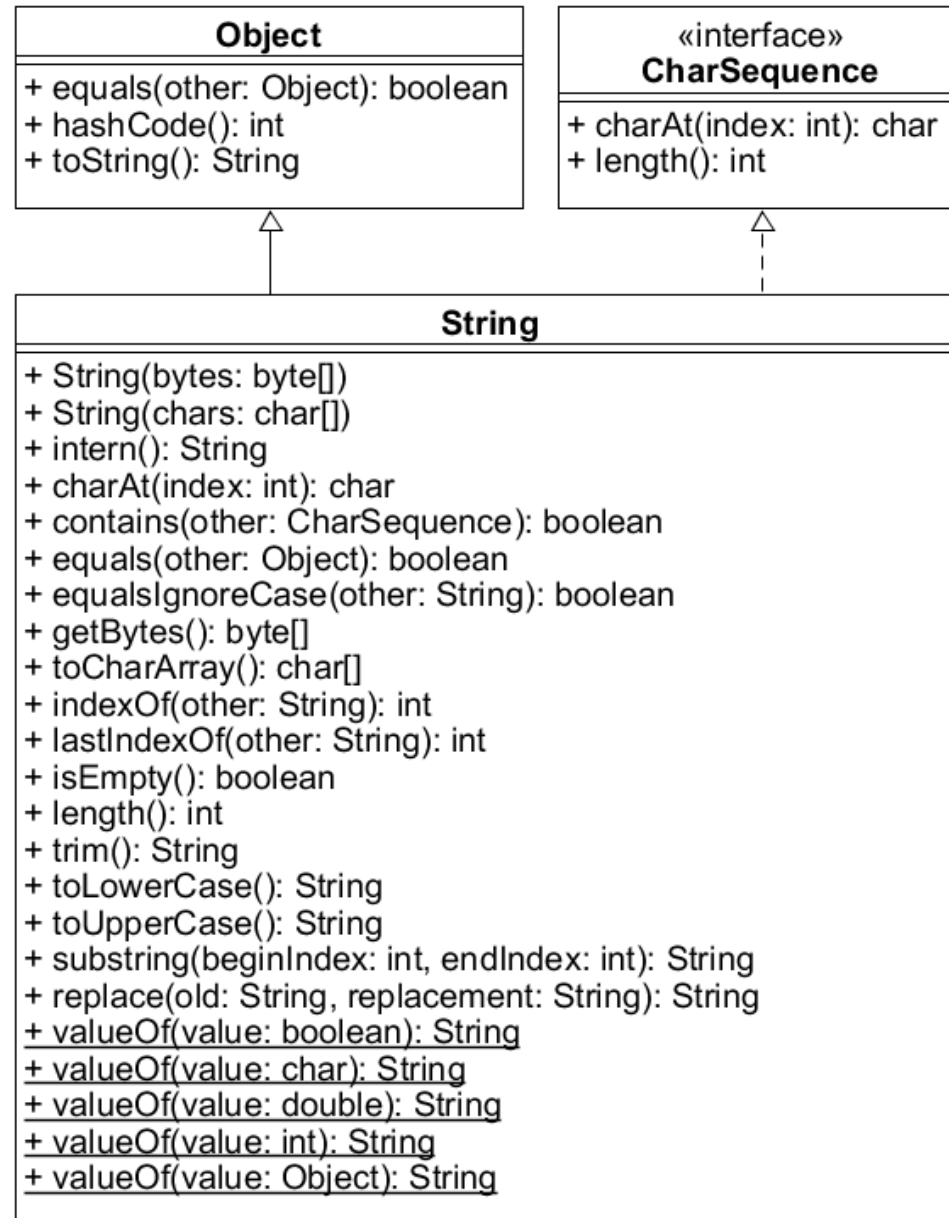
- Auch Objektreferenzen können nicht geändert werden, aber der Objektzustand kann verändert werden

```
final LinkedList list = new LinkedList();  
list.add("x");
```

- **public final class String**
- Strings sind Konstanten, sie können nicht verändert werden
  - Operationen zum Verändern von Strings geben immer einen neuen String zurück, der dann die Veränderung widerspiegelt; der ursprüngliche String bleibt unverändert
  - das hat nichts mit dem Schlüsselwort **final** zu tun
  - **final** wird aber verwendet, um zu verhindern, dass durch Vererbung dieses Verhalten verändert wird



# Überblick: Klasse **String**



# Patterns

- **Patterns**, oder **Entwurfsmuster**, sind konzeptionelle Lösungsschablonen für häufig auftretende Probleme in der Softwareentwicklung
- [Gamma, Helm, Johnson, Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.]
  - Sammlung von 23 typischen Design-Patterns
  - Autoren auch bekannt als “Gang of Four”
- Beispiele im Verlauf der Vorlesung

# Beispiel: Singleton-Pattern

- Klasse, von der es nur eine einzige Instanz geben kann
- Anwendung
  - Printer-Spooler
  - Logger
- Verwendung umstritten

# Singleton-Pattern: UML

Singleton
- <u>instance: Singleton</u>
- Singleton() + <u>getInstance(): Singleton</u>

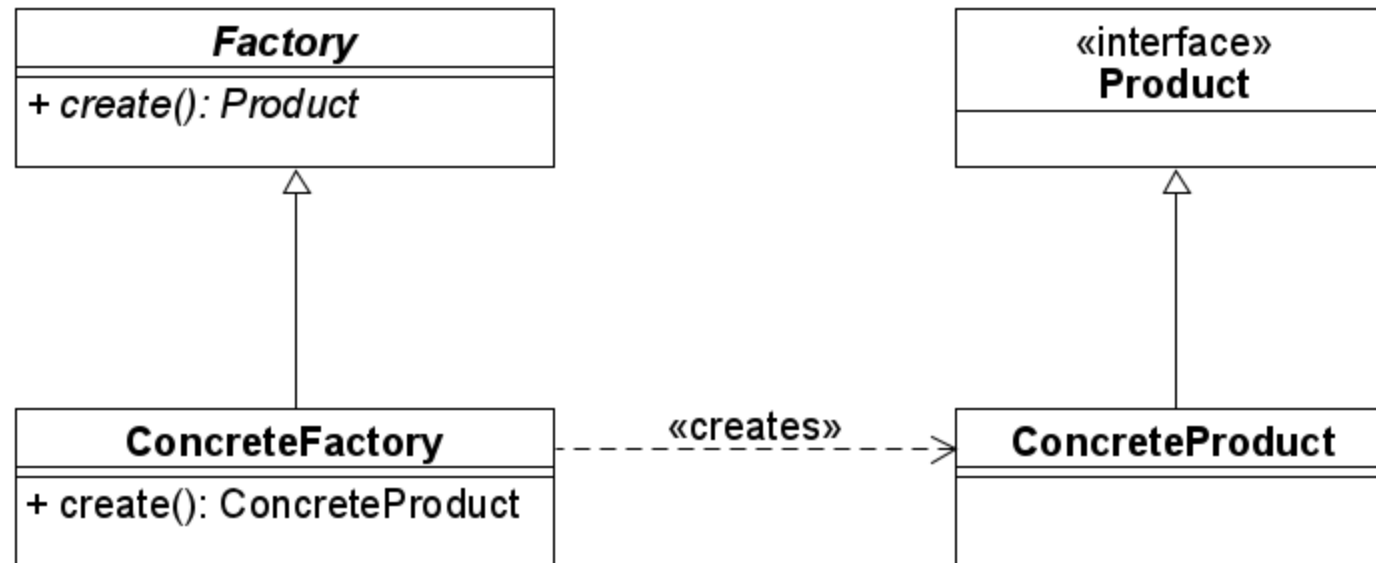
# Singleton-Pattern: Beispiel

```
public class Earth {  
  
    private static Earth instance = null;  
  
    private Earth() { }  
  
    public static Earth getInstance() {  
        if (instance == null) {  
            instance = new Earth();  
        }  
        return instance;  
    }  
}
```

# Factory-Pattern

- Wie kann ich Objekte erzeugen, die zu einer bestimmten Schnittstelle/Oberklasse gehören, ohne die konkrete Implementierung angeben zu müssen?
  - Beispiel: Erzeugen einer Liste ohne anzugeben, ob es eine einfach-verkettete, doppelt-verkettete oder eine Array-Liste sein soll
- Erstellung einer Factory-Klasse mit einer Methode, die Objekte von einem bestimmten Obertyp erzeugt

# Factory-Pattern: UML





# Factory-Pattern: Beispiel

```
public abstract class ListFactory {  
    public abstract List createList();  
}
```

```
public class LinkedListFactory  
    extends ListFactory {
```

```
    @Override  
    public List createList() {  
        return new LinkedList();  
    }
```

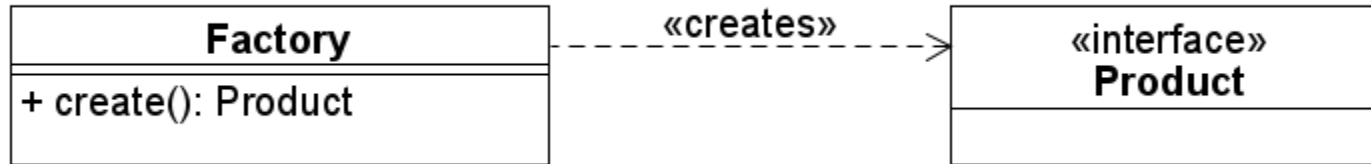
```
public class ArrayListFactory  
    extends ListFactory {  
  
    @Override  
    public List createList() {  
        return new ArrayList();  
    }  
}
```

```
public interface List { }
```

```
public class LinkedList  
    implements List { }
```

```
public class ArrayList  
    implements List { }
```

# Factory-Pattern: Vereinfachung



```
public class ListFactory {  
  
    public List createList() {  
        return new ArrayList();  
    }  
  
}
```

# Pakete und Module

- **Packages** (**Pakete**) dienen dazu, Klassen zu gruppieren
  - nach inhaltlichem Zusammenhang  
z.B. alle Klassen, die thematisch mit dem Studium zusammenhängen (**Student**, **University**, **Exam**, ...)
  - nach technischem Zusammenhang  
z.B. alle Klassen, die mit der Implementierung einer Liste zusammenhängen (**List**, **ListItem**)
- Packages können wiederum in Packages gruppiert werden
  - Paket-Hierarchie
    - z.B. Package **omp** enthält Packages **exercises** und **lecture**
- Benennung:
  - Kleinbuchstaben, Ziffern und Unterstrich, Punkt zur Trennung von Unterpaketen
  - Konvention: umgekehrter Domain-Name (z.B. **de.uni\_oldenburg.inf**)

# Beispiel: Packages

Datei **Domino.java** in **de/uni\_oldenburg/inf/omp/examples**

```
package de.uni_oldenburg.inf.omp.examples;  
  
public class Domino { ... }
```

# Beispiel: Verwendung von Packages

```
package de.uni_oldenburg.inf.omp.game;

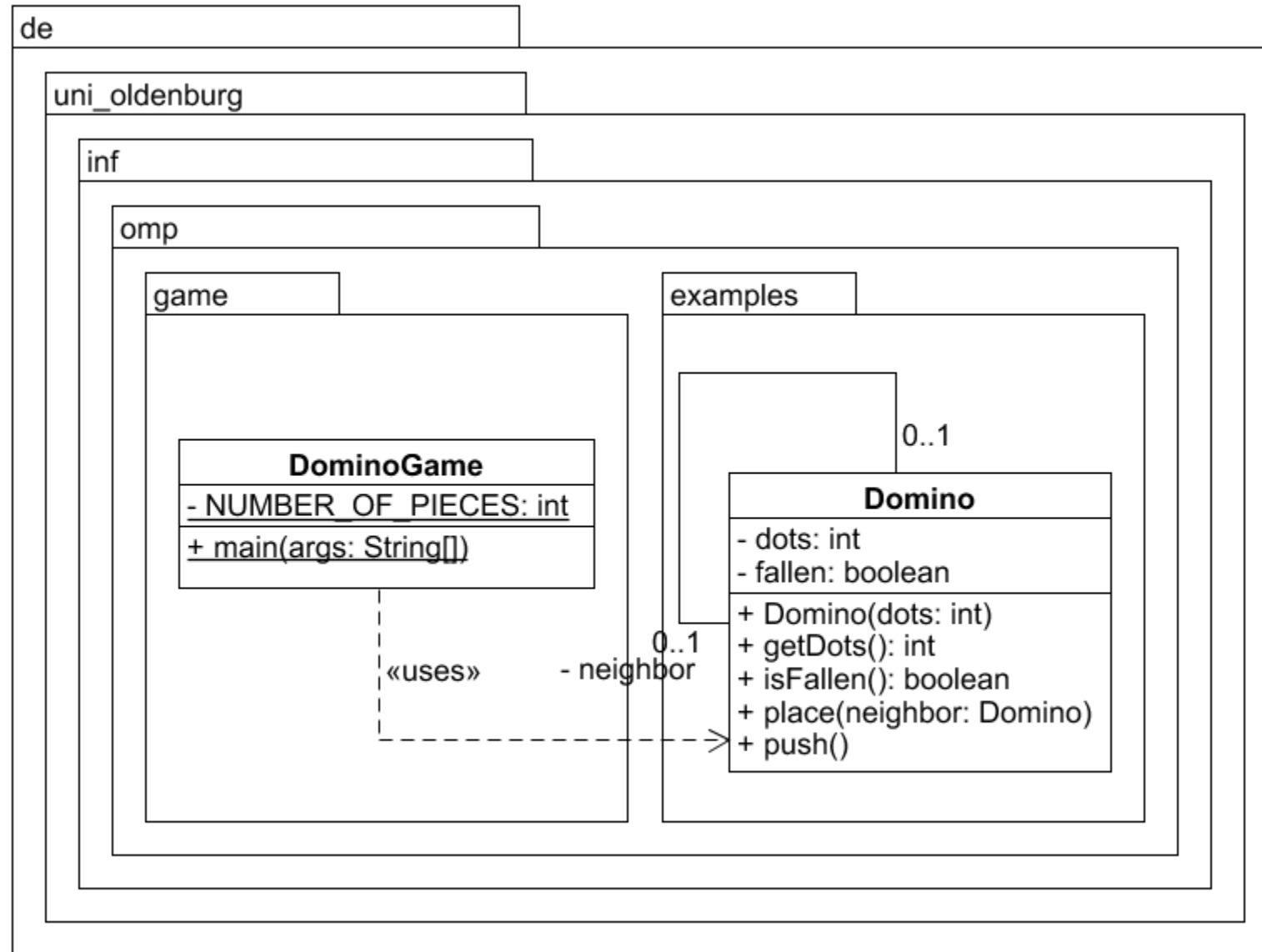
import java.util.Random;
import de.uni_oldenburg.inf.omp.examples.Domino;

public class DominoGame {

    private static final int NUMBER_OF_PIECES = 100;

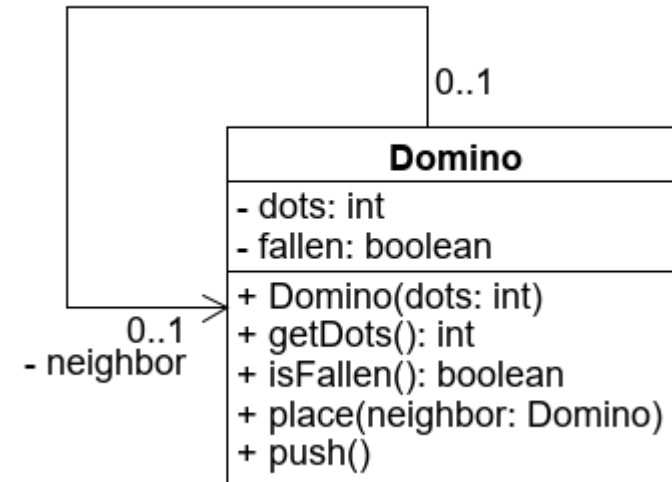
    public static void main(String[] args) {
        Random random = new Random();
        Domino[] pieces = new Domino[NUMBER_OF_PIECES];
        pieces[0] = new Domino(random.nextInt(12) + 1);
        for (int i = 1; i < NUMBER_OF_PIECES; i++) {
            pieces[i] = new Domino(random.nextInt(12) + 1);
            pieces[i - 1].place(pieces[i]);
        }
        pieces[0].push();
    }
}
```

# Packages in the UML



# Zugriffsrechte (Wiederholung)

- Die Sichtbarkeit von Attributen, Konstruktoren und Methoden wird über das erste Zeichen angegeben
- **public:** **+**
  - Zugriff aus beliebigen Klassen
- **protected:** **#**
  - Zugriff nur aus dem gleichen Paket, der gleichen Klasse und aus deren Unterklassen
- *package-private:* **~**
  - Zugriff nur aus dem gleichen Paket
- **private:** **-**
  - Zugriff nur aus der gleichen Klasse





- **Module** sind ab Java 9 eine zusätzliche Möglichkeit, Klassen zu gruppieren
- Sie sind oberhalb der Pakete angesiedelt
  - ein Modul enthält **Pakete**
  - ein Modul enthält die verwendeten **Ressourcen** (Bilder, Konfigurationsdateien, Wörterbücher, ...)
  - ein Modul enthält eine Beschreibung (**Module Descriptor**)
- Modultypen
  - **Systemmodule**: Java SE und JDK
  - **Anwendungsmodule**: Module für jeweils eine konkrete Anwendung/Library
  - **Automatische** und **unbenannte Module**: Module für das Laden von Klassen ohne Modulbeschreibung

# Module Descriptor

```
module omp.module {
```

```
}
```

Beschreibt ein Modul mit dem Namen **omp.module**.

```
module omp.module {  
  
    requires pda.module;  
  
}
```

Definiert eine Abhängigkeit zum Modul **pda.module**.

Dieses Modul muss sowohl zur Compile-Zeit als auch zur Laufzeit verfügbar sein.

Ohne das **pda.module** kann das **omp.module** nicht verwendet werden.

# Module Descriptor

```
module omp.module {  
  
    requires pda.module;  
  
    requires static powerpoint.module;  
  
}
```

Definiert eine Compile-Zeit Abhängigkeit zum Modul **powerpoint.module**.

Dieses Modul muss zur Compile-Zeit verfügbar sein.

Ohne das **powerpoint.module** kann das **omp.module** nicht erstellt, aber verwendet werden.

```
module omp.module {  
    requires pda.module;  
    requires static powerpoint.module;  
    requires transitive umlet.module;  
}
```

Definiert eine transitive Abhängigkeit zum Modul **umlet.module**.

Wenn das Modul **st1.module** eine Abhängigkeit auf das Modul **omp.module** definiert, dann ist **st1.module** auch automatisch von **umlet.module** abhängig.

```
module omp.module {  
  
    requires pda.module;  
  
    requires static powerpoint.module;  
  
    requires transitive umlet.module;  
  
    exports de.uni_oldenburg.inf.omp;  
  
}
```

Macht das Paket **de.uni\_oldenburg.inf.omp** für die Außenwelt sichtbar.

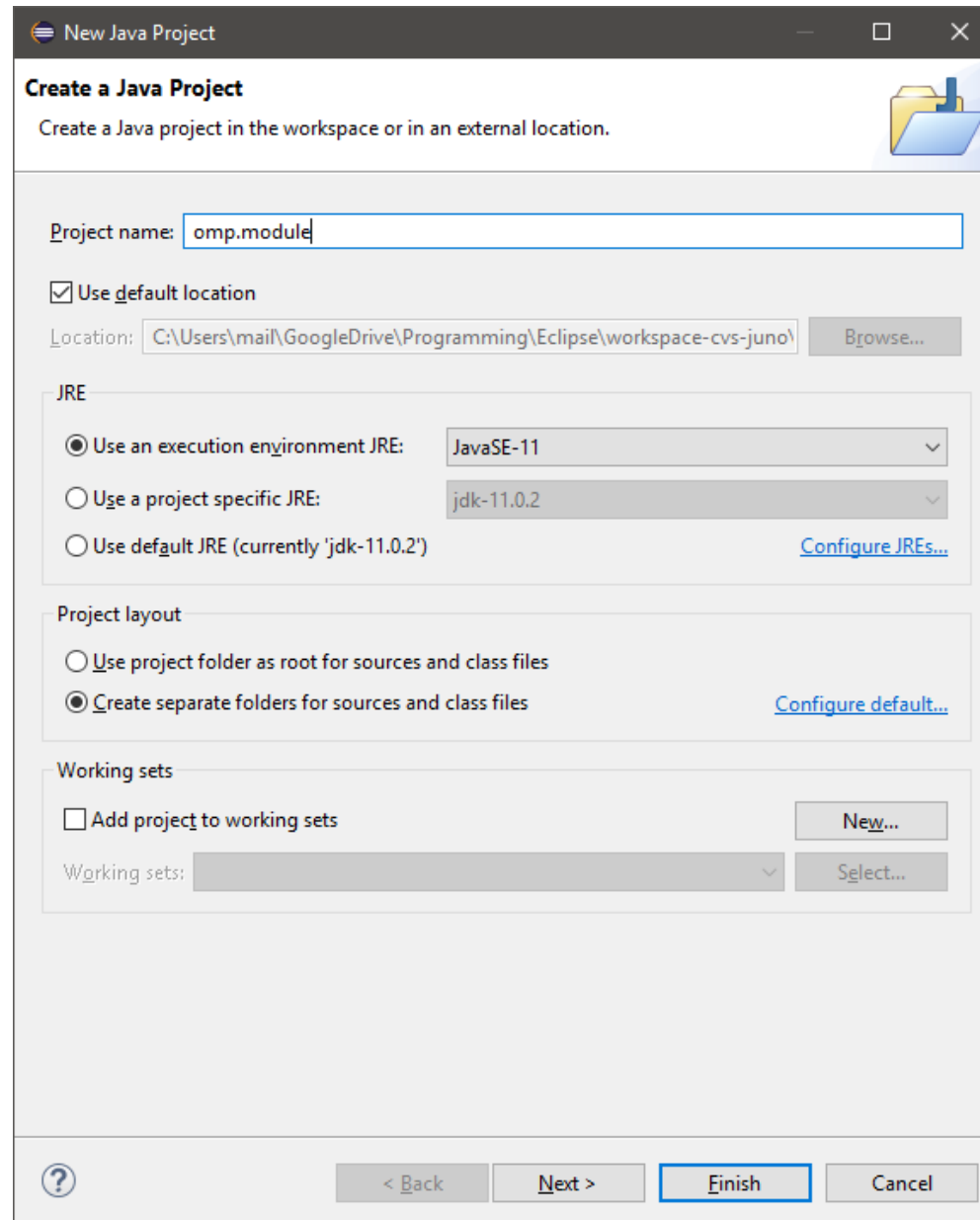
Standardmäßig sind alle Pakete in einem Modul für die Außenwelt **unsichtbar**. Dies ist eine konsequente Weiterführung des **Kapselungs-Prinzips**.

# Module Descriptor

```
module omp.module {  
  
    requires pda.module;  
  
    requires static powerpoint.module;  
  
    requires transitive umlet.module;  
  
    exports de.uni_oldenburg.inf.omp;  
  
}
```

Speichern der Datei als **module-info.java** im Wurzel-Verzeichnis der Pakete.

# Module in Eclipse



The screenshot shows the 'New Java Project' dialog box in the Eclipse IDE. The dialog has a title bar 'New Java Project' and a subtitle 'Create a Java Project'. Below the subtitle is the instruction 'Create a Java project in the workspace or in an external location.' and a folder icon.

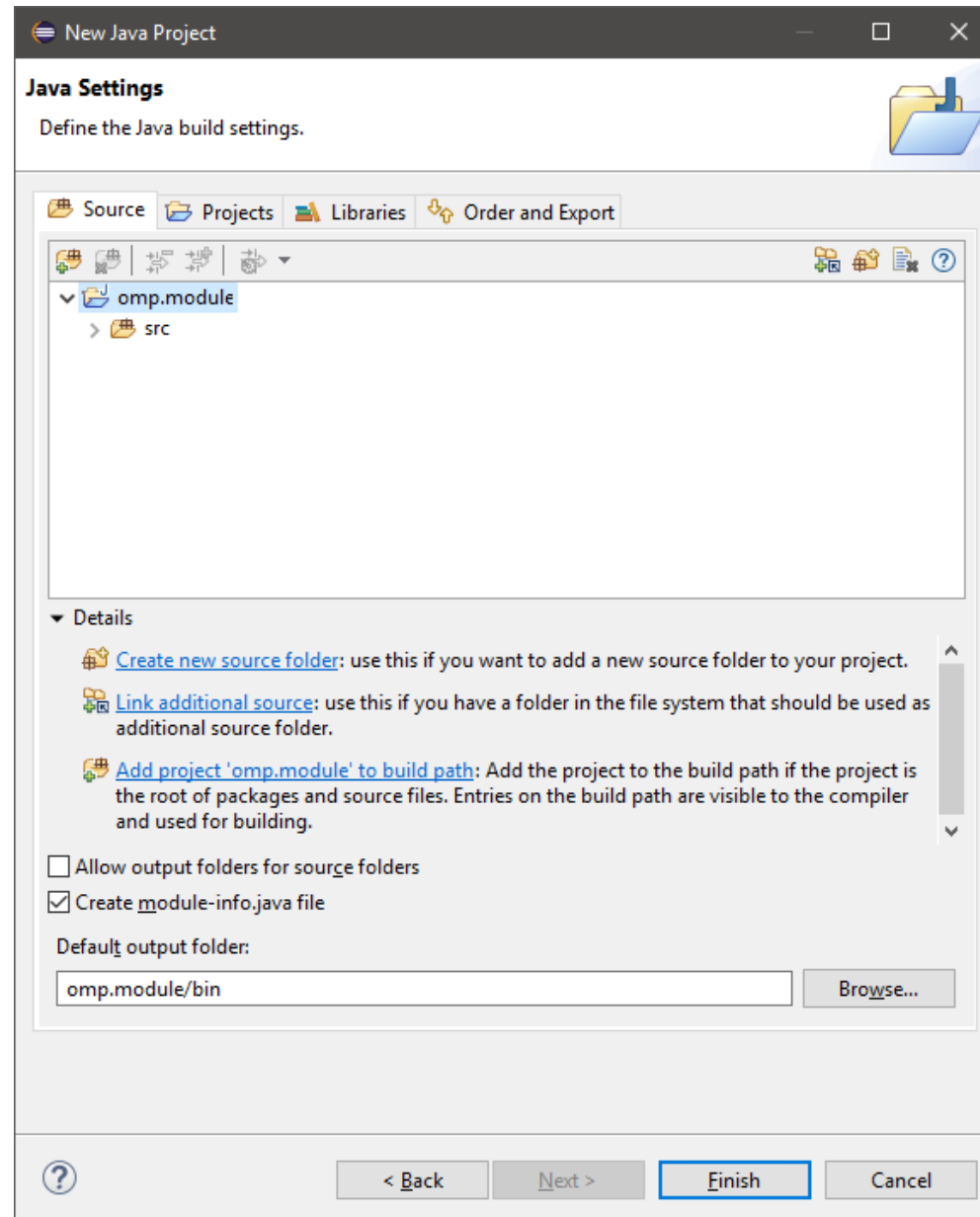
The main content area is divided into several sections:

- Project name:** A text field containing 'omp.module'.
- Use default location:** A checked checkbox.
- Location:** A text field showing 'C:\Users\mail\GoogleDrive\Programming\Eclipse\workspace-cvs-juno\'. To the right is a 'Browse...' button.
- JRE:** A section with three radio buttons:
  - ☒ Use an execution environment JRE: A dropdown menu showing 'JavaSE-11'.
  - ☐ Use a project specific JRE: A dropdown menu showing 'jdk-11.0.2'.
  - ☐ Use default JRE (currently 'jdk-11.0.2').To the right of the third option is a link 'Configure JREs...'.
- Project layout:** A section with two radio buttons:
  - ☐ Use project folder as root for sources and class files.
  - ☒ Create separate folders for sources and class files.To the right of the second option is a link 'Configure default...'.
- Working sets:** A section with a checkbox 'Add project to working sets' and a 'New...' button. Below this is a 'Working sets:' dropdown menu and a 'Select...' button.

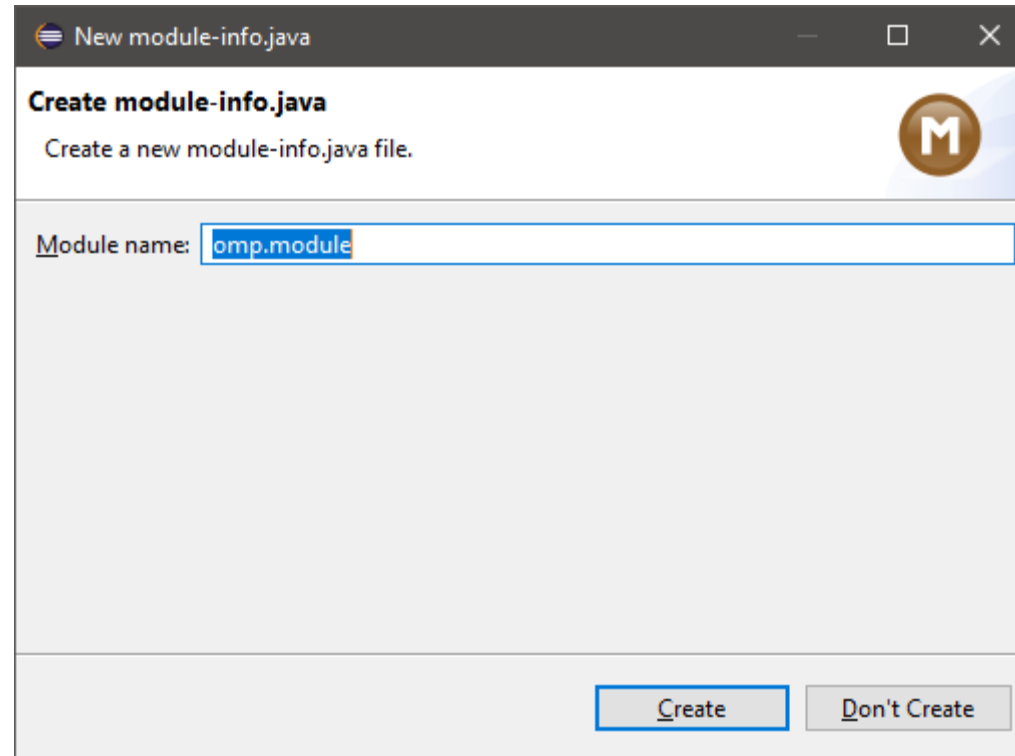
At the bottom of the dialog are navigation buttons: a help icon (?), '< Back', 'Next >', 'Finish' (highlighted with a blue border), and 'Cancel'.



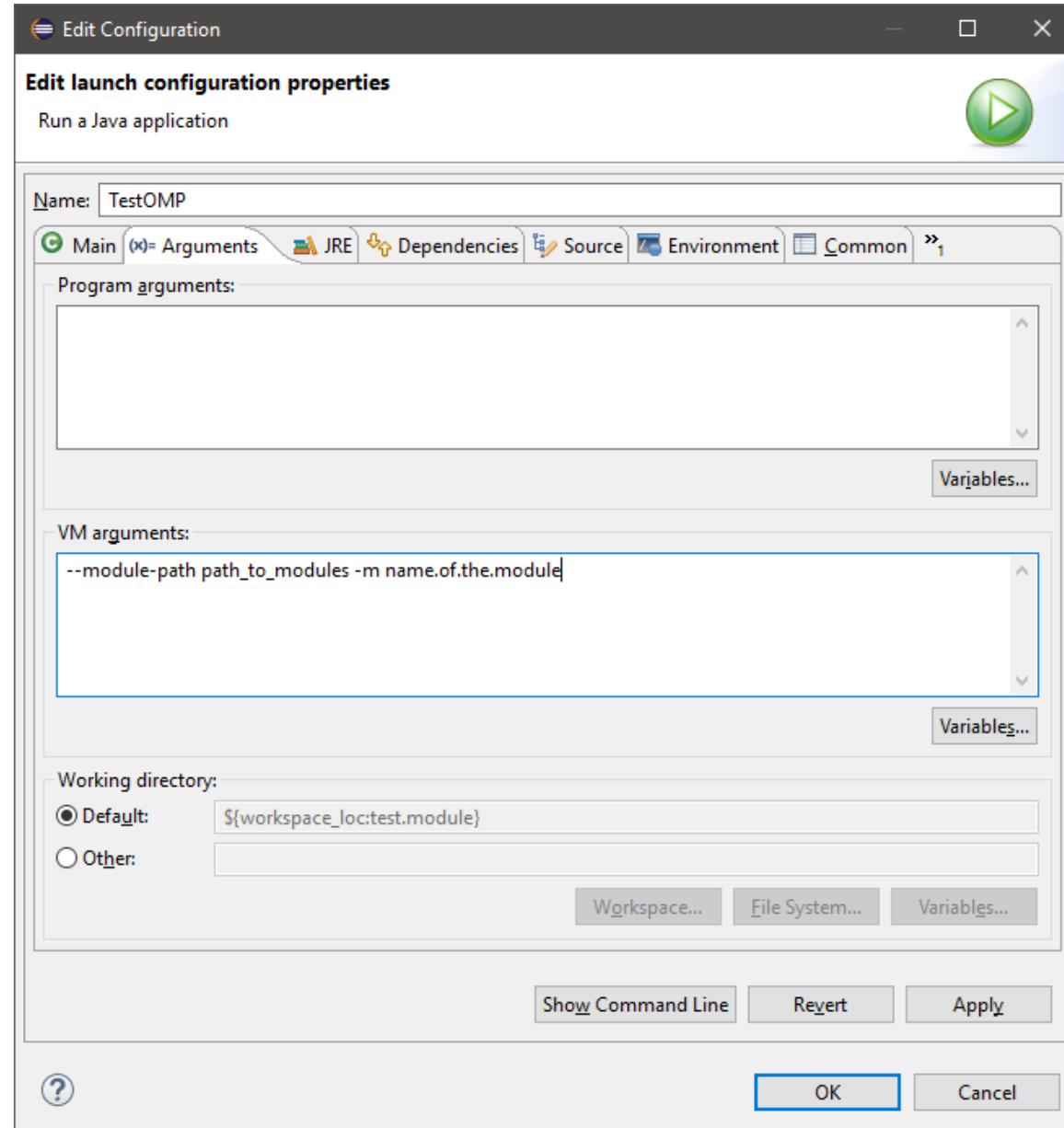
# Module in Eclipse



# Module in Eclipse



# Module in Eclipse



- Modellierung und Programmierung
- Verschachtelte Klassen
  - Innere Klassen
  - Statische Klassen
- Finale Klassen, Methoden und Platzhalter
- Patterns
- Packages
- Module