

Datenverarbeitung

Teil des Moduls 5CS-DPDL-20

Prof. Dr. Deweß

Thema 3



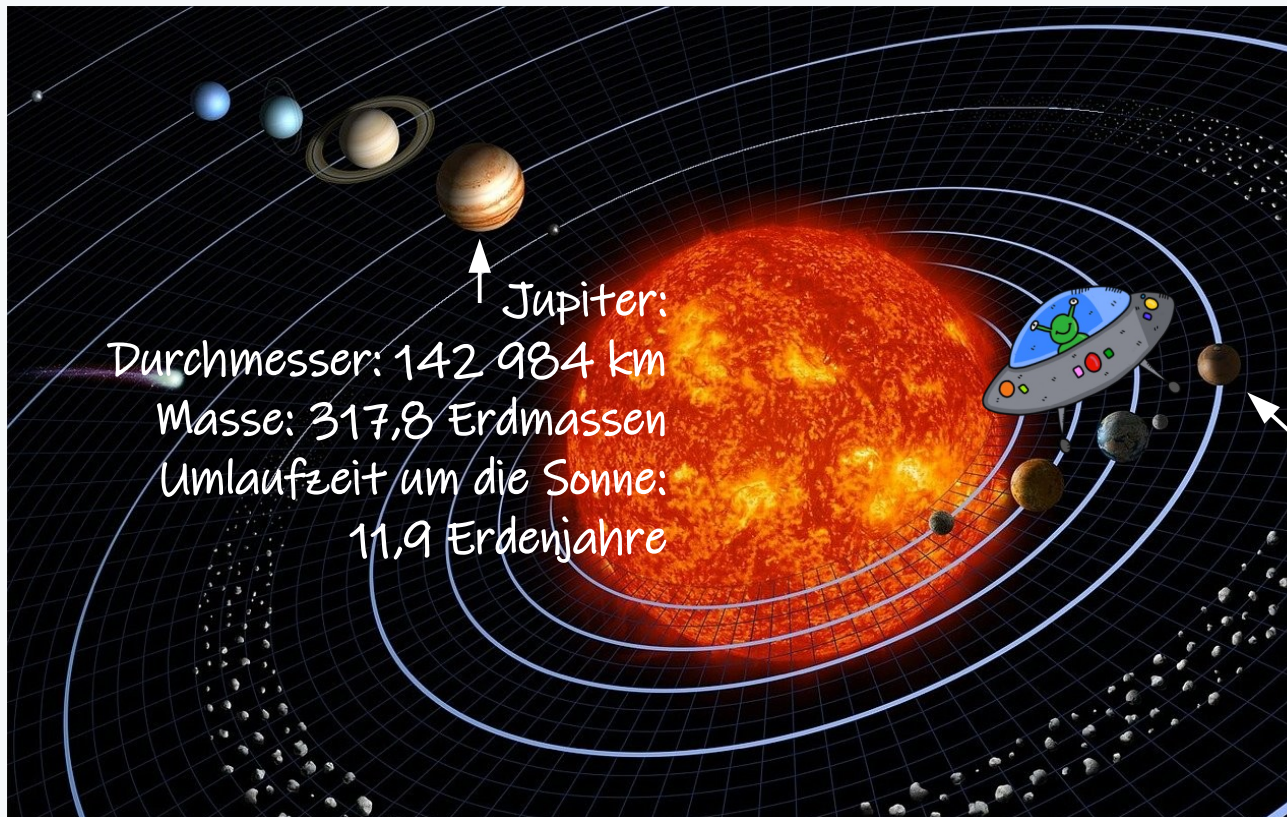
3

Neuere Typen und andere fortschrittliche Dinge

- „enum“: Aufzählungstyp
- „record“: Datenhalter
- „default“ und „static“: Implementierte Methoden in Schnittstellen
- „sealed“ und „non-sealed“: Einschränkung von Klassen-Benutzbarkeit ergänzend zu den klassischen Zugriffsmodifizierern

enum – „Aufzählungstyp“

- Spezialklasse „Enum“ für die Festlegung einer festen Menge benannter, konstanter Objekte (z.B. die Planeten unseres Sonnensystems) und zugehöriger Methoden
- benutzbar über Schlüsselwort „enum“



viel mehr als „enums“
vieler anderer Sprachen,
die teilweise nur die
Zuordnung einer Zahl
zu einem Namen
übernehmen

Mars:
Durchmesser: 6 794 km
Masse: 0,107 Erdmassen
Umlaufzeit um die Sonne:
687 Erdentage

enum – „Aufzählungstyp“

- einfachster Fall: Sammlung von Werten
- Festlegung über Schlüsselwort „enum“
- intern als Klasse erzeugt:
 - entweder als „Top-Level-Klasse“ oder
 - als (implizit statisch) eingebettete Klasse
- derartig erzeugte „enum“-Klassen sind entweder implizit „final“ (so wie obiges Beispiel) oder implizit „sealed“ (Schlüsselwort lernen wir später kennen)
- kann Zugriffsmodifizierer „private“, „public“ oder „protected“ haben, aber nicht die Modifizierer „abstract“, „final“, „sealed“ oder „non-sealed“
- die Superklasse einer enum-Klasse „T“ ist „Enum<T>“, auch wenn die enum-Deklaration keine Erweiterungsklauseln („extends“/„implements“) vorsieht und daher derartige Spezifikationen nicht explizit deklariert werden können
- die einzigen Instanzen einer „enum“-Klasse sind die, die durch die „enum-Konstanten“ definiert sind; eine weitere Instanzierung ist nicht möglich

```
package de.baleipzig.classes;  
  
public enum Planet {  
    MERCURY,  
    VENUS,  
    //...  
    NEPTUNE;  
}
```

enum – „Aufzählungstyp“

```
package de.baleipzig.classes;

public enum Planet {
    MERCURY (4879, 0.055),
    VENUS (12104, 0.815),
    //...
    NEPTUNE (49528, 17.1);
    private final int diameter; // in km
    private final double mass; // in Earth masses
    Planet(int aDiameter, double aMass) {
        diameter = aDiameter;
        mass = aMass;
    }
    public int getDiameter() { return diameter; }
    public double getMass() { return mass; }
    // surface area (km2)
    public double surfaceArea() {
        return Math.PI * diameter * diameter;
    }
    // Kepler's Constant (s3 / m3)
    public static final double K = 2.97E-19;
}
```

Schlüsselwort „enum“

(Klasse wird intern vom Compiler in eine Erweiterung „Enum“-Klasse übersetzt, wobei das anders nicht vorgesehen ist)

Namen der konstanten Objekte
und ggf. Eigenschaftswerte zur
Initialisierung

ggf. Instanzvariable

ggf. Konstruktor

(wird automatisch „private“ erzeugt
[daher keine Objekterzeugung mit
„new“ von außen möglich] und erhält
automatisch noch Objektnamen und
Objektindex [beginnt bei 0] als
weitere Parameter)

ggf. weitere Konstanten

ggf. Methoden

enum – „Aufzählungstyp“

```
package de.baleipzig.classes;

public enum Planet {
    MERCURY (4879, 0.055),
    VENUS   (12104, 0.815),
    //...
    NEPTUNE (49528, 17.1);
    private final int diameter; // in km
    private final double mass; // in Earth masses
    Planet(int aDiameter, double aMass) {
        diameter = aDiameter;
        mass = aMass;
    }
    public int getDiameter() { return diameter; }
    public double getMass() { return mass; }
    // surface area (km2)
    public double surfaceArea() {
        return Math.PI * diameter * diameter;
    }
    // Kepler's Constant (s3 / m3)
    public static final double K = 2.97E-19;
}
```

Aufgabe 1:

Fügen Sie mindestens einen weiteren Planeten unseres Sonnensystems hinzu.

Aufgabe 2:

Erweitern Sie unser „enum“ um die weitere Eigenschaft „Umlaufzeit“ um die Sonne in Erdtagen.

Zusatz:

Erweitern Sie unser „enum“ um eine zusätzliche Methode zur Berechnung des jeweiligen Planeten-Volumens

enum – „Aufzählungstyp“

Benutzung von „enums“

- als Variable: Planet p
- als Konstante: Planet.VENUS
- innerhalb von switch-Ausdrücken
- innerhalb von Schleifen (siehe später)
- ...

Außerdem möglich:

- Nutzung von Methoden der Klasse „Enum“ und geerbter bzw. implementierter Methoden dieser Klasse
- Nutzung spezieller vom Compiler erzeugter Methoden für die spezielle „enum“-Klasse

```
package de.baleipzig.classes;

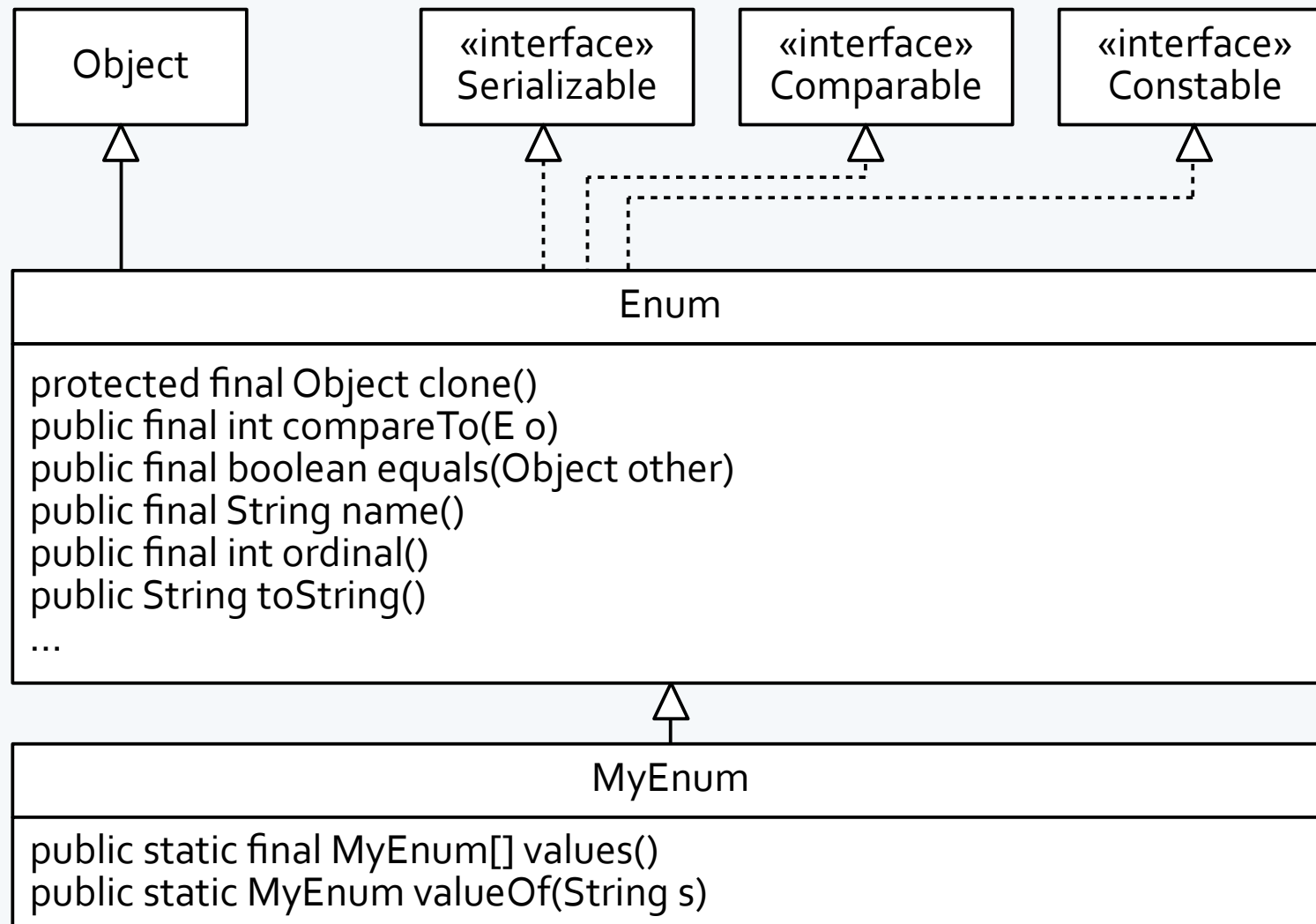
public class TestClass {

    public static boolean isBig(Planet p) {
        boolean b = false;
        switch (p) {
            case MERCURY:
            case VENUS:
                b = false;
                break;
            case NEPTUNE:
                b = true;
                break;
            default:
                b = false;
        }
        return b;
    }

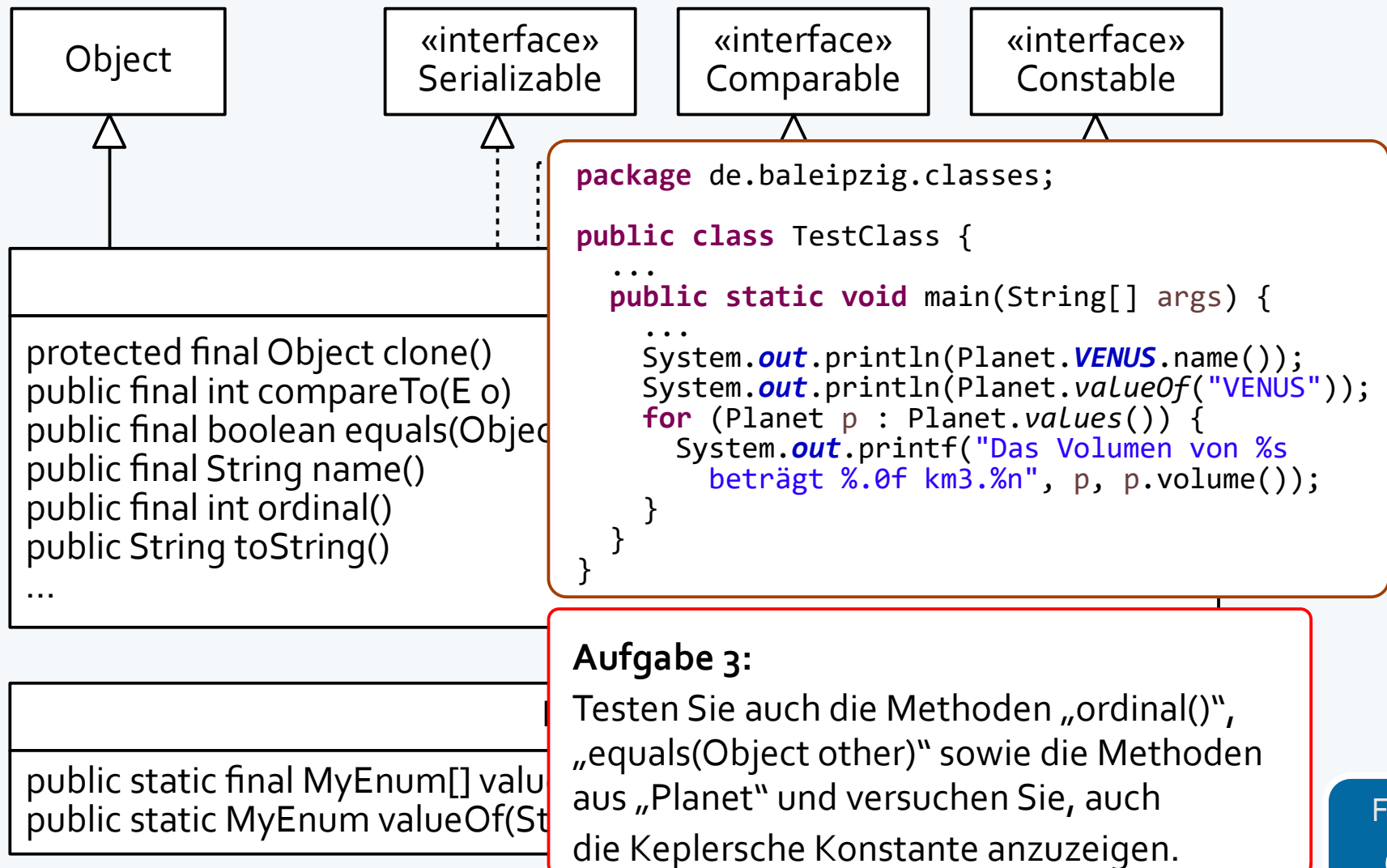
    public static void main(String[] args) {
        Planet p1 = Planet.VENUS;
        System.out.println(isBig(p1));
    }
}
```

neue Syntax als Anmerkung:
`boolean b = switch (p) {
 case MERCURY, VENUS -> false;
 case NEPTUNE -> true;
 default -> false;
};`

enum – „Aufzählungstyp“



enum – „Aufzählungstyp“




enum – „Aufzählungstyp“

```
package de.baleipzig.classes;

public class TestClass {
    ...
    public static void main(String[] args) {
        ...
        System.out.println(Planet.K);
        System.out.println(Planet.VENUS.ordinal());
        Planet p1 = Planet.VENUS;
        Planet p2 = Planet.NEPTUNE;
        Planet p3 = Planet.VENUS;
        System.out.println(p1.equals(p3));
        System.out.println(Planet.VENUS.getDiameter());
        System.out.println(Planet.VENUS.getMass());
        System.out.println(Planet.VENUS.surfaceArea());
    }
}
```

Namen und
Webseiten können
Sie im Internet
recherchieren



Aufgabe 4:

Schreiben Sie ein eigenes „enum“ namens „RIR“, welches für die „Regional Internet Registries“ die Objekte mit den üblichen Namen „AFRINIC“, „ARIN“, „APNIC“, „LACNIC“ und „RIPENCC“ jeweils mit dem ausführlichem Namen und der Webseite der Organisation als (private) Eigenschaften zur Verfügung stellt. Implementieren Sie nötige Getter.

Testen Sie Ihre Klasse, indem Sie sich in Ihrer Testklasse die Webseite von „AFRINIC“ anzeigen lassen.

enum – „Aufzählungstyp“

```
package de.baleipzig.classes;

public enum RIR {
    AFRINIC("African Network Information Centre", "www.afrinic.net"),
    ARIN("American Registry for Internet Numbers", "www.arin.net"),
    APNIC("Asia-Pacific Network Information Centre", "www.apnic.net"),
    LACNIC("Latin America and Caribbean Network Information Centre", "www.lacnic.net"),
    RIPENCC("Réseaux IP Européens Network Coordination Centre", "www.ripe.net");

    private final String name;
    private final String website;

    RIR(String aName, String aWebsite) {
        name = aName;
        website = aWebsite;
    }

    public String getName() { return name; }
    public String getWebsite() { return website; }
}
```

```
System.out.println(RIR.AFRINIC.getWebsite());
```



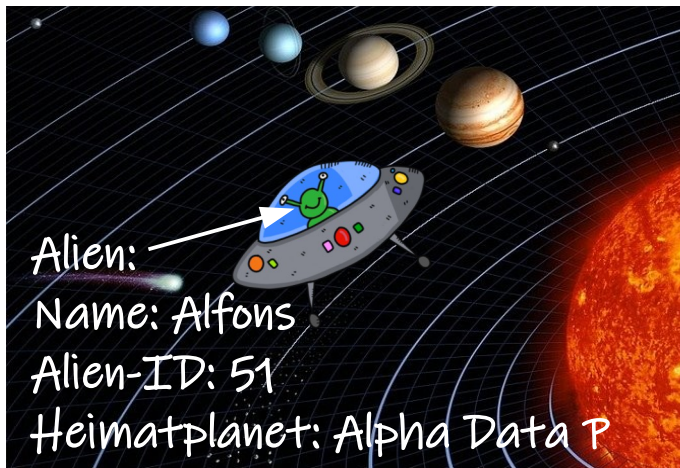
3

Neuere Typen und andere fortschrittliche Dinge

- „enum“: Aufzählungstyp
- „record“: Datenhalter
- „default“ und „static“: Implementierte Methoden in Schnittstellen
- „sealed“ und „non-sealed“: Einschränkung von Klassen-Benutzbarkeit ergänzend zu den klassischen Zugriffsmodifizierern

record – „Datenhalter“

- Spezialklassen als „Datenhalter“ – als Verbesserung für Klassen, die einfach nur unveränderliche Daten, z.B. aus Datenbankabfragen, und zugehörige Konstruktoren, Getter und Setter enthalten
- benutzbar über Identifikator „record“

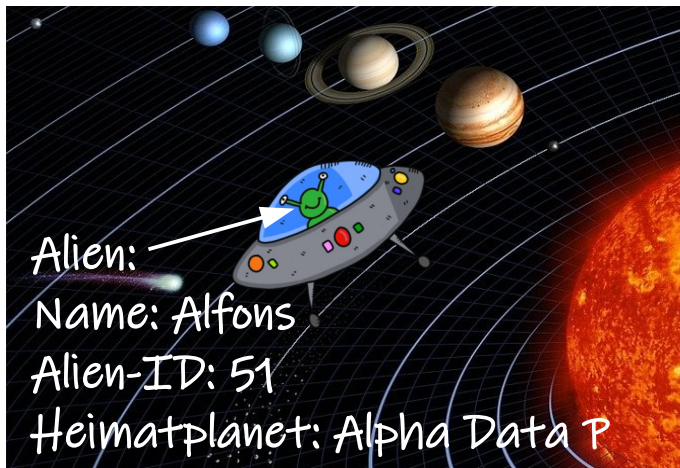


Typisch für bisherige Datenhalter-Klassen

- „privat final“-Instanzvariablen
 - Getter für jede Instanzvariable
 - „public“-Konstruktor zur Wertefestlegung für jede Instanzvariable
 - „toString“-Methode zur Datenausgabe
 - „equals“- und „hashCode“-Methode derart, dass Objekte mit gleichen Datenwerten gleich sind
-
- **also bisher:** viel „Standardcode“ (Aufwand); fehleranfällig bei Erweiterungen (Wenn der Alien noch eine Adresse auf der Erde bekommt, muss das bei „toString“ usw. auch manuell „nachimplementiert“ werden ...); eigentlicher „Datenhalter“-Zweck nicht offensichtlich
 - **Lösung:** stattdessen neuen „record“ benutzen

record – „Datenhalter“

- Beispiel „Alien-record“ als Datenhalter für „Alien“-Daten:
`public record Alien(String name, int alienID, String homePlanet) {}`
- „Alien“ wird wie bei Klassen üblich erzeugt:
- `Alien firstAlien = new Alien("Alfons", 51, "Alpha Data P");`



Typisch für „record“:

- „privat final“-Instanzvariablen („name“, „alienID“ und „homePlanet“) werden erzeugt
- Getter für jede Instanzvariable erzeugt
- „public“-Konstruktor zur Wertefestlegung für jede Instanzvariable erzeugt
- passende „toString“-, „equals“- und „hashCode“-Methode erzeugt

Aufgabe 5: Erstellen Sie einen „Alien-record“ und erzeugen Sie in Ihrer Testklasse zwei Alien mit identischen Werten. Testen Sie die automatisch erzeugte „toString“-Methode durch Ausgabe Ihrer Alien auf der Konsole und die automatisch erzeugte „equals“-Methode, indem Sie für ihren ersten Alien durch Aufruf der Methode mit dem zweiten Alien als Parameter prüfen, inwieweit beide Alien vergleichbare Werte haben.

record – „Datenhalter“- Lösung 5

```
package de.baleipzig.classes;  
  
public record Alien(String name, int alienID, String homePlanet) {}
```

```
public class TestClass {  
  
    public static void main(String[] args) {  
        Alien firstAlien = new Alien("Alfons", 51, "Alpha Data P");  
        Alien secondAlien = new Alien("Alfons", 51, "Alpha Data P");  
        System.out.println(firstAlien);  
        System.out.println(secondAlien);  
        System.out.println(firstAlien.equals(secondAlien));  
    }  
}
```

record – „Datenhalter“

„record“: Besonderheiten und Gemeinsamkeiten mit einer „normale Klasse“

besonders

- kann keine andere Klasse erweitern
- nur die in der Komponentenliste angegebenen Instanzvariablen sind erlaubt; alle anderen müssen „static“ sein
- ist, ebenso wie alle seine Komponenten, implizit „final“
- kann nicht „abstract“ sein

gemeinsam mit anderen Klassen

- können eingebettet in einer Klasse deklariert werden (sind dann allerdings implizit „static“)
- generische „records“ sind möglich
- können Schnittstellen implementieren
- Instanzen werden mit „new“ erzeugt
- statische Variablen sind möglich
- Konstruktoren und Methoden sind möglich (solange der „record“-Gedanke dabei nicht verletzt wird)
- ...

record – „Datenhalter“

„record“ – Beispiel für eigene Konstruktoren:

```
package de.baleipzig.classes;

import java.util.Objects;

public record Alien(String name, int alienID, String homePlanet) {
    public Alien {
        Objects.requireNonNull(name);
        Objects.requireNonNull(homePlanet);
    }

    public Alien(String name, int alienID) {
        this(name, alienID, UNKNOWN_HOME_PLANET);
    }

    public static String UNKNOWN_HOME_PLANET = "Unknown";
}
```

Aufgabe 6: Schreiben Sie im „Alien-record“ einen Konstruktor, der für die Erzeugung von Aliens ohne bisher vergebene Alien-ID genutzt werden kann. Legen Sie dafür eine ganzzahlige, statische Variable mit Initialwert 10 000 fest, die als Ersatz-ID genutzt und bei jeder Instanzerzeugung um eins erhöht wird. Testen Sie Ihren Konstruktor.

record – „Datenhalter“- Lösung 6

```
package de.baleipzig.classes;

public record Alien(String name, int alienID, String homePlanet) {

    public Alien(String name, String homePlanet) {
        this(name, ARTIFICIAL_ID, homePlanet);
        ARTIFICIAL_ID++;
    }

    public static int ARTIFICIAL_ID = 10000;
}
```

```
public class TestClass {

    public static void main(String[] args) {
        ...
        Alien thirdAlien = new Alien("Balfons", "Alpha Data P");
        System.out.println(thirdAlien);
        Alien fourthAlien = new Alien("Calfons", "Alpha Data P");
        System.out.println(fourthAlien);
    }
}
```

Aufgabe 7: Schreiben Sie im „Alien-record“ eine Methode, die eine Begrüßungsnachricht vom Alien mit seinem Namen und seinem Heimatplanet ausgibt. Testen Sie diese.

record – „Datenhalter“- Lösung 7

```
package de.baleipzig.classes;

public record Alien(String name, int alienID, String homePlanet) {
    ...

    public void greetEarth() {
        System.out.println("Hallo, hier ist " + name + " von " + homePlanet + ".");
    }
}
```

```
public class TestClass {

    public static void main(String[] args) {
        ...
        Alien thirdAlien = new Alien("Balfons", "Alpha Data P");
        thirdAlien.greetEarth();
    }
}
```



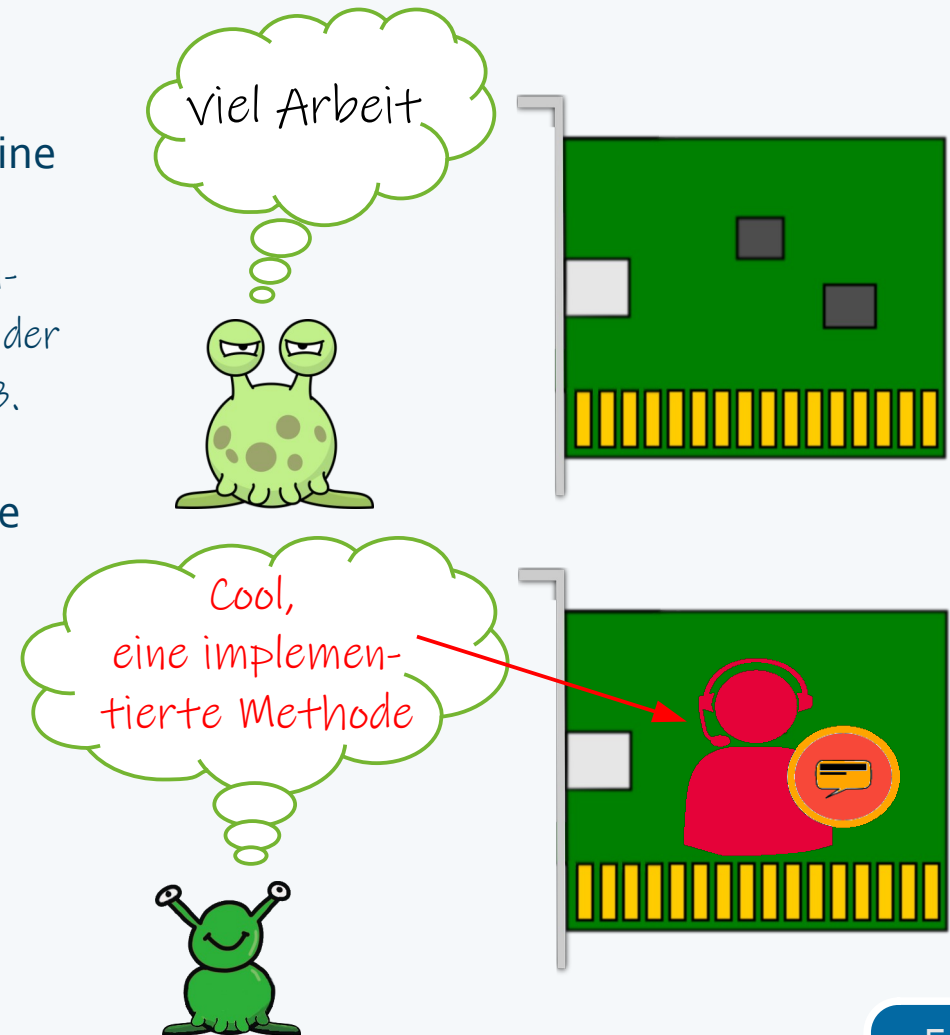
3

Neuere Typen und andere fortschrittliche Dinge

- „enum“: Aufzählungstyp
- „record“: Datenhalter
- „default“ und „static“: Implementierte Methoden in Schnittstellen
- „sealed“ und „non-sealed“: Einschränkung von Klassen-Benutzbarkeit ergänzend zu den klassischen Zugriffsmodifizierern

„default“ und „static“: Implementierte Methoden in Schnittstellen

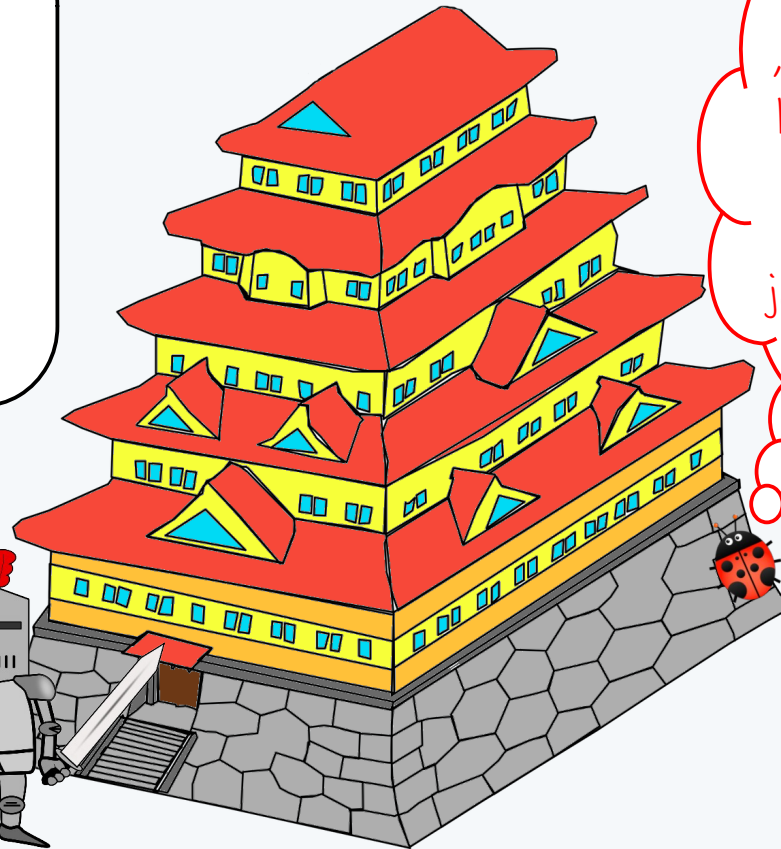
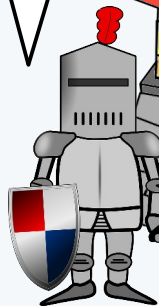
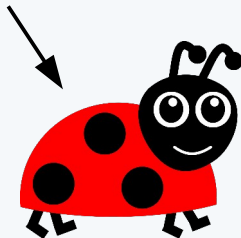
- **bis Java 8**
eine Schnittstelle war so ähnlich wie eine rein abstrakte Klasse (diente aber im Unterschied dazu meist nicht der generalisierten Objektcharakterisierung, sondern der Abbildung von speziellen Fähigkeiten [z.B. „zeichenbar“, „als Modell nutzbar“, ...]), hatte also nur abstrakte Methoden, die das „Was“ ein Objekt kann, aber nicht das „Wie“ abgebildet haben
- **ab Java 8**
automatisch mit Zugriffsmodifizierer „public“ versehene „default“ und „statische“ **Methoden mit Implementierung in Schnittstellen**



“default” und „static“: Implementierte Methoden in Schnittstellen

Du kommst hier
nicht rein!
Wir wollen in
Java keine
„Mehrfach-
vererbung“!

scheinbar gefährliche
„Mehrfachvererbung“

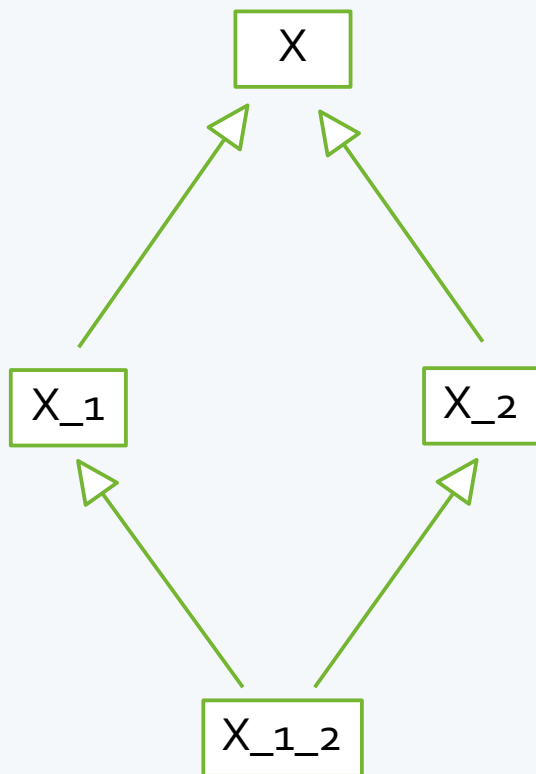


Bei den
„Schnittstellen“
hat jemand das
Fenster
aufgemacht –
jetzt komme ich
rein.

In Java gibt es **bei Klassen** keine Mehrfachvererbung!

“default” und „static“: Implementierte Methoden in Schnittstellen

Warum ist „Mehrfachvererbung“ an sich problematisch?

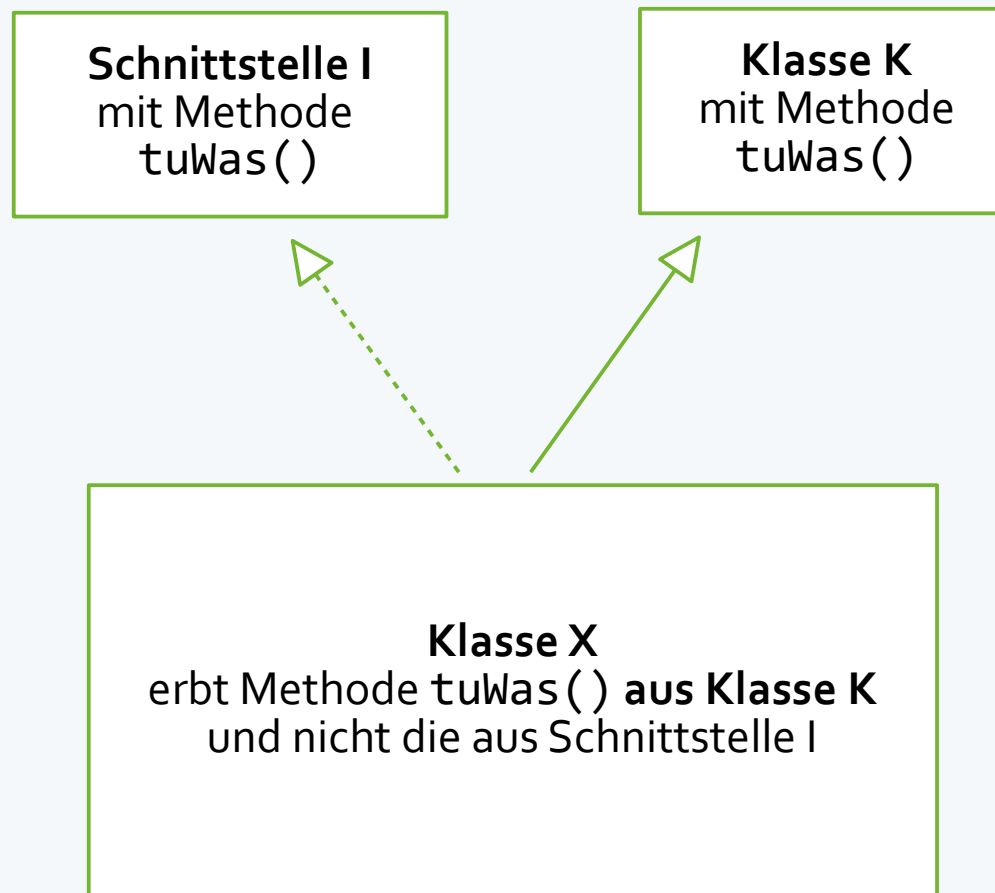


- Wie oft sind Mitglieder von X in X_1_2 enthalten, falls der X-Teil, den X_1_2 über X_1 bzw. X_2 erbt, verschieden ist?
- Bzw. welche Mitglieder sind dann enthalten?

Da es in Java keine Mehrfachvererbung von Klassen gibt und Schnittstellen keine nicht-statischen, nicht-finalen Instanzvariablen haben, stellt sich diese Frage in Java zum Glück nur noch für Methoden und die Situation ist nicht ganz so gefährlich.

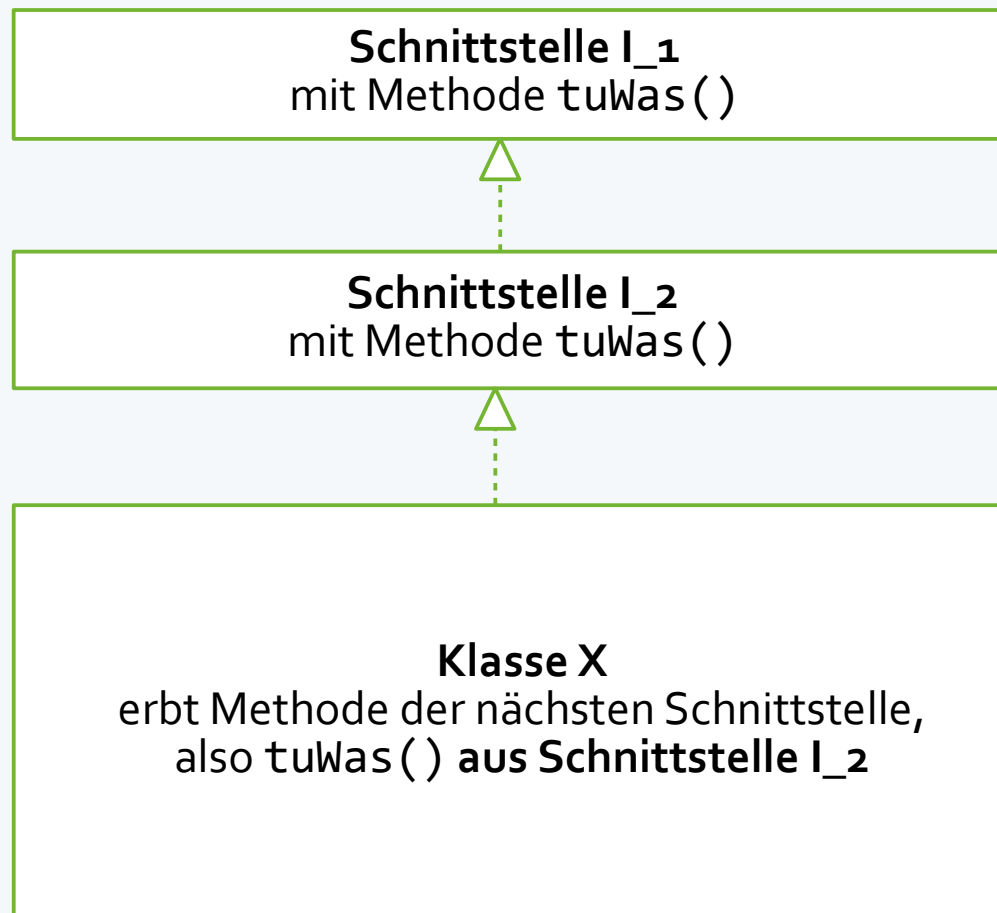
„default“ und „static“: Implementierte Methoden in Schnittstellen

„Mehrfachvererbung“ von Methoden – Regel für Fall 1



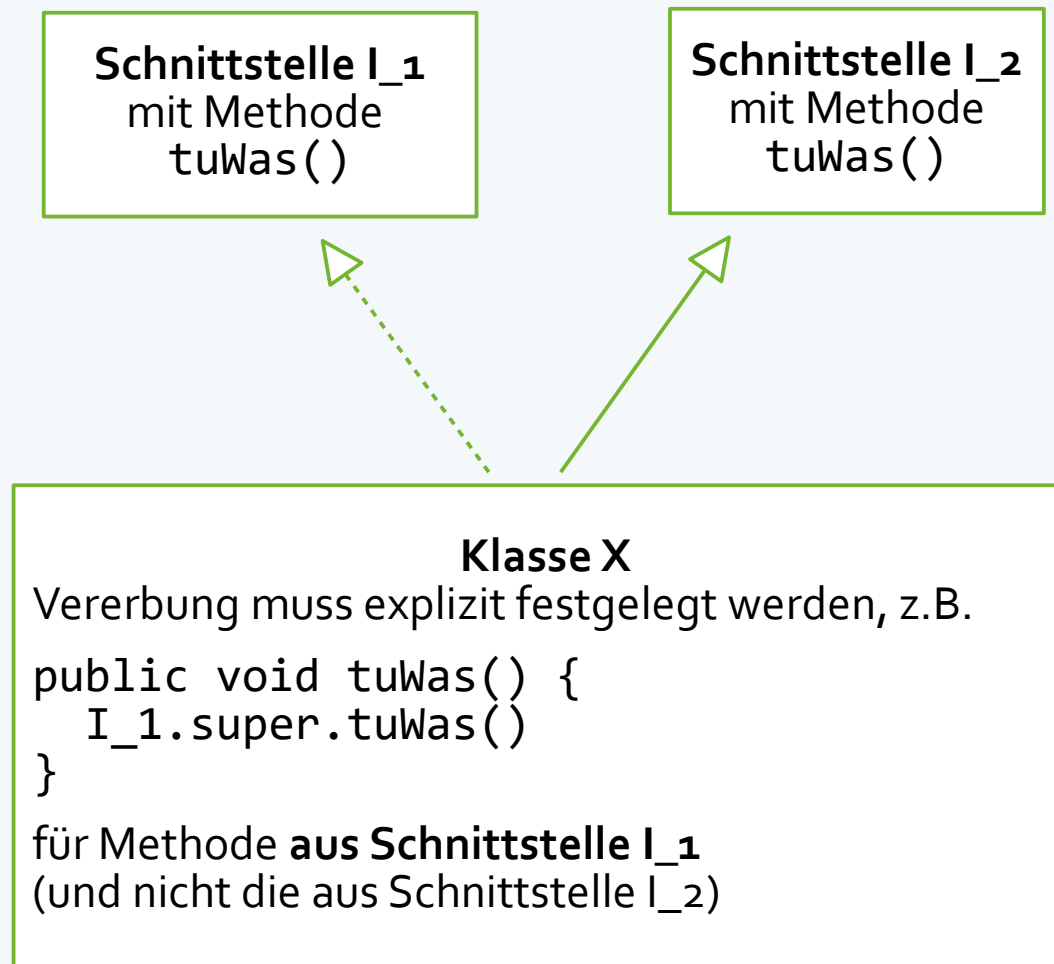
“default” und „static“: Implementierte Methoden in Schnittstellen

„Mehrfachvererbung“ von Methoden – Regel für Fall 2



„default“ und „static“: Implementierte Methoden in Schnittstellen

„Mehrfachvererbung“ von Methoden – Regel für Fall 3



„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel

```
package de.baleipzig.classes;  
  
import java.time.*;  
  
public interface TimeClient {  
    void setDateAndTime(int day, int month, int year, int hour, int minute, int second);  
  
    LocalDateTime getLocalDateTime();  
}
```

bisherige Schnittstelle, die auch schon von anderen implementiert wurde, z.B. von der Klasse „SimpleTimeClient“ auf folgender Folie
[keine Panik, wir haben uns eigentlich noch gar nicht mit Zeitangaben in Java beschäftigt, das machen wir später in diesem Semester]

Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel

```
package de.baleipzig.classes;
import java.time.*;
public class SimpleTimeClient implements TimeClient {
    private LocalDateTime dateAndTime;
    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }
    public void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(dateToSet, timeToSet);
    }
    public LocalDateTime getLocalDateTime() {
        return dateAndTime;
    }
    public String toString() {
        return dateAndTime.toString();
    }
}
```

Klasse „SimpleTimeClient“,
die die Schnittstelle
schon implementiert

Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel

```
package de.baleipzig.classes;  
  
public class TestClass {  
    public static void main(String[] args) {  
        //...  
        TimeClient alienTimeClient = new SimpleTimeClient();  
        System.out.println("Current time: " + alienTimeClient.toString());  
    }  
}
```

Das funktioniert schon gut, jetzt finde ich
mich hier zeitlich zurecht.
Mal schauen, was mein „alienTimeClient“ noch
so kann.



Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel

```
package de.baleipzig.classes;  
  
import java.time.*;  
  
public interface TimeClient {  
    void setDateAndTime(int day, int month, int year, int hour, int minute, int second);  
    LocalDateTime getLocalDateTime();  
}
```

bisherige Schnittstelle, die auch schon von anderen implementiert wurde, z.B. von der Klasse „SimpleTimeClient“ auf folgender Folie
[keine Panik, wir haben uns eigentlich noch gar nicht mit Zeitangaben in Java beschäftigt, das machen wir später in diesem Semester]

Mhhh Mit schnellen UFOs ist nicht nur die jeweils lokale Zeit relevant, man braucht auch eine für andere Zeitzonen.



Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/landl/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel

```
package de.baleipzig.classes;
import java.time.*;
public class SimpleTimeClient implements TimeClient {
    private LocalDateTime dateAndTime;
    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }
    public void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(dateToSet, timeToSet);
    }
    public LocalDateTime getLocalDateTime() {
        return dateAndTime;
    }
    public String toString() {
        return dateAndTime.toString();
    }
}
```

Klasse „SimpleTimeClient“,
die die Schnittstelle
schon implementiert;

← müsste bei einer
Schnittstellenerweiterung
um eine abstrakte
Methode für eine
Zeitangabe mit Zeitzone
modifiziert werden oder
wird unbenutzbar

Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel

```
public interface TimeClient {  
    ...  
    static ZoneId getZoneId (String zoneString) {  
        try {  
            return ZoneId.of(zoneString);  
        } catch (DateTimeException e) {  
            System.err.println("Invalid time zone: " + zoneString +  
                "; using default time zone instead.");  
            return ZoneId.systemDefault();  
        }  
    }  
  
    default ZonedDateTime getZonedDateTime(String zoneString) {  
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));  
    }  
}
```

So, Schnittstelle für mich,
mein UFO und andere
moderne, schnelle Fort-
bewegungsmittel erwei-
tert, ohne alles alte kaputt
zu machen oder
neue Schnittstellen
implementieren zu
müssen.



Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/landl/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel

```
package de.baleipzig.classes;
public class TestClass {
    public static void main(String[] args) {
        //...
        TimeClient alienTimeClient = new SimpleTimeClient();
        System.out.println("Current time: " + alienTimeClient.toString());
        System.out.println("Current time in UTC: " +
            alienTimeClient.getZonedDateTime("Europe/Berlin").
            format(DateTimeFormatter.ISO_INSTANT));

        System.out.println("Time with Japanese Time Zone: " +
            alienTimeClient.getZonedDateTime("Japan"));
        System.out.println("Japanese Time in UTC: " +
            alienTimeClient.getZonedDateTime("Japan").
            format(DateTimeFormatter.ISO_INSTANT));
    }
}
```

Jetzt geht schon mehr.



Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel - Übungsaufgabe

```
public interface TimeClient {  
    ...  
    static ZoneId getZoneId (String zoneString) {  
        try {  
            return ZoneId.of(zoneString);  
        } catch (DateTimeException e) {  
            System.err.println("Invalid time zone: " + zoneString +  
                "; using default time zone instead.");  
            return ZoneId.systemDefault();  
        }  
    }  
  
    default ZonedDateTime getZonedDateTime(String zoneString) {  
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));  
    }  
}
```

Aufgabe 8: Erweitern Sie unsere Schnittstelle um eine „default“-Methode „getCountDownMessage()“, die als Platzhalter für zukünftige Funktionalitäten einen String „Countdown initiated“ zurückliefert.



Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

„default“ und „static“ am Beispiel – Lösung 8

```
public interface TimeClient {  
    ...  
  
    default ZonedDateTime getZonedDateTime(String zoneString) {  
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));  
    }  
  
    default String getCountDownMessage() {  
        return "Countdown initiated";  
    }  
}
```



Beispiel in Anlehnung an:

Default Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (o. D.): [online]
<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html> [abgerufen am 24.03.2023].

„default“ und „static“: Implementierte Methoden in Schnittstellen

Was kann ich bei der Erweiterung einer Schnittstelle mit „default“-Methode allgemein machen?

- nichts – dann wird die Methode einfach von der Erweiterung geerbt

```
public interface AnotherTimeClient extends TimeClient { }
```

- „default“-Methode neu deklarieren – dann wird diese abstrakt und muss von implementierenden Klassen implementiert werden

```
public interface AbstractZoneTimeClient extends TimeClient {  
    public ZonedDateTime getZonedDateTime(String zoneString);  
}
```

- „default“-Methode neu schreiben – dann wird diese überschrieben

```
public interface HandleInvalidTimeZoneClient extends TimeClient {  
    default public ZonedDateTime getZonedDateTime(String zoneString) {  
        try { return ZonedDateTime.of(getLocalDateTime(), ZoneId.of(zoneString)); }  
        catch (DateTimeException e) {  
            System.err.println("Invalid zone ID ... ");  
            return ZonedDateTime.of(getLocalDateTime(), ZoneId.systemDefault());  
        }  
    }  
}
```



3

Neuere Typen und andere fortschrittliche Dinge

- „enum“: Aufzählungstyp
- „record“: Datenhalter
- „default“ und „static“: Implementierte Methoden in Schnittstellen
- „sealed“ und „non-sealed“: Einschränkung von Klassen-Benutzbarkeit ergänzend zu den klassischen Zugriffsmodifizierern

„sealed“ und „non-sealed“ – Weitere Benutzbarkeitseinschränkungen

- bisher gab es Benutzungseinschränkungen nur nach dem „alles (zumindest inhaltlich nicht eingeschränkt) oder nichts (per final)“-Prinzip
- aufgrund von Polymorphie-Anforderungen kann dies problematisch sein



„Flugobjekte“
(unvollständig):

Wie könnte man
Landemöglichkeiten für
„Flugobjekte“ allgemein
spezifizieren?

- Rollbahn
- Hubschrauber-
Landeplatz
- Ballon-Landeplatz
- UFO-Landeplatz
- ...

„sealed“ und „non-sealed“ – Weitere Benutzbarkeitseinschränkungen

- bisher gab es Benutzungseinschränkungen nur nach dem „alles (zumindest inhaltlich nicht eingeschränkt) oder nichts (per final)“-Prinzip
- aufgrund von Polymorphie-Anforderungen kann dies problematisch sein



„Flugobjekte“
(unvollständig):

Wie könnte man
Landemöglichkeiten für
„Flugobjekte“ allgemein
spezifizieren?

- Rollbahn
- Hubschrauber-Landeplatz
- Ballon-Landeplatz
- UFO-Landeplatz



„sealed“ und „non-sealed“ – Weitere Benutzbarkeitseinschränkungen

Identifikator „sealed“

- gibt Beschränkung von Klassen und Schnittstellen hinsichtlich Erweiterbarkeit an
- Identifikator „**permits**“ spezifiziert die erlaubten Klassen und Schnittstellen

```
... sealed class Flugobjekt ... permits Airplane, Helicopter, Balloon, UFO {...}
```



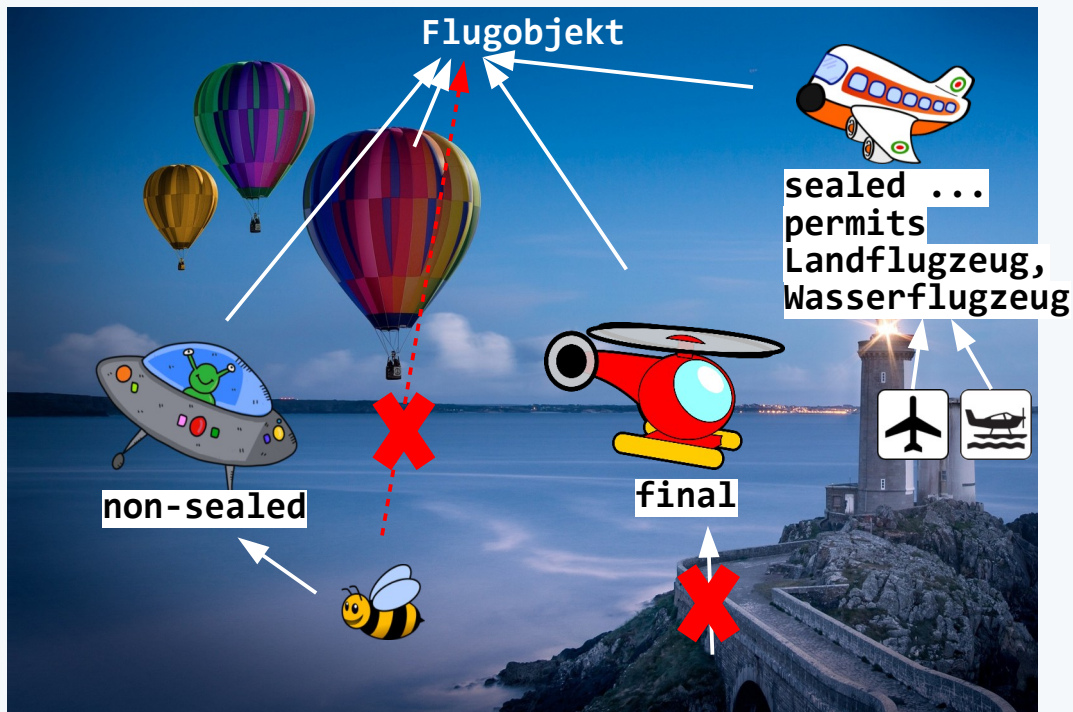
- **Ausnahme:**
bei verschachtelten Klassen und Schnittstellen kann „permits“ entfallen und eingebettete, erweiternde Klassen **mit Namen** werden automatisch als erlaubte Klassen angesehen

Anonyme Klassen und lokale, innere Klassen können so oder so nicht erlaubt werden

„sealed“ und „non-sealed“ – Weitere Benutzbarkeitseinschränkungen

Bedingungen, die die erlaubten, erweiternden Dinge erfüllen müssen

- müssen im gleichen Modul oder im gleichen Paket (bei unbenanntem Modul) sein
- müssen die versiegelte Klasse/Schnittstelle direkt erweitern bzw. implementieren
- müssen durch einen Modifizierer angeben, wie die „Versiegelung“ weitergegeben wird:



- **final:**
weitere Erweiterung ist nicht möglich
- **sealed:**
Unterklasse/Schnittstelle ist auch „sealed“ und spezifiziert von sich erlaubte Dinge
- **non-sealed:**
Erweiterung ist grundsätzlich möglich

Vielen Dank für Ihre
Aufmerksamkeit.

