



PM2 Java: Aufzählungstypen = Enums



Definition

- Eine Aufzählungstyp (eng. enumerable datatype kurz **Enum**) ist ein Datentyp, der einen endlichen Wertebereich repräsentiert.
- Es gibt in der realen Welt eine Reihe von Beispielen für Aufzählungstypen:
 - Geschlecht {weiblich männlich}
 - Wochentag {Montag, ..., Sonntag}
 - Jahreszeiten {Frühjahr,..., Winter}
 - Währungssymbole
- In Programmiersprachen muss zur Abbildung von Aufzählungstypen die Eigenschaft eines endlichen Wertebereichs gewährleistet sein.
- Enums verhalten sich ähnlich wie Konstanten. Nach der Definition eines Enums darf der Wertebereich eines Enums nicht mehr verändert werden.
- Alle Datentypen, die wir bisher in Java kennengelernt haben, unterstützen diese Anforderungen nicht per se.
- Es lassen sich allerdings mit den uns bekannten Sprachmitteln Klassen definieren, die sich wie Enums verhalten.



Beispiel *EventCalendar*

- Wir wollen einen Veranstaltungskalender (*EventCalendar*) modellieren, der Jahreszeiten auf Listen von Veranstaltungen abbildet. Veranstaltungen stellen wir zur Vereinfachung als Strings dar.
- Jahreszeiten haben einen endlichen Wertebereich.
- Jahreszeiten modellieren wir als Konstanten der Klasse *Season1*, die wir als ganzzahlige Werte darstellen.
- Um sicher zu stellen, dass es nur 4 Jahreszeiten gibt, wird der Konstruktor von *Season1* *private* und die Klasse *Season* *final*.

```
package seasons.classes;
```

```
public final class Season1 {
```

```
    public static final int SPRING = 0;
```

```
    public static final int SUMMER = 1;
```

```
    public static final int AUTUMN = 2;
```

```
    public static final int WINTER = 3;
```

```
    private Season1() {}
```

```
}
```



Beispiel

- Der **EventCalendar1** hat die Methoden:

add(int,String) zum Eintragen von Veranstaltungen zu einer Jahreszeit

getEvents(int), die zu einer Jahreszeit die Liste der Veranstaltungen zurückliefert

toString(), die den Kalender als String aufbereitet und dies auf die **toString()** Methode einer **Map** zurückführt

- Die Parametertypen für die Jahreszeiten sind **int**, da die Konstanten von **Season1** auf ganzzahlige Werte abgebildet werden.

```
public class EventCalendar1 {  
    private Map<Integer,List<String>>  
        events;  
  
    public EventCalendar1() {  
        this.events = new  
            HashMap<Integer,List<String>>();  
    }  
  
    public void add(int season, String  
        event){ ... }  
  
    public List<String> getEvents(int  
        jahreszeit){ ... }  
  
    @Override  
    public String toString() {...}  
        return events.toString();  
    }  
}
```



Beispiel: *EventCalendar1* Source

```
public class EventCalendar1 {
    private Map<Integer,List<String>> events;
    public EventCalendar1(){
        this.events = new HashMap<Integer,List<String>>();
    }
    public void add(int season, String event){
        if (events.get(season)== null){
            events.put(season, new ArrayList<String>());
        }
        events.get(season).add(event);
    }
    public List<String> getEvents(int jahreszeit){
        return events.get(jahreszeit);
    }
    @Override
    public String toString() {
        return events.toString();
    }
}
```



Nicht typsichere Modellierung von *Season*

- **Nachteil:**
 - Enums durch Konstanten einer finalen Klasse abzubilden ist nicht typsicher: wir können für beliebige Zahlenwerte Einträge in den Kalender vornehmen und werden nicht gezwungen im Wertebereich von *Season1* zu bleiben.

```
public class EventCalendar1 {  
    public static void main(String[] args) {  
        EventCalendar1 calendar = new EventCalendar1();  
        calendar.add(Season1.SPRING, "Frozen Festival");  
        // Hinzufügen über beliebige Integer Werte ist möglich  
        // erlaubt sind aber nur die 4 Jahreszeiten  
        calendar.add(4, "Event in einem bizarren Kalender");  
        p(calendar);  
        // Auslesen über beliebige Integer Werte ist möglich  
        p(calendar.getEvents(5));  
        p(calendar.getEvents(Season1.SPRING));  
    }  
}
```



Nicht typsichere Modellierung von *Season*

- **Nachteil:**
 - In der Ausgabe sind nur Zahlenwerte aber keine Bezeichner für Jahreszeiten zu sehen.



```
{0=[Frozen Festival], 4=[Event in einem bizarren  
Kalender]}
```

```
null  
[Frozen Festival]
```

```
public class EventCalendar1 {  
    public static void main(String[]  
        args) {  
        EventCalendar1 calendar = new  
            EventCalendar1();  
        calendar.add(Season1.SPRING,  
            "Frozen Festival");  
        // Hinzufügen über beliebige  
        // Integer Werte ist möglich  
        // erlaubt sind aber nur die 4  
        // Jahreszeiten  
        calendar.add(4, "Event in einem  
            bizarren Kalender");  
        p(calendar);  
        // Auslesen über beliebige  
        // Integer Werte ist möglich  
        p(calendar.getEvents(5));  
        p(calendar.getEvents(Season1.SP  
            RING));  
    }  
}
```



Nicht typsichere Modellierung von *Season*

- **Nachteil:**
 - um in der Ausgabe sprechende Bezeichner für die Konstanten in *Season1* zu erhalten müssen wir in der *toString()* Methode von *EventCalendar1* diese Konstanten in Namen übersetzen.
 - Das verstößt gegen die Prinzipien **Entwurf nach Zuständigkeit, minimale Koppelung und maximale Kohäsion**.

```
public String toString() {  
  
    StringBuilder sb = new  
        StringBuilder("{}");  
    for (Map.Entry<Integer, List<String>>  
        entry : events.entrySet()) {  
        switch (entry.getKey()) {  
            case Season1.SPRING:  
                sb.append("Frühjahr="); break;  
            case Season1.SUMMER:  
                //... etc  
        }  
        sb.append(entry.getValue()).append(",");  
    }  
    sb.append("{}");  
    return sb.toString();  
}
```




Eine typsichere Alternative zu Variante 1

Nachteile Variante 1

- Obwohl wir einen begrenzten Wertebereich in der Klasse **Season1** definieren, können bei der Nutzung Werte verwendet werden, die außerhalb des Wertebereichs liegen.
- Dies liegt an der Abbildung durch Klassenkonstanten, deren Werte keine Instanzen der Aufzählungsklasse sind.
- Dadurch kann auch die **toString** Methode nicht auf eine für den Typ **Season1** definierte **toString** Methode zurückgeführt werden.

Typsichere Alternative

- Die Klassenkonstanten der **final** Klasse **Season2** sind Instanzen der Klasse.
- Ein privater Konstruktor garantiert, dass keine weiteren Instanzen außer den definierten Konstanten erzeugt werden können.
- Die **toString** Methode implementieren wird typabhängig in **Season2**.
- Um für jede Konstante eine Darstellung zu erzeugen, die unabhängig von Namen der Konstanten ist, übergeben wir im Konstruktor eine **String**-Darstellung.
- Bei der Nutzung verwenden wir den Typ **Season2**. Als Werte sind dann nur noch die Konstanten möglich.



Typsichere Alternative

- Die Klassenkonstanten der **final** Klasse **Season2** sind Instanzen der Klasse.
- Ein privater Konstruktor garantiert, dass keine weiteren Instanzen außer den definierten Konstanten erzeugt werden können.
- Die **toString()** Methode implementieren wird typabhängig in **Season2**.
- Um für jede Konstante eine Darstellung zu erzeugen, die unabhängig von Namen der Konstanten ist, übergeben wir im Konstruktor eine **String**-Darstellung.

```
public final class Season2 {  
  
    public static final Season2 SPRING =  
        new Season2("Frühjahr");  
    public static final Season2 SUMMER =  
        new Season2("Sommer");  
    public static final Season2 AUTUMN =  
        new Season2("Herbst");  
    public static final Season2 WINTER =  
        new Season2("Winter");  
    private String asString;  
    private Season2(String name) {  
        this.asString = name;  
    }  
    public String getAsString() {  
        return asString;  
    }  
    @Override  
    public String toString() {  
        return asString;  
    }  
}
```



Typsichere Alternative

- **EventCalendar2** verwendet in den Signaturen der Methoden und für den Typ der Schlüssel in **events** jetzt den Typ **Season2**.
- **toString()** kann jetzt auf das **toString()** einer **Map** zurückgeführt werden, da die **Season2** Konstanten eine Methode **toString()** implementieren.

```
public class EventCalendar2 {  
  
    private Map<Season2, List<String>>  
        events;  
  
    public EventCalendar2() {  
        this.events = new  
            HashMap<Season2, List<String>>();  
    }  
  
    public void add(Season2 season,  
        String event) {...}  
  
    public List<String>  
        getEvents(Season2 jahreszeit) {  
        ... }  
  
    public String toString() {  
        return events.toString();  
    }  
}
```



Typsichere Alternative

- Zulässige Werte für die Schlüssel in *EventCalendar2* sind dann nur Instanzen der Klasse *Season2*, von den es genau vier Konstanten gibt.

```
public class EventCalendar2 {  
  
    public static void main(String[] args) {  
        EventCalendar2 calendar = new EventCalendar2();  
        calendar.add(Season2.SPRING, "Frozen Festival");  
        calendar.add(Season2.SUMMER, "Wacken Festival");  
        // calendar.add(4, "Event in einem bizarren Kalender"); // Fehler  
        p(calendar);  
        //p(calendar.getEvents(5)); // Fehler  
        p(calendar.getEvents(Season2.SPRING));  
    }  
}
```



Verwendung von Java Aufzählungstypen (*enum*)

- Mit Java 5 sind Aufzählungstypen fester Bestandteil der Sprache. (Schlüsselwort *enum*)
- Sie erleichtern das Aufschreiben von Aufzählungstypen. Der Compiler übersetzt diese Definitionen in ähnliche Definitionen wie die der vorausgehenden Alternative und stellt eine Subklassenbeziehung zur Klasse *Enum* her.
- Enums werden ähnlich wie Klassen definiert.
- Die einfachste Form eines Enums ist die Aufzählung der Werte in einem Block.
- Wir verwenden nun den Enumtyp *Season0*, um in der Klasse *EventCalendar0* einen typischeren Wertebereich für die erlaubten Jahreszeiten zu definieren.

```
public enum Season0 {  
    SPRING, SUMMER, AUTUMN, WINTER;  
}  
  
public class EventCalendar0 {  
    private Map<Season0, List<String>>  
        events;  
  
    public EventCalendar0() { ... }  
    public void add(Season0 season,  
        String event) {}  
  
    public List<String>  
        getEvents(Season0 season) {...}  
    public String toString() {  
        return events.toString();  
    }  
}
```



enum für typsichere Aufzählungen

- Dann erzeugt die Verwendung von Werten außerhalb des Wertebereichs des Enum-Typen einen Compilerfehler.

```
public class EventCalendar0 {  
  
    public static void main(String[] args) {  
        EventCalendar0 calendar = new EventCalendar0();  
        calendar.add(SPRING, "Frozen Festival");  
        calendar.add(Season0.SUMMER, "Wacken Festival");  
        // calendar.add(4, "Event in einem bizarren Kalender"); // Fehler  
        p(calendar);  
        //p(calendar.getEvents(5)); // Fehler  
        p(calendar.getEvents(Season0.SPRING));  
    }  
}
```



enum für typsichere Aufzählungen

- Allerdings ist die Ausgabe noch nicht in der gewünschten Form. Ausgegeben werden noch die Namen der Enumwerte.
- Der Compiler erzeugt bei der Übersetzung eines Enum-Typen eine *toString()* Methode für den Enum-Typ, die den Namen der Konstanten ausgibt.

```
public class EventCalendar0 {  
  
    public static void main(String[] args) {  
        EventCalendar0 calendar = new EventCalendar0();  
        calendar.add(SPRING, "Frozen Festival");  
        calendar.add(Season0.SUMMER, "Wacken Festival");  
        // calendar.add(4, "Event in einem bizarren Kalender"); // Fehler  
        p(calendar);  
        //p(calendar.getEvents(5)); // Fehler  
        p(calendar.getEvents(Season0.SPRING));  
    }  
}
```



```
{SPRING=[Frozen Festival], SUMMER=[Wacken Festival]}  
[Frozen Festival]
```



Methoden für *enum* Typen

- Um die Ausgabe unabhängig vom Namen der Enumwerte zu machen, überschreiben wir die Methode *toString()* in *Season*.
- Dazu definieren wir die Instanzvariable *asString* und einen Konstruktor, so dass bei der Erzeugung der Enumwerte die Darstellung übergeben werden kann.
- Die Enum-Instanzen können wir dann verwenden wie Konstruktoren einer Klasse (*WINTER("Winter")*)


```
public enum Season2 {  
  
    SPRING("Frühjahr"), SUMMER("Sommer"),  
    AUTUMN("Herbst"), WINTER("Winter");  
  
    private String asString;  
    private Season2(String name) {  
        this.asString = name;  
    }  
    @Override  
    public String toString() {  
        return asString;  
    }  
}
```




Ergebnis mit der Methode *toString* in *Season*

- Dann liefert das gleiche Programm wie vorher die gewünschte Ausgabe.

```
public class EventCalendar2 {  
  
    public static void main(String[] args) {  
        EventCalendar2 calendar = new EventCalendar2();  
        calendar.add(SPRING, "Frozen Festival");  
        calendar.add(Season2.SUMMER, "Wacken Festival");  
        // calendar.add(4, "Event in einem bizarren Kalender"); // Fehler  
        p(calendar);  
        //p(calendar.getEvents(5)); // Fehler  
        p(calendar.getEvents(Season2.SPRING));  
    }  
}
```

 {Frühjahr=[Frozen Festival], Sommer=[Wacken Festival]}
[Frozen Festival]



Die Werte eines *enum* Typen sind *public static*

- Durch einen statischen Import können die Enumwerte direkt ohne Präfix des Enum-Typen verwendet werden.

```
import static seasons.enums.Season0.*;

public class EventCalendar0 {
    public static void main(String[]
        args) {
        EventCalendar0 calendar = new
            EventCalendar0();
        calendar.add(SPRING, "Frozen
            Festival");
        calendar.add(SUMMER, "Wacken
            Festival");
        // calendar.add(4, "Event in
            einem bizarren Kalender"); //
            Fehler
        p(calendar);
        //p(calendar.getEvents(5));
        // Fehler
        p(calendar.getEvents(SPRING));
    }
}
```



Die Werte eines *enum* Typen sind *public static*

- Daher müssen wir bei Initialisierungen von Enum-Werten außerhalb der Definition **statische Initialisierer** verwenden.
- **Beispiel:** Wir wollen Jahreszeiten nach ihrem Vorgänger und Nachfolger befragen. Dazu erweitern wir das Enum *Season3* um 2 Instanzvariablen *pred* und *succ*.
- Bei der Definition des ersten Enum-Wertes *SPRING* sind die nachfolgenden Enum-Werte noch nicht definiert. Daher können wir *SPRING* nur mit einem Bezeichner erzeugen. Die Beziehung zu Vorgänger und Nachfolger müssen wir im statischen Initialisierungsblock nachholen.

```
public enum Season3 {  
    SPRING("Fruehjahr"),  
    SUMMER("Sommer", SPRING),  
    AUTUMN("Herbst", SUMMER),  
    WINTER("Winter", AUTUMN, SPRING);  
  
    private Season3 succ;  
    private Season3 pred;  
    private String asString;  
  
    static {  
        SPRING.succ = SUMMER;  
        SPRING.pred = WINTER;  
        AUTUMN.succ = WINTER;  
        SUMMER.succ = AUTUMN;  
    }  
    ...  
}
```



Season Source

```
public enum Season3 {  
    SPRING("Fruehjahr"), SUMMER("Sommer", SPRING), AUTUMN("Herbst", SUMMER),  
        WINTER("Winter", AUTUMN, SPRING);  
    private Season3 succ;  
    private Season3 pred;  
    private String asString;  
    static {  
        SPRING.succ = SUMMER;  
        SPRING.pred = WINTER;  
        AUTUMN.succ = WINTER;  
        SUMMER.succ = AUTUMN;  
    }  
    private Season3(String name) {this(name, null);}  
    private Season3(String name, Season3 succ) {this(name, succ, null);}  
    private Season3(String name, Season3 succ, Season3 pred) {this.asString = name;  
        this.succ = succ; this.pred = pred;}  
  
    public Season3 getSucc() {return succ;}  
    public Season3 getPred() {return pred;}  
    public String getAsString() {return asString;}  
    @Override  
    public String toString() {return asString;}  
}
```



Die Klasse *Enum*: Basismethoden für Aufzählungstypen

- Entdeckt der Compiler das Schlüsselwort *enum*, dann erzeugt er intern eine Subklasse der Klasse *Enum*.
- *Enum* enthält eine Reihe von hilfreichen Methoden:
 - *values* (statische Methoden) liefert die Enum-Instanzen in der Reihenfolge der Definition als Array.
 - *ordinal* liefert die Position einer Enum-Instanz in der Aufzählung zurück.
 - *toString* liefert den Namen einer Enum-Instanz
 - *valueOf(String)* (statische Methoden) liefert die einem Namen zugehörige Enum-Instanz
 - *name()* liefert den Namen einer Enum-Instanz
 - *equals* und *hashCode* aus *Object*
 - *compareTo()* vergleicht Enum-Instanzen über ihre Position.



Enum Methoden am *Season* Beispiel

```
package enums.methods;

public class EnumMethodsDemo {

    public static void main(String[] args) {
        Season2[] seasons = Season2.values();
        for (Season2 sea : seasons) {
            printf("%1$s Position %2$d Typ %3$s %4$s " +
                "compareTo(WINTER) %5$d\n",
                    sea.name(),
                    sea.ordinal(),
                    Season2.valueOf(sea.name()).getClass().getSimpleName(),
                    Season2.valueOf(sea.name()),
                    sea.compareTo(WINTER));
        }
    }
}
```



```
SPRING Position 0 Typ Season2 Frühjahr compareTo(WINTER) -3
SUMMER Position 1 Typ Season2 Sommer compareTo(WINTER) -2
AUTUMN Position 2 Typ Season2 Herbst compareTo(WINTER) -1
WINTER Position 3 Typ Season2 Winter compareTo(WINTER) 0
```



Enum Methoden am *Season* Beispiel

```
package enums.methods;

public class EnumMethodsDemo {

    public static void main(String[] args) {
        Season2[] seasons = Season2.values();
        for (Season2 sea : seasons) {
            printf("%1$s.equals(%2$s):%3$s %1$s==%2$s:%4$s " +
                "%1$s.hashCode:%5$d %2$s.hashCode:%6$d\n",
                sea, WINTER,
                sea.equals(WINTER),
                sea == WINTER,
                sea.hashCode(),
                WINTER.hashCode());
        }
    }
}
```



```
Frühjahr.equals(Winter):false Frühjahr==Winter:false Frühjahr.hashCode:1476323068
Winter.hashCode:2038935242
Sommer.equals(Winter):false Sommer==Winter:false Sommer.hashCode:1414964377
Winter.hashCode:2038935242
Herbst.equals(Winter):false Herbst==Winter:false Herbst.hashCode:72377361
Winter.hashCode:2038935242
Winter.equals(Winter):true Winter==Winter:true Winter.hashCode:2038935242
Winter.hashCode:2038935242
```



Enum-Typen versus Klassen

Gemeinsamkeiten

- Sie erben die Methoden von *Object*.
- Sie haben Konstruktoren. (→ *Season2*)
- Sie haben Objektvariablen. (→ *Season2*)
- Sie können Methoden definieren und überschreiben. (→ *Season2.toString*)
- Enum-Instanzen können abstrakte Methoden des Enumtyps überschreiben.

Unterschiede

- Es gibt keine Subklassen von Enumtypen, da sie immer *final* sind.
- Enum-Typen können im Gegensatz zu Klassen wie integrale Typen in *switch-case* Statements verwendet werden.



Enum-Instanzen überschreiben abstrakte Methoden des Enum-Typs

- **Beispiel:** Wir wollen ein Enum für die mathematischen Grundoperationen schreiben. Die Methode *eval* soll den Operator auf die übergebenen Argumente anwenden.
- Wir schreiben zunächst die Methode *eval* als Methode des Enum-Typen *Operation*.
- Diese Lösung hat mehrere Nachteile:
 1. Die *Exception* am Ende der Methode.
 2. Wird die Menge der Operatoren erweitert, muss jedesmal die *eval* -Methode angepasst werden.
 3. Ein *switch – case* Statement ist schlechter Stil. Die Auswahl der Berechnung sollte polymorph an die Enum-Instanz gebunden sein.

```
public enum Operation0 {  
    PLUS, MINUS, TIMES, DIV;  
    public double eval(double d1, double  
        d2) {  
        switch (this) {  
            case PLUS:  
                return d1 + d2;  
            case MINUS:  
                return d1 - d2;  
            case TIMES:  
                return d1 * d2;  
            case DIV:  
                return d1 / d2;  
        }  
        throw new AssertionError("Unknown  
            op: " + this);  
    }  
}
```



Enum-Instanzen überschreiben abstrakte Methoden des Typs

- Es ist möglich bei der Definition von Enum-Instanzen eine abstrakte Methode des Enum-Typs zu überschreiben.
- Wir überschreiben die abstrakte Methode *eval* in jeder Enum-Instanz. Die Implementierung der Methoden steht in dem Block, der der Definition der Enum-Instanz folgt.

```
public enum Operation1 {  
    PLUS("+") {public double eval(double d1,double d2) {return d1+d2;}},  
    MINUS("-") {public double eval(double d1,double d2) {return d1-d2;}},  
    TIMES("*") {public double eval(double d1,double d2) {return d1*d2;}},  
    DIV("/") {public double eval(double d1,double d2) {return d1/d2;}};  
  
    private String sym;  
    private Operation1(String sym){ this.sym = sym; }  
    public abstract double eval(double d1,double d2);  
    @Override  
    public String toString() {return sym;}  
}
```



Enum-Instanzen überschreiben abstrakte Methoden des Typs

```
package enums.operation.sample;

import static util.Printer.*;

public class OperationDemo {

    public static void main(String[] args) {
        double d1 = 4;
        double d2 = 3;
        String result;
        for (Operation op : Operation.values()) {
            result =
                String.format("%1$g %2$s %3$g = %4$g", d1, op, d2, op.eval(d1, d2));
            p(result);
        }
    }
}
```



```
4.00000 + 3.00000 = 7.00000
4.00000 - 3.00000 = 1.00000
4.00000 * 3.00000 = 12.0000
4.00000 / 3.00000 = 1.33333
```



Enum-Typen können wie ganzzahlige Typen in *switch-case* Statements verwendet werden

- Durch die Methode *ordinal* werden Enum-Instanzen Zahlen zugeordnet.
- Auf Basis dieser Abbildung ist die Verwendung von Enum-Typen in *switch-case* Statements möglich, die sonst in Java nur ganzzahlige Werte zulassen.
- Rechts nutzen wir dies, um die Ampelphasen durchzuschalten.
- **Beachte:** in den *case* Klauseln können die Enum-Instanzen ohne Voranstellen des Enum-Typs verwendet werden.
- Dies gilt nur für „echte“ Enum-Typen, nicht für die über Klassen „simulierten“ Enum-Typen.

```
enum Signal {GREEN, YELLOW, RED}

public class TrafficLight {
    private Signal color= Signal.RED;
    public void change() {
        switch (color) {
            case RED: color = Signal.GREEN;
                    break;
            case GREEN: color =
                Signal.YELLOW; break;
            case YELLOW: color =
                Signal.RED; break;
        }
    }

    public String toString() {
        return "Traffic Light is " +
            color;
    }
}
```

Enum-Typen können wie ganzzahlige Typen in *switch-case* Statements verwendet werden



- Wenn wir die Ampel 5 mal umschalten, erzeugt das Programme die Ausgabe:

```
Traffic Light is RED
Traffic Light is GREEN
Traffic Light is YELLOW
Traffic Light is RED
Traffic Light is GREEN
```



```
enum Signal {GREEN, YELLOW, RED}

public class TrafficLight {
    private Signal color= Signal.RED;
    public void change() {...}
    public String toString() {...}

    public static void main(String[]
        args) {
        TrafficLight light = new
            TrafficLight();
        for (int i = 0; i < 5; i++) {
            p(light);
            light.change();
        }
    }
}
```

Enum-Typen können wie ganzzahlige Typen in *switch-case* Statements verwendet werden



- Dies gilt nur für „echte“ Enum-Typen, nicht für die über Klassen „simulierten“ Enum-Typen.
- Wenn wir das *enum Signal* durch die Klasse *SignalClass* ersetzen, die sich ähnlich verhält wie *Signal*, dann erzeugt die Verwendung der Konstanten in *switch-case* einen Compilerfehler.

```
public final class SignalClass {
    static final SignalClass GREEN = new SignalClass("GREEN");
    static final SignalClass YELLOW = new SignalClass("YELLOW");
    static final SignalClass RED = new SignalClass("RED");
    private String color;
    public String toString() {return color;}
    private SignalClass(String color) {this.color = color;}

    public static void main(String[] args) {
        SignalClass color = GREEN;
        switch (color) { // Compilerfehler
            // Cannot switch on a value of type SignalClass.
            // Only convertible int values or enum constants are permitted
            case GREEN:
        }
    }
}
```



Zusammenfassung

- Aufzählungstypen oder Enums sind Typen mit einem endlichen Wertebereich.
- Vor Java 5 mussten die Aufzählungstypen mit den Mitteln der Sprache simuliert werden. Man erreicht trotz einer geschickten Implementierung nicht vollständige Typsicherheit.
- Mit Java 5 sind Enums fester Bestandteil der Sprache. Sie werden mit dem Schlüsselwort **enum** gekennzeichnet.
- Echte Enum Typen sind mächtiger als die durch Klassen simulierten Aufzählungstypen, da
 - der Compiler bei der Übersetzung die Ableitungsbeziehung zur Klasse **Enum** herstellt, die eine Reihe nützlicher Methoden für Enums enthält.
 - Enum-Instanzen in **switch-case** Statements verwendet werden können.
 - es eine Reihe von Kurzschreibweisen bei der Definition und Verwendung gibt.
- Enum Typen verhalten sich nahezu wie Klassen, mit den Einschränkungen, das
 - sie nicht von einer Klasse ableiten dürfen und
 - nicht von ihnen abgeleitet werden darf