Inhaltsverzeichnis

1. Java Enums

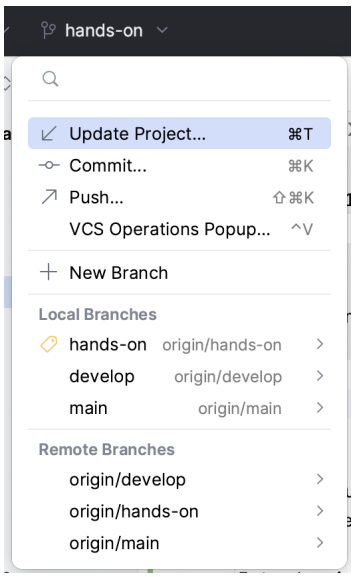
1.1. Preparation

Das **Projekt** bzw. der "*lokale Workspace*", d.h. euer lokales Arbeitsverzeichnis, in dem alle Sourcen liegen, muss als allererstes zum Start in den Tag aktualisiert werden, d.h. …

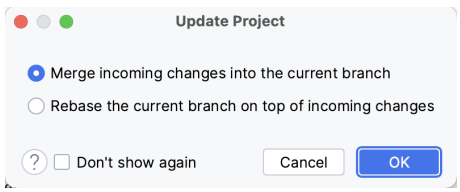
↙ `Update Project…`

Das geht am besten mithilfe der IDE im Menü oder über das GIT Icon:

Das geht am besten mithilfe der IDE im Menü oder über das GIT Icon:



Danach muss - im sich öffnenden Dialog - noch folgendes bestätigt werden: *merge incoming changes into the current branch*



Der Vorgang sollte mit einer Erfolgsmeldung abschließen.

1.2. Theorie & Einführung

`Enumerations` (Aufzählungen) sind vordefinierte **Wertelisten** und können typsicher genutzt werden.

Java 5 hat das Schlüsselwort `enum` eingeführt. Es bezeichnet einen speziellen Typ einer Klasse und erbt immer (implizit) von `java.lang.Enum`. Die offizielle Dokumentation findet sich hier: → Enum.html (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Enum.html)

Konstanten, die mithilfe von Enums definiert werden,

- machen den Code besser lesbar,
- erlauben typ-sichere Nutzung zur Compile-Time,
- listen den Wertebereich von vorneherein auf und
- vermeiden Fehler durch die Verhinderung von falschen Werten, die nicht aus der Menge der definierten stammen.

Die Aufzählungswerte haben einen **Index** (=Ordinal), beginnend mit `0`, und können automatisch von einem String in den gleichlautenden Enum-Wert umgewandelt werden.

Ein einfaches Beispiel:

JAVA

```java
public enum Color {
    Red,   // ordinal 0
    Green, // ordinal 1
    Blue;  // ordinal 2
}

// Die Nutzung erfolgt einfach durch:
Color blue = Color.Blue;

// Das Ordinal
int ordinal = Color.Blue.ordinal();

// oder bzgl. der Umwandlung von String in den enum Typ
Color blue = Color.valueOf("Blue");
```

Der Aufruf der von `enum` zur Verfügung gestellten Methode `ordinal()` gibt entsprechend die **Ordnungszahl** bzw. den **Positionsindex** des jeweiligen Enum-Wertes zurück.

1.2.1. Zusätzliche Methoden

`Enums` erlauben darüber hinaus die Implementierung von **zusätzlichen Methoden** in der Enum-Klasse.

*→ **Demo 1** in EnumsTest.java*

JAVA

```java
public enum Color {
    Red,
    Green,
    Brown,
    Unknown;

    public String getPalette() {
        return "RGB";
    }
}
```

1.2.2. Enums in Switch-Ausdrücken

Ein weiterer Vorteil ist die bequeme Nutzung der enum Konstanten in `switch` Statements:

*→ **Demo 2** in EnumsTest.java*

JAVA
```java
public enum ColorPalette {
    RGB, CYMK;
}

switch (color) {
    case RGB  -> { /* ... do stuff ...       */ }
    case CYMK -> { /* ... do other stuff ... */ }
}
```

Ergänzung zur Nutzung vom **logischen UND** in den Unit-Tests → siehe `src/test/java/de/dhbw/enums/OperatorTests.java`

1.2.3. Zusätzliche Datenfelder

Das Enums "normale" Klassen sind, lassen sie sich mit **instanz-spezifischen Datenfeldern** (*Instanzvariablen*) erweitern. Dazu müssen lediglich die Datenfelder selbst sowie ein oder mehrere **Konstruktoren** erzeugt werden, um die (ebenfalls konstanten) Werte für die Enum-Konstanten zu definieren.

Dazu ein Beispiel:

*→ **Demo 3** in EnumsTest.java*

JAVA
```java
public enum FoodGroup {

  Fruchtsaefte("Fruchtsäfte"),
  Suesswaren("Süßwaren"),
  Sonstiges("Sonstiges");

  private final String label;

  FoodGroup(String label) {
    this.label = label;
  }
}
```

1.3. Logische Operatoren

Die **logischen Operatoren** dienen zum Vergleich von Wahrheitswerten und werden z. B. für Bedingungen von `if` Anweisungen oder Schleifen verwendet.

Durch die logischen Operatoren können einzelne Wahrheitswerte negiert oder verknüpft werden, wobei das Ergebnis immer ein `boolean` ist.

Es gibt verschiedene **logische Operatoren**:

Operator	Erläuterung/Bedeutung
&&	*Doppeltes UND ist eine logische UND-Verknüpfung, bei der wir nur ein wahres Ergebnis erhalten, wenn beide Werte wahr sind. Ist an dieser Stelle bereits der erste Operator falsch (false) so wird der zweite Operand nicht mehr ausgewertet, da false und irgendwas bei einer logischen UND-Verknüpfung als Resultat immer false hat.*

Operator	Erläuterung/Bedeutung
&	*Einfaches UND ist eine logische UND-Verknüpfung. Bei dieser logischen UND-Verknüpfung werden beide Operanden ausgewertet.*
||	*Doppeltes ODER ist eine logische ODER-Verknüpfung, bei der wir nur ein falsches Ergebnis erhalten, wenn beide Werte falsch sind. Ist an dieser Stelle bereits der erste Operator wahr (true) so wird der zweite Operand nicht mehr ausgewertet, da true und irgendwas bei einer logischen ODER-Verknüpfung als Resultat immer true hat.*
|	*Einfaches ODER ist eine logische ODER-Verknüpfung. Bei dieser logischen ODER-Verknüpfung werden beide Operanden ausgewertet.*
!	*Das Ausrufezeichen ist der Negierungsoperator in der booleschen Logik. Aus wahr (true) wird falsch (false) und umgekehrt.*
^	*Das "Dach" wird als Exklusiv-Oder bezeichnet. Entweder der erste oder der zweite Ausdruck muss wahr sein. Es dürfen aber nicht beide Ausdrücke gleich sein, damit das Ergebnis wahr wird.*

Tests zur **Demonstration**:

```
module-enums/src/test/java/de/dhbw/enums/OperatorTests.java
```

1.4. Demonstrationen

Die Unit-Tests zur **Demonstration** finden sich hier:

```
src/test/java/de/dhbw/enums/EnumTests.java
```

Der zugehörige, in den Tests genutzte **Quellcode** findet sich hier:

```
src/main/java/de/dhbw/enums/demo/*.java
```

1.5. Exercises

Nutze folgendes Package für deine **Unit-Tests**:

```
src/test/java/de/dhbw/enums/ExerciseTests.java
```

Die im Test benutzten **Implementierungen** gehören in das Package:

```
src/main/java/de/dhbw/enums/exercises/*.java
```

Übung 1:

Ändere die Methode `mix` in der Enumeration `Color`, sodass sie anstatt des Konditional-Ausdruckes `if-elseif-else` nun den `switch` Ausdruck nutzt, um die Mischfarben zurückzugeben.

Übung 2:

Erweitere die vorbereitete Enumeration `de.dhbw.enums.exercises.RgbColor` mit drei Attributen `r` (red), `g` (green) und `b` (blue), jeweils mit Datentyp `int`, sodass jede Enum-Konstante direkt den RGB-Wert erhält.

Nutze die Methode `getRGB()`, um die jeweils zugeordneten RGB-Werte in einem Unit-Test zu prüfen.

Übung 3:

Erstelle eine `boolean` Variable, *negiere* diese und teste das Ergebnis in einem Unit-Test.

1.6. Tipps, Patterns & Best Practices