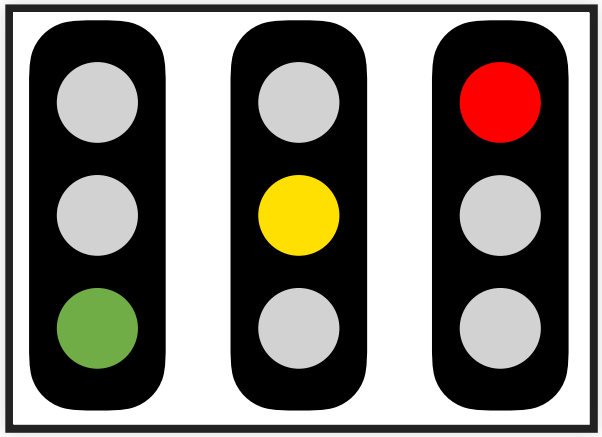
CHAPTER 4: ENUMS, WRAPPER AND AUTOBOXING

LEARNING OBJECTIVES

- Be able to explain what a Java Enum is and how it is defined
- Be able to explain and apply the wrapper concept in Java
- Be able to explain autoboxing and autounboxing

4.1 INTRODUCTION TO ENUMS

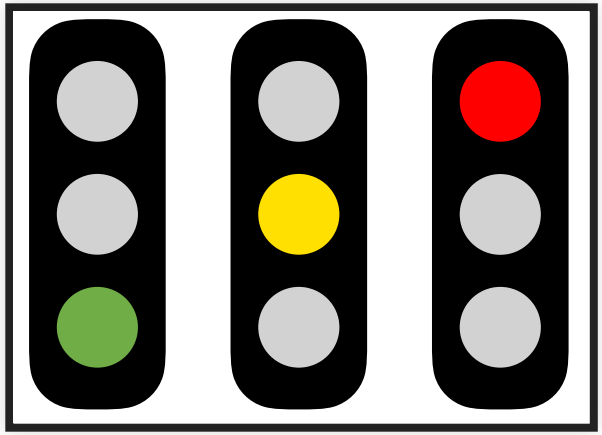
Example: Traffic Lights



A traffic light in Germany typically works through the following cycle:

RED → RED + YELLOW → GREEN → YELLOW → RED etc .

Example: Traffic Lights



To implement the control software, the following states might make sense: `INACTIVE`, `SHOW_RED`, `SHOW_RED_YELLOW`, `SHOW_GREEN`, `SHOW_YELLOW`, whereby `INACTIVE` indicates a state in which the traffic light does not run through the phase cycle.

How could this idea be implemented in Java?

SOLUTION WITH STATIC ATTRIBUTES

```
public class TrafficLightController {
    public static final int INACTIVE = 0;
    public static final int SHOW_RED = 1;
    public static final int SHOW_RED_YELLOW = 2;
    public static final int SHOW_GREEN = 3;
    public static final int SHOW_YELLOW = 4;

    private int state;

    public TrafficLightController(int state) {
        this.state = state;
    }
    public static void main(String[] args) {
        // start with red
        new TrafficLightController(TrafficLightController.SHOW_RED);
    }
}
```

How does it work?

SOLUTION WITH ENUMS

Enum(eration) types help to make source code more readable and maintainable.
Since Version 5, Java offers a concept for defining enum types.

```
enum TrafficLightState {
    INACTIVE, SHOW_RED, SHOW_RED_YELLOW, SHOW_GREEN, SHOW_YELLOW
}
```

```
public class TrafficLightController {
    private TrafficLightState state;

    public TrafficLightController(TrafficLightState state) {
        this.state = state;
    }
    public static void main(String[] args) {
        new TrafficLightController(TrafficLightState.SHOW_RED);
    }
}
```

Compare both solutions - what is different?

Enumeration types are data types with a finite set of values.

A Java Enum can be understood as a class that makes the enumeration values publicly accessible as static, constant attributes, e.g. `TrafficLightState.SHOW_RED`.

EXERCISE: INTRODUCITON TO ENUMS

Which statements are correct?

```
public enum TShirtSize {
    XS, S, M, L, XL, XXL, XXXL, XXXXL;
}
```

- [ ] Access a value with: TShirtSize.XL
- [ ] The enum TShirtSize is a special class.
- [ ] Enum values cannot be passed as parameters
- [ ] Access a value with: TShirtSize->M
- [ ] Attributes can use the enum TShirtSize as a data type.
- [ ] Access a value with: XS.TShirtSize

4.2 USING JAVA ENUMS

```
public class TraficLightController {
    private TraficLightState state;

    public TraficLightController(TraficLightState state) {
        this.state = state;
    }
}
```

The variable `state` is a reference to an Enum object of the type TrafficLightState. A specific enum always inherits from the special class Enum and therefore offers the following standard methods.

ENUM INSTANCE METHODS

- `name()`
  yields the name of the enum value
  Example: `state.name()` yields `"INACTIVE"`
- `ordinal()`
  yields the index (=position) of the enum value
  Example: `state.ordinal()` yields `0` since the compiler starts enumerating enum index values from 0

ENUM STATIC METHODS

- `values()`

  Each enum type offers a static method values(), which returns an array of the enum objects. Thus, it is easy to iterate over the values:

```
for (TraficLightState state : TraficLightState.values())
  System.out.printf("name: %s, index: %d", state.name(), state.ordinal());
```

- `valueOf(String name)`

  yields the enum instance corresponding to `name`

```
TraficLightState state = TraficLightState.valueOf("INACTIVE");
```

  Please bear in mind that an Exception is thrown when `name` is invalid!

DEFINE YOUR OWN METHODS IN ENUMS

Furthermore, it is also possible to extend enums with your own methods:

```
public enum Enum {
    VALUE1, VALUE2;

    public String toString() {
        return "Enum(" + name() + ")";
    }
    public static void main(String[] args) {
        Enum e1 = VALUE1;
        System.out.println("e1 = " + e1); // output: Enum(VALUE1)
    }
}
```

CONSTRUCTORS AND ATTRIBUTES IN ENUMS

It is also possible to use constructors and attributes in enums:

```
public enum colour {
    RED(0xff0000), YELLOW(0x00ff00), BLUE(0x0000ff);
    private int code;

    public colour(int c) {
        this.code = c;
    }
    public static void main(String[] args) {
        System.out.println(colour.YELLOW.code); // output: 65280
    }
}
```

TOSTRING AND EQUALS

Furthermore, the methods toString and equals (inherited from Object) are overwritten for enums:

- `toString()` returns the value of name(), i.e. the symbolic name.
- `equals()` performs a content comparison

Accordingly, the following lines have the same effect

```
System.out.println(obst1.equals(obst2));
System.out.println(obst1.ordinal() == obst2.ordinal());
```

EXERCISE: ENUM APPLICATION

Which statements are correct?

- [ ] Enums can also be realised via static attributes.
- [ ] Outputs with toString() are no different from other objects
- [ ] Enum values have a name and an index
- [ ] The enum TrafficLightState inherits from the special class Enum
- [ ] Enum values cannot be compared.
- [ ] The index of an enum starts at 1

4.3 WRAPPER CLASSES

Every primitive data type in Java has a matching class.

Primitive Data Type	Wrapper
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Wrapper classes have the same value range as the primitive data type.

While primitive data types can be used efficiently (memory, performance) for computations in Java, there are situations where it is necessary to use wrapper types (see chapter 6 - Generics)

MEMORY CALCULATION EXAMPLE

An `int` consumes 32 bits, whereas a `byte` consumes 8 bits.

For an object instance we need at least a header and housekeeping information, which in Java is about 64 bits.

An instance of `Integer` needs at least 32 bits for the value and 64 bits for meta data, i.e. 96 bits, for the instance - for a `Byte` this would be 72 bits.

To enable efficient memory management, memory requirements are rounded up to a multiple of 64 bits - so-called padding. Accordingly, both instances require 128 bits.

An int can be converted into an integer in two ways:

```
Integer i = Integer.valueOf(9);
Integer i = new Integer(9);
```

Notice: As with strings, wrapper instances are objects, which has an impact on comparisons:

```
new Integer(9) == new Integer(9)          // false
Integer.valueOf(9) == new Integer(9)      // false
Integer.valueOf(9) == Integer.valueOf(9) // true
```

4.4 AUTOBOXING / AUTOUNBOXING

From Java 5 onwards, the conversion of primitive values into wrapper objects and back again is carried out automatically by the compiler if required. The automatic conversion of a primitive value into a wrapper object is called autoboxing, the opposite direction is called autounboxing:

```
char c = 'a';
// autoboxing: c is embedded in a Character object
Character cWrapper = c;
// Autounboxing: char value from cWrapper is stored in c2
char c2 = cWrapper;
```

Wrappers and autoboxing / autounboxing are very important in connection with parameterisable classes (generics), see the corresponding chapter.

EXERCISE: WRAPPER CLASSES AND AUTOBOXING

Given the following arrays of type `int[]` and `Integer[]`.

```
// Array of the primitive data type int
int[] list1 = new int[] {1, 2, 3, 4, 128};
// Array of type Integer (wrapper class for int)
Integer[] list2 = new Integer[list1.length];

for (int i = 0; i < list1.length; i ++) {
    list2[i] = list1[i]; // Does this work?
}
```

Which statements are true?

- ☐ The assignment `list2 = list1;` does **not** work, as there is no wrapper functionality at array level
- ☐ The assignment `list2[i] = list1[i];` works, because here an int is converted into an integer wrapper object via autoboxing
- ☐ The assignment `list2[i] = list1[i];` works, as here an int is converted into an integer wrapper object via autounboxing
- ☐ The assignment `list2 = list1;` works, as here an `int` array is converted into a wrapper class array via autoboxing
- ☐ The assignment `list2[i] = list1[i];` in the loop does **not** work

The following comparison is carried out using `list1` and `list2`. Which statements are true?

```
if (list2[0] == Integer.valueOf(list1[0])) {
    System.out.println("Test 1 ok");
}
```

- ☐ `Test 1 ok` is output because `list2[0]` and `Integer.valueOf(list1[0])` are 1
- ☐ `Test 1 ok` is not output
- ☐ `Test 1 ok` is output as `list2[0]` and `Integer.valueOf(list1[0])` refer to the same reference

The following comparison is carried out using `list1` and `list2`. Which statements are correct?

```
if (list2[4] == Integer.valueOf(list1[4])) {
    System.out.println("Test 2 ok");
}
```

- ☐ `Test 2 ok` is output because `list2[4]` and `Integer.valueOf(list1[4])` are 128
- ☐ `Test 2 ok` is output because `list2[4]` and `Integer.valueOf(list1[4])` refer to the same reference
- ☐ `Test 2 ok` is not output

The following comparison is carried out using `list1` and `list2`. Which statements are correct?

```
if (list1[4] == list2[4]) {
    System.out.println("Test 3 ok");
}
```

- ☐ `Test 3 ok` is output because `list1[4]` and `list2[4]` refer to the same reference - autoboxing occurs here
- ☐ `Test 3 ok` is output because `list1[4]` and `list2[4]` are 128 - an autounboxing occurs here
- ☐ `Test 3 ok` is not output

4.5 EXERCISE

DOG BREED ENUM

In the following multi-part exercise, create a simple enumeration type 'dog-breed' with its own constructor, attributes and methods as well as a class 'dog' that uses this enumeration type.

Unzip dog-enum.zip and import the folder as "Existing Maven Project".

EXERCISE: CREATE AN ENUM

Implement an enum 'dog breed' with values for different dog breeds:

Golden Retriever

Dalmatian

Beagle

Dachshund

Pug

Please be aware: The general Java convention uses capital letters and underscores for the names of values of an enum - as with constants.

EXERCISE: ENUM ATTRIBUTES, METHODS AND CONSTRUCTOR

Extend the enum dog breed by:

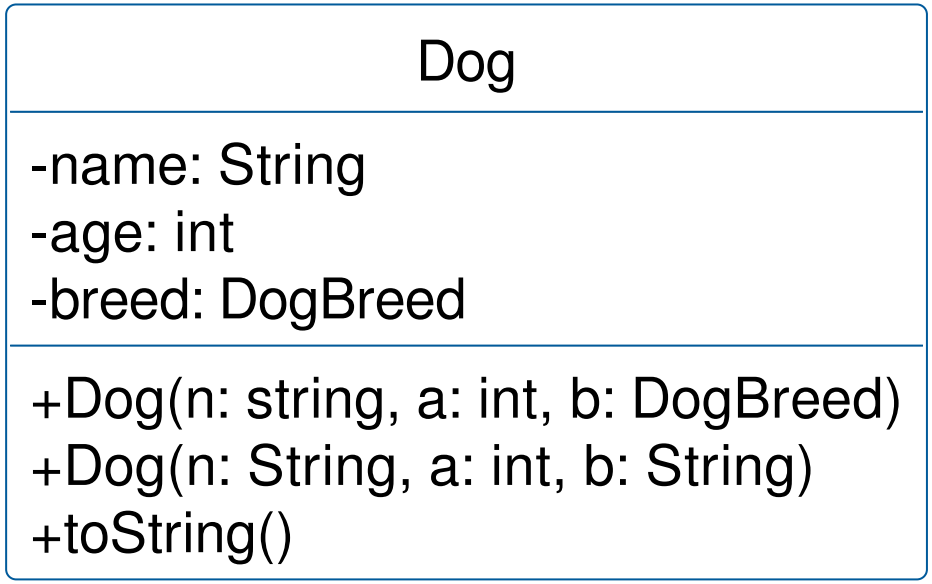
- an attribute `height` for the average height of the breed
  - Golden Retriever 0.61
  - Dalmatian 0.60
  - Beagle 0.41
  - Dachshund 0.30
  - Pug 0.29
- a constructor that takes the height as a parameter and assigns it to the attribute
- a getter to return the height

Which statements about enum attributes, methods and constructors are correct?

- [ ] The constructor of the enum with the attribute `height` must be called with the keyword `new
- [ ] Each enum is an instance of the enum class type and has the defined properties (attribute height, method getHeight())
- [ ] Custom methods in an enum are not public
- [ ] The constructor of an enum is not Public

EXERCISE: USING ENUMS

Implement the following class `Dog`.



Implementation Hints

- Create a new class `Dog` that has a name of type `String`, an age of type `int` and a breed of type `DogBreed` as attributes.
- Add a constructor that expects name, age and breed as parameters.
- Add another constructor that also expects name and age, but requires the breed to be of type string. Convert breed to an instance of `DogBreed` and call the first constructor.
- Add the method `toString()`, which outputs the following:
  `<name> is <height> metres tall and <age> years old.`

Use the following main method for testing.

```
public class Main {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        System.out.print("Name: ");
        String name = scanner.nextLine();
        System.out.print("Breed: ");
        String breed = scanner.nextLine();
        System.out.print("Age: ");
        int age = scanner.nextInt();
        System.out.println(new Dog(name, age, breed));
    }
}
```