brightboost

DESIGN PATTERNS AND PRINCIPLES

OCP DAY 2 AND 3

brightboost

CONTENT

- Interfaces
- Functional programming
- Lambdas
- Polymorphism
- Design principles: encapsulation, JavaBeans, is-a, has-a, composing objects
- Working with design patterns: singleton, immutable objects, builder pattern, factory pattern
- Cucumber

brightboost

INTERFACES

- Abstract data type, similar to a class
- It may include:
  - Constants (public static final fields)
  - Abstract methods
  - Default methods > implicitly public, provide functionality to implementing classes, without breaking them
  - Static methods > available in classes not implementing the interface as well
- Interfaces can extend other interfaces
- Classes implement interfaces (implementing more than one is allowed)

brightboost

WILL THIS COMPILE?

```
public interface Test {
        static void test(){};
}
public class Tester extends Test {
}
```

brightboost

WILL THIS COMPILE?

```
public interface Test extends Tester{
        static void test(){
        }
}
public class Tester {
}
```

brightboost

WILL THIS COMPILE?

```
public final interface Test {
        static void test(){
        }
}
public class Tester implements Test {
}
```

brightboost

WILL THIS COMPILE?

```
public interface Test {
        void test(){}
}
public class Tester implements Test {
        @Override
         public void test(){
                System.out.println(“testing”);
        }
}
```

brightboost

WILL THIS COMPILE?

```
public interface Test {

        static void test(){}

}

public class Tester implements Test {

}
```

brightboost

TYPES OF INTERFACES

- Marker interface: no methods or fields
- Functional interface: an interface with exactly one abstract method
  - Use @FunctionalInterface to ensure that there is only one abstract method, this way it can be avoided that unintentionally a method is added and that it can no longer be used for lambda expressions (coming soon). It is not required to add it, but it's considered a best practice

brightboost

EXERCISE INTERFACES

Create a class Animal, and some subclasses like Tiger, Bird, Deer, Rabbit, Snake, Insect. Add some useful fields in Animal and the subclasses.

Create two interfaces:

Predator (method catchMeal(Prey p)) and Prey (method getAway()).

Also add a method with implementation hide() in prey. Should this be static or default, and why?

Implement the interfaces in the classes that are prey/predator or both

brightboost

EXERCISE INTERFACES PART II

Add to both interfaces the default method run();

What is happening? Why? And how to fix this?

brightboost

EXERCISE INTERFACES PART III

Add a method with implementation to prey. This method should be called calculateCalories, take the weight of the animal, and the average kcal per kg. It should then return an int with the kcal the prey animal is.

Should this method be static or default, why?

brightboost

FUNCTIONAL PROGRAMMING

- Computations are written as functional programming functions: mathematical functions evaluated in expression context
- Declarative: logic described without flow control, focus is not on how to do it, but on what needs to be done
- One way of doing that: lambda expression that uses functional interface

brightboost

LAMBDA EXPRESSIONS

- Use functional interfaces
- Anonymous method that implements functional interface
- Can be passed around like variables to a method

brightboost

LAMBDA EXPRESION SYNTAX

(n) -> n*n

Left: input parameter, no () needed when it's only one without an explicit type

Arrow operator: divides lambda expression in two parts, can be read as "becomes", so "n becomes n*n"

Right: method body that will be returned, no {} needed when it is only one statement

brightboost

IMPLEMENTING FUNCTIONAL INTERFACE I

```
@FunctionalInterface
    public interface Exercise {
                public void run(int km);
}

public class Person implements Exercise {
    public void run(int km) {
                System.out.println("running " + km + " kms");
    }
}
```

Exercise: rewrite to lambda expression

brightboost

EXERCISE LAMBDA EXPRESSION

- Create a class Cat that uses the functional interface CheckHunter
- The functional interface CheckHunter should have a method called “test” that returns a boolean and takes the cat as parameter
- The cat should have a method canCatchSpider
- The cat should have properties the maxSpeed and agilityScore. If maxSpeed is higher than 5 and agilityScore is higher than 75, the method canCatchSpider should return true, else it should return false

- In a new class UsefulCats, create a method that checks how useful a cat is. This method should have as input a cat and a CheckHunter object that is implemented using the method can catch spider on the cat. Then make a main method in which you call this method using a lambda expression to implement the test method in CheckHunter to check how much of a hunter your cat is

brightboost

LAMBDA EXPRESSIONS AND PREDICATE

- Java has many functional interfaces for common formats
- One of these is the Predicate interface

```
public interface Predicate<T> {
    public boolean test(T t);
}
```

Exercise: rewrite the previous exercise using the Predicate interface

brightboost

WILL THESE LAMBDA EXPRESSIONS (IN THE CORRECT CONTEXT) COMPILE?

1. `n -> { return n*n; }`
2. `Animal a -> return a.trueOrFalse();`
3. `() -> new Animal()`
4. `x,y -> x + y`
5. `(x, y) -> x + y`
6. `(int x, int y) -> x + y`
7. `(int y) -> {return;}`
8. `a -> {return a*a}`
9. `(int x, y) -> x*y`
10. `(int x) -> {int x = 3; return x;}`

brightboost

POLYMORPHISM AND INTERFACES

- Single interface can support multiple forms

brightboost

```
public interface Cardio { public void run(int km);}

public class HealthyPerson implements Cardio {
    public void run(int km) { System.out.println("run " + km + " km");}
}
public class UnhealthyPerson implements Cardio {
    public void run(int km) { System.out.println("I don't run " + km + " km");}
}


public class Gym {
    public void treadmillExercise(Cardio person, int km) {
                person.run(km);
    }
    public static void main(String[] args) {
                Gym gym = new Gym();
                gym.treadmillExercise(new HealthyPerson());
                gym.treadmillExercise(new UnhealthyPerson());
    }
}
```

brightboost

POLYMORPHISM EXERCISE

Use your own hobby / sport for this to show you understand the benefits of polymorphism

- Create a functional interface
- Create a class (A) that implements the functional interface
- Create another class (B) that implements the functional interface differently
- Create a class that can use both created classes
- Create a method in this last class that takes a parameter of type functional interface
- Call this methods twice with your (A) and (B)

brightboost

OBJECT VS REFERENCE

- You access objects via the reference and never directly the memory location. The reference can be a superclass of the actual object in memory
  - Objects can be accessed via their reference, so when an object has a superclass, we can only access it via the properties of the superclass and not the properties of the object specifically
  - Changing the reference changes which properties you can access, but it doesn't change the properties the object has
  - Casting: by casting the object to the type of the reference, you get access back to the specific properties

Exercise: change the type in your previous example to the superclass, and solve the new problem by casting it back to the object of the reference type

brightboost

DESIGN PRINCIPLES

- Best practices for software design
- Following design principles will lead to:
  - Logical and structured code
  - Readable and understandable code
  - Reusable code
  - Maintainable code that can easily be modified
- Often related to how to structure classes to represent the data model best

brightboost

ENCAPSULATION

- Methods operate on data
- Data belongs to the class
- Implemented by:
  - Private instance members
  - Public methods (getters/setters)
- Allows invariants: property that is always true about the class, and data can only be altered if the conditions are met

brightboost

JAVABEANS

Name for classes that store data using the the design principles of encapsulation:

- Private properties
- Getters for non-boolean properties are called getProperty
- Getter for boolean properties are called isProperty or getProperty
- Setter methods are called setProperty

brightboost

WHAT NEEDS TO CHANGE HERE BASED ON JAVABEAN NAMING CONVENTION?

```
public int getAge(){
        return age;
}

public boolean getBold(){
        return boldness;
}

public Boolean isParent(){
        return parent;
}
```

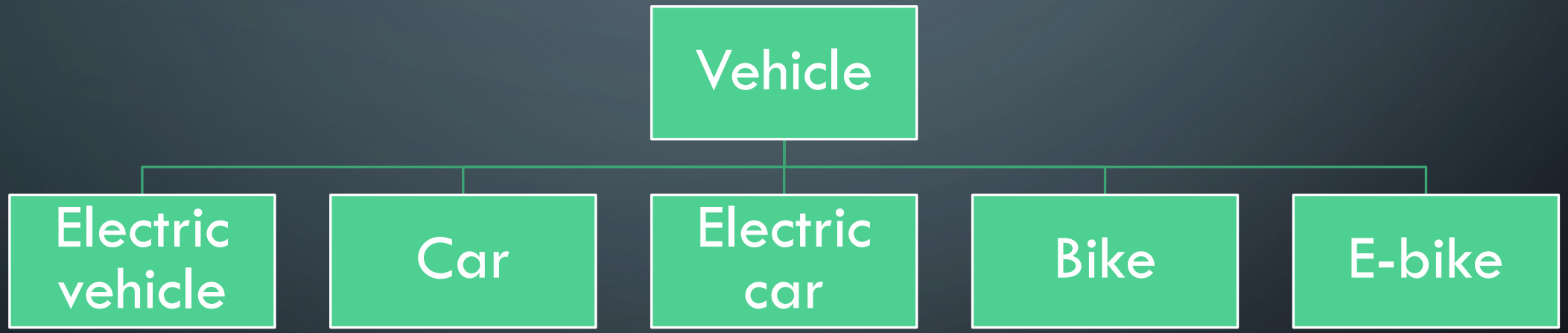
brightboost

IS-A

- When an object is an instance of a data type, this relationship between object and datatype is called an is-a relationship
- When instanceof would return true, there is an is-a relationship
- Good design models real world is-a relationships

brightboost

IS THIS A GOOD DESIGN?



brightboost

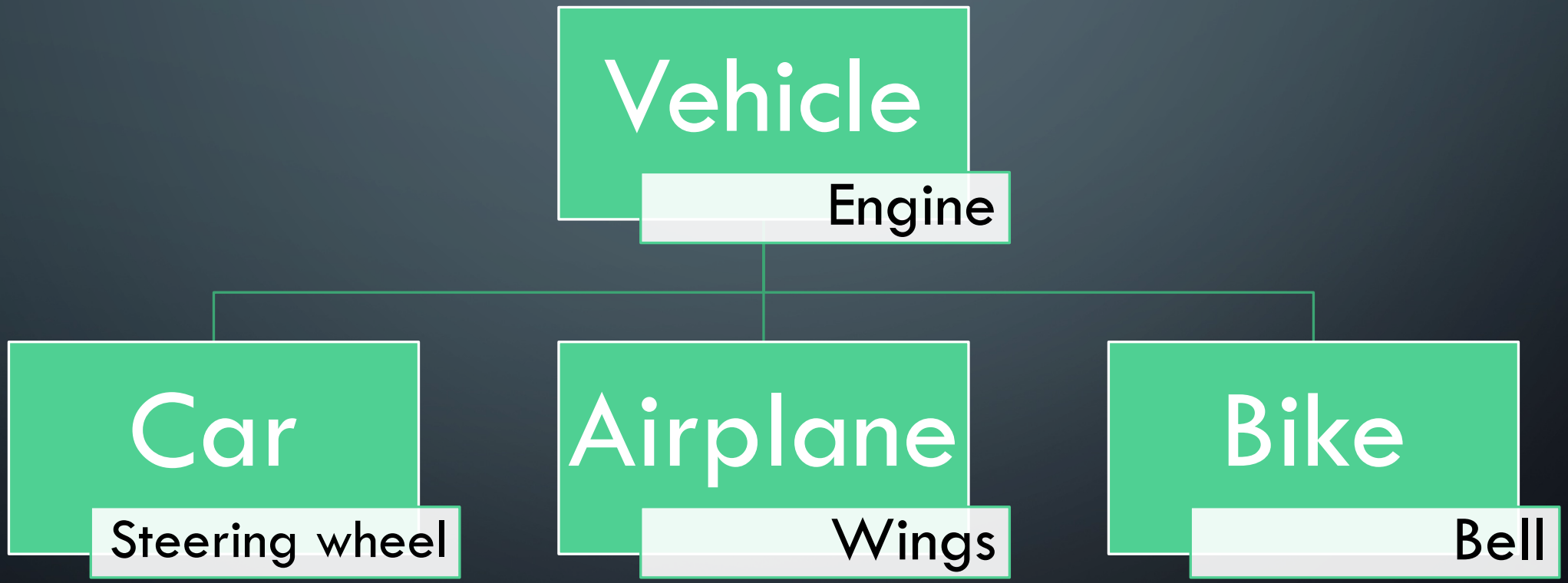
HAS-A

- If an object contains a property, there is a has-a relationship
- If the parent of an object contains a property, the child also has a has-a relationship with that property (not if the property is private)

- Aggregation: has-a relationship with another class

brightboost

GOOD DESIGN?



brightboost

AGGREGATION

```
public class Person{
        private Job job;
        private String name;
        public Person(){
            this.job = new Job();
            this.job.setHours(36);
        }
        etc…

}
```

```
public class Job{
    private String function;
    private int hours;

     public getFunction(){
            return function;
     }
     etc….
}
```

brightboost

COMPOSING OBJECTS

- Object composition is contructing a class with reference to another class in order to use that functionality
- Mutually dependent, part-of relationship
- Alternative for inheritance, but you should not use it when:
  - There is a clear is-a relationship
  - You want to override methods
- Make code very reusable

brightboost

COMPOSITION

```
public class Car{
        private final Engine engine;
        private String brand;
        public Car(int x){
            this.engine = new Engine();
            this.engine.setHp(x);
        }
        etc…

}
```

```
public class Engine{
    private int hp;
    etc;

     public getHp(){
            return hp;
     }
     etc….
}
```

brightboost

DESIGN PATTERNS

- Solutions for common software development problems
- Types of design patterns:
  - Creational: deal with object creation >> discussed here
  - Structural: deal with relationships between entities
  - Behavioral: deal with communication patterns among objects
  - Concurrency: deal with multi-threading issues
- For OCP:
  - Singleton pattern
  - Immutable objects

brightboost

SINGLETON

- Pattern to create only one object in memory that is shared by all classes using it
- Centralizing data and/or a specific functionality available for all threads, improve performance because it only needs to create the class and load the data once
- How to achieve this: make a class that can only have one instance in the memory

brightboost

SINGLETON OPTION I – CREATE OBJECT DIRECTLY IN DEFINITION OF INSTANCE

```
public class OnlyOne{
        private static final OnlyOne instance = new OnlyOne();
        private OnlyOne(){}

        public static OnlyOne getInstance(){
                return instance;
        }
}
```

brightboost

WHAT IS WRONG WITH THIS?

```
public class ChildOnlyOne extend OnlyOne {
        private static final ChildOnlyOne instance = new ChildOnlyOne();
        private ChildOnlyOne(){}

        public static ChildOnlyOne getInstance(){
                return instance;
        }
}
```

brightboost

SINGLETON OPTION II – CREATE OBJECT WITH STATIC BLOCK

```
public class OnlyOne{
        private static final OnlyOne instance;
        static {
                instance = new OnlyOne();
        }
        private OnlyOne(){}

        public static OnlyOne getInstance(){
                return instance;
        }
}
```

brightboost

SINGLETON OPTION III – LAZY INSTANTIATION

```
public class OnlyOne {

        private static OnlyOne instance;

        private OnlyOne(){}


        public static synchronized OnlyOne getInstance(){
                if(instance == null) {
                        instance = new OnlyOne();
                }
                return instance;
        }
}
```

brightboost

SINGLETON EXERCISE

- Code the class SantaClaus and make it implement the singleton pattern, since there is only one santa.

brightboost

IMMUTABLE OBJECTS

- Pattern that creates a read-only object that can be shared by different classes
- Share object amongst classes, but without that the value can be edited and without having to make many copies of the same class whilst avoiding concurrency issues
- Create objects whose state doesn't change and that can be easily shared

brightboost

STRATEGY TO CREATE IMMUTABLE OBJECTS

1. Constructor to set all the properties
2. Private and final instance variables
3. No setter methods
4. Referenced objects in immutable objects cannot be modified or accessed directly
5. Avoid overriding

brightboost

WHAT'S WRONG WITH THIS IMMUTABLE OBJECT?

```
public final class ImmutableObject {
        private final List<Animal> maaikesPets;
        public ImmutableObject(List<Animal> maaikesPets){
                this.maaikesPets = maaikesPets;
        }

        public List<Animal> getMaaikesPets(){
                return maaikesPets;
        }
}
```

brightboost

WHAT'S WRONG WITH THIS IMMUTABLE OBJECT?

```
public class ImmutableObject {
        private final Animal maaikesFavoritePet;
        public ImmutableObject(Animal maaikesFavoritePet){
                this. maaikesFavoritePet = maaikesFavoritePet;
        }

        public String getMaaikesFavoritePet(){
                return maaikesFavoritePet.getName();
        }
}
```

brightboost

IMMUTABLE OBJECT EXERCISE

Create an immutable shopping list

The shopping list should contain a list of items that need to be bought

The shopping list needs to have a date

brightboost

WHAT IS THE MOST COMMON IMMUTABLE JAVA OBJECT?

And what happens if you assign a new value to it?

brightboost

BUILDER PATTERN

- Pattern for instantiating a lot of values on object creation
- Contructor may grow to contain many input parameters. You could use setters after creating objects, but not an option for immutable objects. And it could also break invariants (temporarily), because one value can be set which requires another one to be set as well.
- Parameters are passed to a builder object, usually via method chaining, and object is actually created with a call to the build method

brightboost

BUILDER PATTERN EXERCISE

Create a new class Animal, and give it 10 properties

Create a new class AnimalBuilder and implement the builder pattern

Create another class from which you create a new Animal using the AnimalBuilder

brightboost

FACTORY PATTERN

- Create objects of which the exact type is not known until runtime
- The exact (sub)class of the object can be selected at runtime
- Based on the input parameters the exact object is being created from the factory class >> allowing class polymorphism
- Implemented using static methods that return objects
- Classname ends with -Factory

brightboost

```
public class ShapeFactory {


public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();


      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();


      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }


      return null;
   }
}
```

brightboost

FACTORY PATTERN EXERCISE

- Create a PetFactory with at least three different animals
- Animals should have a method hello(), this method should print “hello *animaltype*”
- Use the PetFactory from a main method for the different animals
- Check if the output is correct

brightboost

CUCUMBER

- Testing tool
- Test automation
- BDD
- Gherkin: software language in which the behavior of the software is described in a way that it is understandable for non-technical people as well
- Gherkin files -> .feature extension
- Gherkin: White space has a meaning, unlike Java

brightboost

CUCUMBER – BUILDING BLOCKS – FEATURE

- Feature -> use case

Feature: Test Gherkin Name

Some description as documentation to describe behavior. This goes on until scenario starts.

Scenario: Developers want to learn cucumber

……

brightboost

CUCUMBER – BUILDING BLOCKS - SCENARIO

- Scenario -> flow of events with an expected result

Scenario: Developers want to learn cucumber

Given developer is “smart”

When they have had a training in cucumber

Then they should “understand cucumber”

brightboost

CUCUMBER – BUILDING BLOCKS - STEPS

- Steps -> building blocks of scenario's. First word of a step is a keyword, most common: given, when, then, and, but

- Steps need to be implemented

```
public class Stepdefs {
    private String developer;
    private String expectedAnswer;
@Given("^developer is smart$")
    public void developers_is_smart(){
        developer = "smart";
    }
    @When("^they have had a training in cucumber$")
    public void they_have_had_a_training_in_cucumber() {
        actualAnswer = UnderstandCucumber. UnderstandCucumber(developers);
    }

    @Then("^I should be told \"([^\"]*)\"$")
    public void they_should_understand_cucumber() {
        assertEquals(expectedAnswer, actualAnswer);
    }
}
```

```
class UnderstandCucumber {
    static String UnderstandCucumber(String developer)
{
        if(developer.equals("smart"){
            return "understand cucumber";
        else{
            return "don't understand";
        }
    }
}
```

brightboost

CUCUMBER – GET STARTED

- https://cucumber.io/docs/guides/10-minute-tutorial/