JAVA FUNDAMENTALS & CLASS DESIGN

OCP DAY 1

JAVA FUNDAMENTALS & CLASS DESIGN

OCP DAY 1

brightboost

INTRODUCTION ROUND

- Name
- Job / team / background
- Home situation
- Hobbies and passions
- What do you hope to take from this course?

brightboost

WHAT ARE WE GOING TO DO?

- Get OCP11 (or 17?) certified!
- How are we going to do it? By having fun with Java!
  - Theory most mornings
  - Case most of the afternoons
- Topics we'll cover: fundamentals, class design, design patterns, annotations, generics and collections, functional programming, exceptions, localization, concurrency, IO, NIO2, JDBC, modular programming, secure coding and how to prepare for the exam

brightboost

TODAY'S SCHEDULE

- Test to see where you are standing with Java
- Quick recap
- instanceof
- Invoking virtual methods
- Annotation: @Override
- Inheritance
- Overriding equals, hashcode and tostring
- Enums
- Nested classes
- Interface members
- Basics of functional programming

brightboost

CATCH THE MOUSE!

- Download this project: https://github.com/BrightBoost/catch-the-mouse
- Do the exercises from tasks.md
- Run unit tests to see if you solved the problems!

brightboost

ACCESS MODIFIERS

- Private
- Default (no keyword)
- Protected
- Public

brightboost

ACCESS OF DIFFERENT MODIFIERS

	Private	Default	Protected	Public
Member in same class	X	X	X	X
Member in another class, same package		X	X	X
Member in another class that inherits from class, in another package			X	X
Member in a class in another package without inheritance				X

brightboost

QUESTION: WILL THIS COMPILE?

```
package accessmodifiers.subpackage;

import accessmodifiers.ClassA;

public class ClassC extends ClassA {
    public ClassC(String name) {
        super(name);
        sayName(); //protected in classA
    }
    public static void main(String[] args) {
        ClassA cA = new ClassA("Maria");
        cA.name = "Emmy";
        cA.sayName(); //protected in classA
        cA.countName(); //default in classA
        cA.shoutName(); //public in classA

    }
}
```

brightboost

QUESTION: WILL THIS COMPILE?

```
package accessmodifiers.subpackage;

import accessmodifiers.ClassA;

private class ClassC extends ClassA {
    public ClassC(String name) {
        super(name);
        sayName();
    }
    public static void main(String[] args) {
        ClassA cA = new ClassA("Maria");
        //cA.name = "Emmy";
       // cA.sayName();
        // cA.countName();
        cA.shoutName();

    }
}
```

brightboost

OVERLOADING

- Same method name, but different parameter list

- Rules:
  - Return type may be different, but just that is not overloading
  - Overloaded method can be static / non-static, but just that is not overloading
  - Java looks for the closest match to the call:
    - Exact match
    - Matching superclass
    - Converting to larger primitive
    - Converting to an autoboxed type
    - Varargs

brightboost

OVERRIDING

- Same name and signature, but in a different class with its own implementation

- Rules:
  - Access modifier is the same or less restrictive
  - Return type same or more restrictive
  - If checked exceptions are thrown, only same or subset of these are allowed to be thrown

brightboost

@OVERRIDE

- Indicate explicitly that a method is being overridden
- Useful, because the compiler notices when it's not overriding something

- Can be used in three situations:
  - Override method from an interface
  - Override method from a superclass
  - Override a method that is in the Object class (such as toString, equals and hashcode)

brightboost

QUESTION: IS THIS OVERRIDING?

```
public class OverrideExercise extends OtherClass {

    @Override
        Public static void show(){
                System.out.println("tada");
        }
}


public class OtherClass {
        public static void show(){
                System.out.println("TADA");
        }
}
```

brightboost

QUESTION: HOW TO MAKE THIS COMPILE?

```
public class OverrideExercise extends OtherClass {

    @Override
        Public static void show(){
                System.out.println("tada");
        }
}


public class OtherClass {
        public static void show(String text){
                System.out.println(text);
        }
}
```

brightboost

OVERLOADING EXERCISE

- Create a class named 'PrintStuff' to print various things of different datatypes by creating different methods with the same name having a parameter for each datatype.
- Make sure to use
  - Primitive type char, int, byte
  - An array
  - Wrapper classes
  - Varargs
- Call from another class (Main for example) with different parameters and see which version is being called

brightboost

OVERRIDING EXERCISE

- Create a class ‘Certificate’ with a method ‘printCertificate' that prints "I got a certificate". It has two subclasses (child) namely ‘OCA' and ‘OCP' each having a method with the same name the right certificate. Call the method by creating an object of each of the three classes.
- Call the method by:
  Certificate c = new OCA(); c.printCertificate; >> what do you think this will print?
- Can you call the method in the parent class from a non-static method in the child class?

brightboost

ABSTRACT CLASSES

- An abstract class cannot be instantiated
- When a class has an abstract method, the entire class needs to be abstract
- An abstract class does not need to have an abstract method
- Concrete (non abstract) child classes must implement all abstract methods from the parents and grandparents or interfaces

brightboost

HOW TO MAKE THIS CODE COMPILE? – COME UP WITH 3 OPTIONS

```
public abstract class Vehicle {

        abstract void start();

}

public class Car extends Vehicle {


}
```

brightboost

STATIC KEYWORD

- Static can be applied to: variables, methods, blocks and inner classes

- Variable: all instances of that object share the value of that member, it is only stored in one place
- Methods: belongs to all instances of the class, can be accessed without an instance of that clas and can only access static members from outside
- Blocks: use to instantiate static fields and it is executed on class loading before the main method
- Nested classes: can access static members from outer class

brightboost

WILL THIS COMPILE?

```
public ClassA extends ClassWithDoSomethingMethod {

        static void doSomething(){

                super.doSomething();

        }

}
```

brightboost

WILL THIS COMPILE?

```
class ClassA {
 int a = 40;

 public static void main(String args[]){
        System.out.println(a);
 }
}
```

brightboost

FINAL KEYWORD

- Final can be applied to: variables, methods, classes

- Variables: can only be given a value once, so for constants
- Methods: can not be overridden
- Classes: cannot have subclasses

brightboost

WILL THIS COMPILE?

```
public final abstract ClassA {
        void doSomething(){
                System.out.println("pretend to be busy");
        }
}
```

brightboost

INSTANCEOF

- Operator (comparison operator)
- Used to test whether a certain object is an instance of a certain type
- Return true for same class, subclass of, and when an interface is implemented

- Exercise: create a class named TestInstanceof. Add a main method to it, create a new instance of a class, and test whether it is an instance of that classes superclass and a class that it is not an instance of.

brightboost

VIRTUAL METHOD INVOCATION

- Virtual method: regular non-static method
- The exact method that gets invoked by a call depends on the type of the instance it gets called on

brightboost

WHAT DOES THIS DO?

```
public class A {
        void print(){
                System.out.println("print A");
        }

        public static void main(String… args) {
                A a = new B();
                a.print();
        }
}
```

```
public class B extends A {
        void print(){
                System.out.println("print B");
        }
}
```

brightboost

OVERRIDING METHODS FROM JAVA.LANG.OBJECT

- All Java classes inherit from java.lang.Object
- Methods in java.lang.Object can be overriden
- Common for overriding are:
  - toString: useful for giving a human readable representation of the object
  - equals: method that tests whether another object is equal to the current object
  - hashCode: puts instance of a class in a category, used for storing object as a key in maps

brightboost

TOSTRING

Used for printing something useful about the object. For your own classes you need to override this. For many standard Java classes this has been done.

```
@Override
public String toString() {
    return “some sort of useful string”;
}
```

Exercise: override toString in your own class and print this object (without calling toString)

brightboost

EQUALS

- Equals method is used to compare two objects

- Should return true if:
  - Object 1 is an instance of object 2
  - Object 1 is not null
  - Other requirements are met

EQUALS EXAMPLE

```
@Override
    public boolean equals(Object o) {

        if (o == this) return true;
        if (!(o instanceof OtherObject)) {
            return false;
        }

      OtherObject oo = (OtherObject) o;

       return oo.stringVar.equals(stringVar) &&
               oo.intVar == intVar &&
               oo.other.equals(other);
    }
```

brightboost

HASHCODE

- Should be overridden as well if equals has been overridden
- If two objects are equal, hashcode should be equal
- If two objects are not equal, hashcode does not need to be equal
- Value of hashcode can only change if a property that is in equals also changes

HASHCODE EXAMPLE

```
@Override
public int hashCode() {
    int result = 17;
    if (stringVar != null) {
        result = 31 * result + stringVar.hashCode();
    }
    if (intVar != null) {
        result = 31 * result + intVar.hashCode();
    }
    if (other != null) {
        result = 31 * result + other.hashCode();
    }
    return result;
}
```

brightboost

ENUMS

- Special class to capture group of constants

```
enum Season {
  SPRING,
  WINTER,
  FALL,
  SUMMER
}
```

Exercise: print all the values of this enum

brightboost

ENUMS WITH CONSTRUCTORS, FIELDS AND METHODS

- Constructor: always private, and only executed after first call to a value, after that all values have been contructed already
- Fields: can be added per constant and set with the constructor
- Methods: any method can be there and called by e.g. Season.SUMMER.getAverageTemp()

brightboost

EXERCISE ENUM

- Create an enum with days of the week. For each day there needs to be a special greeting. And it needs to store whether it is a weekend day or not.
- In the constructor the greeting and weekend variable need to be set
- There need to be methods to get the greeting and to get whether it is a weekend day or not

brightboost

NESTED CLASSES

- A nested class is a class in a class
- Usually not a best practice, especially not the cases you will need to know for OCP exam

- Exercise: find best practice examples

brightboost

MEMBER INNER CLASS

- Non-static inner class that is at the level of instance variables
- Can have all access modifiers
- Can extend any class
- Can be abstract or final, cannot be static
- Cannot have static fields or methods
- Can access all member of outer class

- Exercise: create an outer class, with a member inner class. And create this inner class from a main method. Call a method from the inner class.

brightboost

LOCAL INNER CLASS

- An inner class that is a nested class that is defined in a method
- A local inner class does not exist until the method is invoked (just like local variables)
- And it is eligible for garbage collection as soon as the method ends
- They don't have an access modifier
- Cannot be static or have static fields/methods
- All fields and methods of outer class
- No access to local variables, except for final or effectively final ones

ANONYMOUS INNER CLASS

- Local inner class without a name

```
public class Outer{
        abstract class Anonymous{
                abstract int random();
        }
        public void test(){
                Anonymous a = new Anonymous() {
                        int random() { return 8; }
                };
        }
}
```

Exercise: change this example to use an interface instead of abstract class

brightboost

STATIC NESTED CLASSES

- Static class that is defined at the same level as static variables
- Regular class, except for:
  - Can use all access modifiers
  - Outer class can use all fields and methods of static nested class, also the private ones

brightboost

INTERFACE MEMBERS

- Constants (implicitly public, static, final)
- Methods with no body (implicitly public, abstract)
- Methods with a body
  - Default (implicitly public, overriding is optional)
  - Static (implicitly public)
  - Private
  - Private static

brightboost

QUESTIONS:

- Why have default methods in interfaces?
- What if a class implements two interfaces with the same default method signature?
- Why would you want a private method in an interface?

brightboost

EXERCISE INTERFACE

- Create an interface with an abstract method calculate that takes an argument int and returns a double
- Default method saySomething
- Static method alwaysRight returning true, that calls some private method that prints the current time
- And a constant

brightboost

BASICS OF FUNCTIONAL PROGRAMMING - LAMBDA

Why would you use lambda expressions?

- Readable simplified efficient code
- Stream API
- Functionality as an argument for methods

brightboost

EXERCISE FUNCTIONAL INTERFACE

- Check if your interface from last exercise is a functional interface
- If necessary make adjustments

brightboost

FUNCTIONAL INTERFACES

- Interfaces with only one abstract method
- Abstract methods with the signature of the ones in Object class don't count for the one abstract method (since they are always implemented): toString, equals, hashCode
- Can have methods with body, but only one abstract method
- They should be annotated with @FunctionalInterface
- Built-in functional interfaces

brightboost

LAMBDA EXPRESSION

- Implement a functional interface
- (parameters) -> (body)

- Easy example:
- x -> x + 2

- Read the arrow as “becomes”:
- x becomes x + 2

brightboost

EXERCISE LAMBDA

- Implement your interface using a lambda expression
- Create a method calculate that takes the interface type as an argument, and calls the method on the interface from there
- Call the method using a lambda
- Bonus: try adjusting the method in your functional interface, see what happens if it returns void, int String and if it doesn't take arguments vs taking arguments