# Java Training

Day 6: Short Coding Exercise, Scope, Constructors & Accessors

Download today’s slides:
go/java+espresso-training/day6

## Returning briefly to equality between Bobs....

```
public class Scoping {
        public static void main(String[] args){
                Bob bob1 = new Bob();
                Bob bob2 = new Bob();
                bob1.age = 4;
                bob2.age = 4;
                System.out.println(bob1.equals(bob2));
        }
}

class Bob {
        int age;
        public boolean equals(Object object){
                return (object instanceof Bob &&
                         this.age == ((Bob)object).age);
        }
}
```

Clearer to more experienced programmers, but maybe not to beginners

```
> javac Scoping.java
> java Scoping

true
```

## Rewrite of `Bob` to make `equals()` (hopefully) easier to follow

```
class Bob {
        int age;

        public boolean equals(Object x){
                if ( !(x instanceof Bob) ){
                        return false;
                } else {
                        Bob b = (Bob)x;
                        return (this.age == b.age);
                }
        }
}
```

## Calculator Solution with method calls, *Coding Exercise*

```
public class Calculator {
    public static void main(String[] args){
        if (args.length != 3) {
            System.out.println("Error: wrong number of arguments");
        } else {
            int x = Integer.valueOf(args[0]);
            int y = Integer.valueOf(args[1]);

            switch(args[2]) { // Gary’s switch from the Slack channel
                case "+":
                    System.out.println( add(x,y) ); // System.out.println(x+y)
                    Break;

         . . .
    }
    static int add(int x, int y){
        return (x + y);
    }
}
```

Why `static`?

Update your Calculator application so that it calls static methods: `subtract()`, `multiply()` and `divide()` rather than doing the calculations in-line.

An implementation for `add()` is given

## Scope & Variable **Shadowing**

Member variables may be *shadowed* by local variables

```
public class Scoping {
        public static void main(String[] args){
                Bob bob1 = new Bob();
                bob1.age = 34;
                bob1.doSomething();
        }
}

class Bob {
        int age;
        void doSomething() {
                int age = 5;
                System.out.println("my age is " + this.age);
        }
}
```

```
> javac Scoping.java
> java Scoping

my age is 34
```

Use the `this` reference to explicitly reference a member variable

Without `this`, it will output `5`

# On brief word on **Scope** and **Indentation**

Scopes are defined by Curly Brackets...
...but indentation allows you to keep your sanity

```
{scope 1 {scope 2 {scope 3}} {scope 4 {scope 5 {scope 6}}}}

                  |---- scope 3 --|                         |------ scope 6 -------|

                                       |----------- scope 5---------------------|

         |----------- scope 2-----------------|  |------------------------------scope 4 ----------------------|

 |------------------------------------------------ scope 1 --------------------------------------------------------|
```

## Bob's **Constructor**

It's convenient to set the state of an instance when it's created

```
public class Constructor {
        public static void main(String[] args){
                Bob bob1 = new Bob(64);
                System.out.println(bob1.age);
        }
}

class Bob {
        int age;
        public Bob(int age){
                this.age = age;
        }
}
```

A **constructor** can have as many **arguments** as you like and you may have as many constructors as you have **unique argument lists**

```
> javac Constructor.java
> java Constructor

64
```

## *Deluxe* `Bob`, with **Constructor** and `toString()` override

```
public class Constructor {
        public static void main(String[] args){
                Bob bob1 = new Bob(64);
                // bob1.age = 22;   - compile error
                System.out.println(bob1);
        }
}

class Bob {
        private int age;  // access limited to Bob methods
        public String toString(){
                return "Bob: "+age+" yrs";
        }
        public Bob(int age){
                this.age = age;
        }
}
```

```
> javac Constructor.java
> java Constructor

Bob: 64 yrs
```

How do you get a Bob’s age as a simple `int`?

## Accessors for Bob’s age

```
public class Constructor {
        public static void main(String[] args){
                Bob bob1 = new Bob(64);
                System.out.println(“Age of bob1=“+bob1.getAge());
        }
}

class Bob {
        private int age;  // access limited to Bob methods
        public Bob(int age){
                this.age = age;
        }

        public int getAge(){
                return this.age;
        }

}
```

This type of *Accessor* is called a **getter.** A **private** member variable and only a **getter** gives you *read-only* data

```
> javac Constructor.java
> java Constructor

Age of bob1=64
```

There are also **setter** *Accessors*.

But then what’s the point of *private*?

## A *setter* for Bob’s age (including validation)

```
public void setAge(int a){
    if ((a < 0) || (a > 115)){
        System.out.println("Bad Bob age:"+a);
    } else {
        this.age = a;
    }
}
```

If you try:

```
bob1.setAge(400)
```

You’ll get and error and `bob1`’s age won’t be corrupted