OOP TA Session II

- Arithmetic and logical operators
- Control flow
- Constructors
- Method overloading
- Unit testing: JUnit
- Exercise I - Introduction

Arithmetic Binary Operators

- **+** Addition
- **-** Substraction
- ***** Multiplication
- **/** Division
- **%** Modulo

Remember :
result type = more “complicated“ operand type

Assignment Binary Operators

- **Assignment:**
- **n = n * 3;**
- **Shorthands:**
  - **n += x (n = n+x)**
  - **n *= x (n = n*x)**
  - **n %= x (n = n%x)**
  - **n /= x (n = n/x)**

(What‘s **n>>=2** ?)

Assignment Unary Operators

- `n++`
- `++n`

`(n += 1)`

- `n--`
- `--n`

`(n -= 1)`

Comparison Operators

- ==
- >, >=
- <, <=
- !=

Logical Operators

- **&& (and)**
- **|| (or)**
- **! (not)**

What does this do?

if ((a || b) && !(a && b)) ...

Control Flow - if

```
if (a == b) {
   // ..
} else if {
   // ..
} else {
 // ..
}
```

Control Flow – Switch!

```
switch (someChar) {
case 'a':
    // ..
    break;
case 'b':
    // ..
    break;
//more cases...
default:
    // ..
    break;
}
```

Control Flow – Switch!

- What will this do ?

```
switch(buttonPressed) {
case  1:
    makeCoffee();
case 2:
    makeHotChocolate();
case 3:
    makeFrenchVanilla();
}
```

Control Flow - Conditions

```
switch(buttonPressed) {
  case  1:
      makeCoffee();
      break;
  case 2:
      makeHotChocolate();
      break;
  case 3:
      makeFrenchVanilla();
      break;
}
```

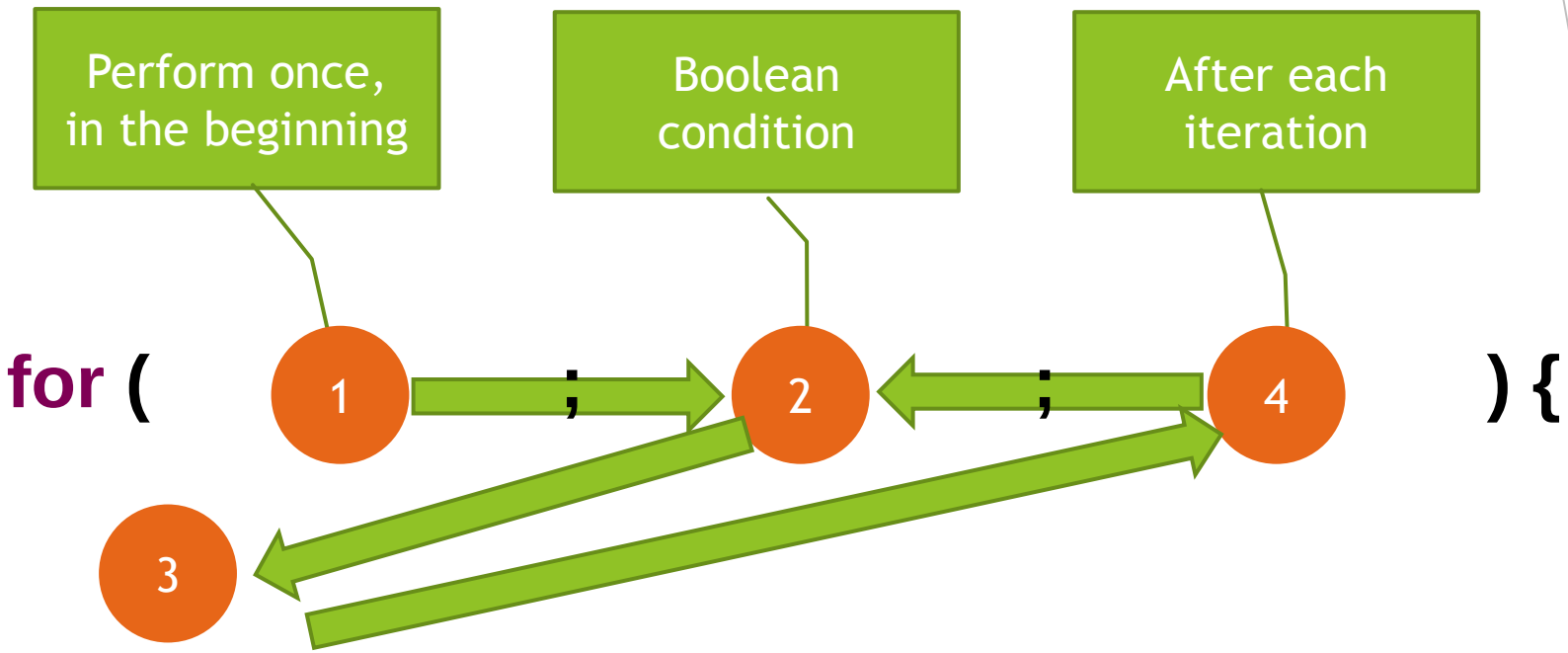
Switch works with:

- **int**
- **short**
- **byte**
- **char**
- **String**

Control Flow – While

```
while (num > 0) {
    //.. Do something
}
```

Control Flow - For



Control Flow - For

```
for (    int i = 0    ;    i < 10    ;    i++        ) {

}
```

Control Flow - For

```
String str = “_______3____“;
int i;
for (i = 0 ;
     i < str.length && str.charAt(i) != ‘3‘ ;
     i++) {}
```

Can be replaced by “;”

Control Flow – Break, Continue

```
String str = "_______3____";
int i;
for (i = 0 ; i < str.length ; i++) {
   if (str.charAt(i) == '3')
      break;
}
```

One-line blocks will compile without brackets

Continue: also like in Python

Control Flow – For: 2nd syntax

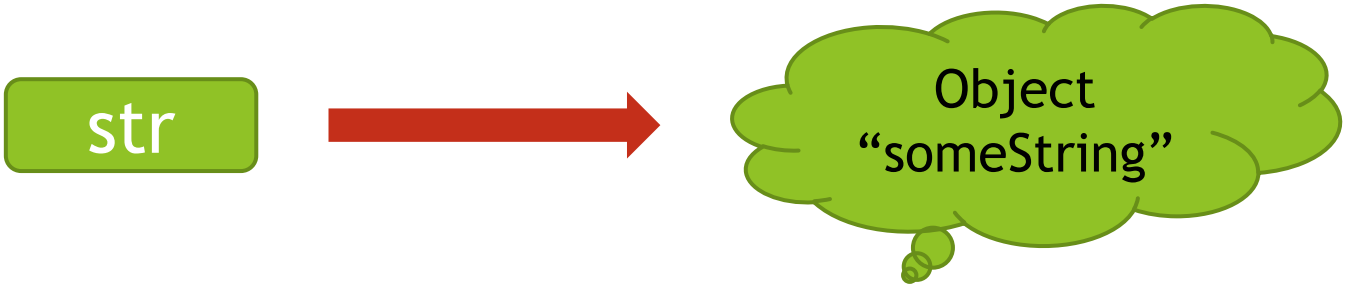
```
int[] array = new int[]{0,1,2,3};
for (int i : array) {
    //.. Do something
}
```

Why is this bad?

```
int[] array = new int[]{5,3,10,4};
for (int i : array) {
    //.. Do something
}
```

null (“None“ equivalent)

String str = “someString“;



str = null;

Constructors

```
public class ComplexNumber {
    private double _real;
    private double _img;

    public void setReal(final double realPart) {
        _real = realPart;
    }
    public void setImg(final double imgPart) {
        _img = imgPart;
    }
    //..
}
```

What's the value before using the setters?

Constructors – default constructor

```
ComplexNumber  myComplexNumber = new ComplexNumber();
```

- The **default constructor** was called.
- The default constructor:
  - Is generated when no other constructor was defined.
  - Initializes fields to default values.

Constructors

```
public class ComplexNumber {
    private double _real;
    private double _img;

    public ComplexNumber (final double realPart, final double imgPart) {
        _real = realPart;
        _img = imgPart;
    }
//..
}
```

Constructor name is the name of the class

No return type or return value

Constructors

- The **default constructor** is no longer generated.
- there can never be an uninitialized complex number ever again. Ever.

```
ComplexNumber  myComplexNumber = new ComplexNumber();
```

✗

```
ComplexNumber  myComplexNumber = new ComplexNumber(3,2);
```

✓

Constructor overloading

- If we want to enable default complex numbers:

```
public class ComplexNumber {
    //...
    public ComplexNumber () {
        _real = 0;
        _img = 0;
    }
    public ComplexNumber (final double realPart, final double imgPart) {
        _real = realPart;
        _img = imgPart;
    }
}
```

Constructor overloading

- Now there are 2 “versions“ of constructors:

```
ComplexNumber myComplexNumber = new ComplexNumber(5,2);
ComplexNumber firstNumber = new ComplexNumber();
```

- We **overloaded** the constructor.
- Every method can be **overloaded.**

Method Overloading

```
public class ComplexNumber {
    private double _real;
    private double _img;
    public ComplexNumber() { /* .. */}
    public ComplexNumber(final double real, final double img) { /* .. */ }

    public void add(final ComplexNumber other) {
        // .. Add other to the current ComplexNumber.
    }

    public void add(final double real, final double img) {
        // .. Add values to the real and img part.
    }
}
```

Method Overloading

- Overloading is allowed if there‘s **never** any ambiguity:

```
public class ComplexNumber {
    //...
    public double getRealPart() {
        return _real;
    }
    public int getRealPart() {
        return (int)_real;
    }
    //...
}
```

What’s the type of “myComplexNumber.getRealPart()” ?

Method Overloading

- Overloading is allowed if there‘s **never** any ambiguity:

```
public class ComplexNumber {
    //...
    public void add(final double real, final double img) {
        System.out.println(1);
    }
    public void add(final int real, final int img) {
        System.out.println(2);
    }
}
```

myComplexNumber.add(1,2);
What will be printed ?

toString()

- Every object in Java has a string representation.
- This will compile!

```
System.out.println(myComplexNumber);
```

- But will print class name + some hash...
- The string representation can be defined by implementing a special method:

toString()

toString()

```
public class ComplexNumber {
    private double _real;
    private double _img;
    // ..
    public String toString() {
        return _real + "+" + _img + "i";
    }
    //..
}
```

toString()

```
public String toString() {
        return _real + "+" + _img + "i";
}
```

```
ComplexNumber myComplexNumber =
        new ComplexNumber(5,2);

System.out.println(myComplexNumber.toString());  // 5+2i
System.out.println(myComplexNumber);             // 5+2i
```

Unit Testing - JUnit

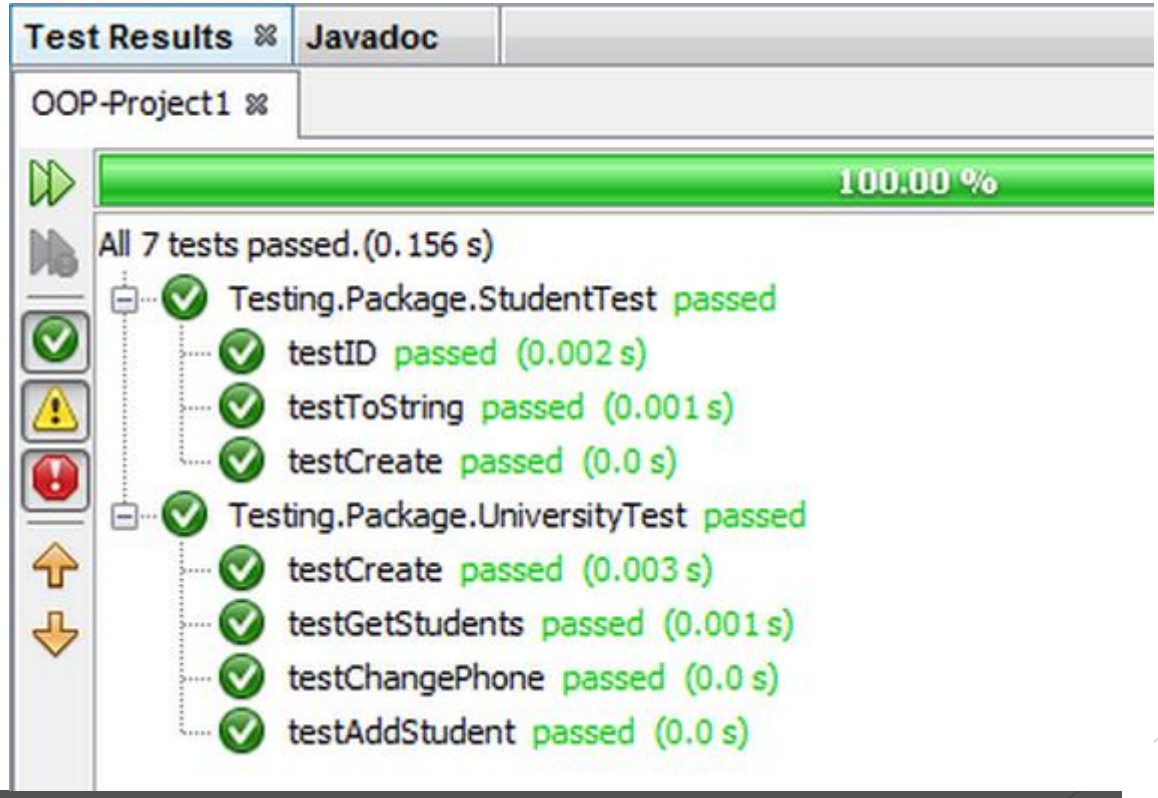
- Test levels:
  - Unit level
  - Integration level
  - System level
- All levels are important.
- “Black-box“ unit testing is partially automated in Java!

Unit Testing - JUnit

```
import static org.junit.Assert.*;
import org.junit.Test;
public class ComplexNumberTest {
        private ComplexNumber cn;
        @Before
        public void setUp() {
                cn = new ComplexNumber(1,1);
        }
        @After
        public void tearDown() {
        // ..
        }
        @Test
        public void testAdd() {
                cn.add(4,3);
                assertEquals(cn.getReal(), 5);
                assertEquals(cn.getImg(), 4);
        }
}
```

Unit Testing - JUnit

- Eclipse supports **JUnit:**
  - Automatically create test classes and methods
  - Run the tests without an explicit main
  - Get tests statistics:



- Tutorial: http://www.vogella.com/tutorials/JUnit/article.html

First Exercise - Introduction

- Mastermind, Bulls and Cows, “Bul Pgiaa“.
- Read the rules: http://en.wikipedia.org/wiki/Mastermind_(board_game)
- Read class definitions (MastermindUI, CodeGenerator, Code)
- Implement Mastermind class (Only main function).