<< Lecture 5 >>

Today's Plan

- The do-while loop!
  - *Didn't cover it last class*
- Modules:
  - Static classes
  - Accessing members in classes
  - Classes as types, values as objects
  - The constructor
  - The null value

Do-while loop

- Last lecture, I forgot to mention the **do-while loop**
- Syntax:

```
do {
  block of instructions to repeat
} while ( guard )
```

- The **do-while** loop behaves like the while loop, except that the guard is evaluated at the end of each block
- In practice, it behaves just like a normal **while** loop, except that the block is executed **at least once**

(check https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html for a review of the basics of Java)

Accessing modules

- In Java modules are either:
  - Classes
  - Packages, which contain classes or other packages
- We will now study **classes**
- Classes represents portions of code that are reusable and that define **types**
- With classes, we can create **custom types** or, if you like, intelligent variables
- We will start with a class as a group of methods, though

EXAMPLE STATIC CLASS

- As mentioned earlier, **static methods** can be called at any time
- The **public static void main** method is one such example
- Let us study the example below:

```
class Demo {

  public static void main(String[] args) {
    IO.print("Hello World with IO!");
  }

}


class IO {

  static void print(String x) {
    System.out.println( x );
  }

}
```

- To access method **println** in **IO**, we just write
  - **IO.print(** some String **)**
- **If we want to access a method by pointing out the class name (eg., IO), then the method must be static**
- We can also have static variables; naturally, static variables can be accessed just like methods

```
class Demo {

  public static void main(String[] args) {
    IO.print("Hello World with IOX!");
    IO.print("Value IO.line = " + IO.line);
    IO.print("Value IO.line = " + IO.line);
  }

}


class IO {
  static int line = 1;

  static void print(String x) {
    System.out.println( line + "..." + x ) ;
    line++ ;
  }

}
```

Example Non Static Class

- **Most important use case!**
- We will now use a class to define a little *data structure*
- Let us define a class that describes a person, without methods

```
class Person {

  String name = "";
  int age  = 0;

  void info() {
      System.out.println("Person "+name+" is "+age+" yo");
  }

}
```

- If we tried to access method **info** in the **main** of **Demo**, we would get an error; this time, method **info** is meant to be used in an **instance** of **Person**, i.e., in an **object**

```
class Demo {

  public static void main(String[] args) {
      Person p1 = new Person();
      p1.name = "Rene";
      p1.age  = 21;
      Person p2 = new Person();
      p2.name = "Renee";
      p2.age  = 12;
      p1.info();
      p2.info();
  }

}


class Person {

  String name = "";
  int age  = 0;

  void info() {
      System.out.println("Person "+name+" is "+age+" yo");
  }

}
```

- Why can't we just do **Person.info()**? Because **Person** can assume many different instances (or *incarnations* if you wish)
- Note the instruction: **Person p1 = new Person();**
  - **Person** is now a **type**
  - Variable **p1** is of type **Person**, so the fields of Person are accessible from p1!
  - **new Person()** creates a **new** instance of Person
- Note the instruction: **p1.name = "Renee"**

- **p1.name** access field name in object **p1**
- Field **name** was defined in **Person**, but can't do **Person.name**
  - Why? Because the field **is not static**
  - But if it was static, it could not have several instances

- Now, let us alter the **main** method of Demo, to check that the equality operation does not compare fields

```
class Demo {

  public static void main(String[] args) {
      Person p1 = new Person();
      p1.name = "Rene";
      p1.age  = 21;
      Person p2 = new Person();
      p2.name = "Rene";
      p2.age  = 21;
      p1.info();
      p2.info();
      System.out.println("p1==p2? " + (p1==p2) );
      System.out.println("p1==p1? " + (p1==p1) );
      System.out.println("p2==p2? " + (p2==p2) );
  }

}
```

- To solve this problem, we implement and use an **equals** method that returns a boolean:

```
class Person {

  String name = "";
  int age  = 0;

  void info() {
      System.out.println("Person "+name+" is "+age+" yo");
  }

  boolean equals(Person x) {
      return (age == x.age) && (name == x.name);
  }

}


class Demo {

  public static void main(String[] args) {
      Person p1 = new Person();
      p1.name = "Rene";
      p1.age  = 21;
      Person p2 = new Person();
      p2.name = "Rene";
      p2.age  = 21;
      p1.info();
      p2.info();
      System.out.println("p1==p2? " + (p1==p2) );
      System.out.println("p1.equals(p2)? " + p1.equals(p2) );
  }

}
```

- Earlier, I had mentioned that the **String** values were just objects of type String, and it happens occurs that there is String.equals!

```
System.out.println( "test".equals("test") );
```

Accessing members... We have seen this!

- Example:
  - **System.out.println("Hello World!");**
  - We are accessing:
    - method **println(String)** in module **out**
    - and module **out** in module **System**
  - **System** is an object (*instance* of a class) representing some properties of the system
  - **out** is an object representing the *standard out stream* (textual output to console)

The constructor

- We know that class Person has only two fields, name and age, so why don't we initialize objects of type Person with these data immediately?
- We can do so using the constructor
- A constructor is a special method that does not return anything explicitly in the code, and it is called upon the use of **new**
- Of course, doing **new Person(...)** does return (create) a new object, but that is a given (**new** allocates memory, etc., for the new instance)

```
class Person {
  String name = "";
  int age  = 0;

  Person(String n, int a) {
     name = n;
     age  = a;
  }

  void info() {
      System.out.println("Person "+name+" is "+age+" yo");
  }

  boolean equals(Person x) {
      return (age == x.age) && (name == x.name);
  }
}

class Demo {
  public static void main(String[] args) {
      Person p1 = new Person("Rene", 21);
      Person p2 = new Person("Rene", 21);
      p1.info();
```

```
        p2.info();
        System.out.println("p1==p2? " + (p1==p2));
        System.out.println("p1.equals(p2)? "+(p1.equals(p2)));
    }
}
```

- Now, we cannot use **new Person()** anymore, because the constructor now has two parameters (arguments)
- If we don't explicitly define a constructor, Java still supplies an implicit constructor!

STATIC AND NON STATIC MEMBERS TOGETHER?

- **You better know what you are doing**
- It is possible to combine static and non static variables (fields) and methods in classes
- Having static methods is commonplace
  - Static methods can't access non static fields or methods directly
  - Non static methods can access everything
- Static variables are some sort of **global variable for all instances of the class**
  - They are weird! **You better know what you are doing**
  - The code can be complicated to read if you mix stuff

```
class Person {
    String name = "";
    int age  = 0;
    static int people = 0;

    Person(String n, int a) {
        name = n;
        age  = a;
        people++;
    }

    void info() {
        System.out.println("Person "+name+" is "+age+" yo");
    }

    boolean equals(Person x) {
        return (age == x.age) && (name == x.name);
    }
}


class Demo {
    public static void main(String[] args) {
        Person p1 = new Person("Rene", 21);
        Person p2 = new Person("Rene", 21);
        p1.info();
        p2.info();
        System.out.println("p1.people = "+ p1.people);
        System.out.println("p2.people = "+ p2.people);
    }
}
```

- Now, we cannot use **new Person()** anymore, because the constructor now has two parameters (arguments)
- If we don't explicitly define a constructor, Java still supplies an implicit constructor!

THE THIS KEYWORD

- We defined the constructor as **Person(String n, int a)**, but what if we called the arguments **name** and **age**? (We could do so for the desirable goal of readability.)

```
Person(String name, int age) {
      name = name;
      age  = age;
  }
```

- This is clearly faulty—by **scope** (i.e., what is available in the block) variables name and age correspond to the arguments, not the fields
- The solution: to use **this.name** and **this.age** to refer to the fields

```
Person(String name, int age) {
      this.name = name;
      this.age  = age;
  }
```

- So, what is **this**? **this** stands for the **instance of the object itself**

```
class Person {
  String name = "";
  int age  = 0;

  Person(String name, int age) {
      this.name = name;
      this.age  = age;
  }

  Person me() {
      return this;
  }

  void info() {
      System.out.println("Person "+name+" is "+age+" yo");
  }
  boolean equals(Person x) {
      return (age == x.age) && (name == x.name);
  }
}

class Demo {
  public static void main(String[] args) {
      Person p1 = new Person("Rene", 21);
      Person p2 = p1.me();
      p1.info();
      p2.info();
      System.out.println("p1==p2? " + (p1==p2));
  }
}
```

<< Lecture 6 will resume from here >>