

Introduction to Systems Programming

Code Review

Introduction to Systems Programming

Reviewing the structure of a class

Reviewing the structure of a class

```
package laboratory;

import java.util.ArrayList;
import laboratory.data.Student;
import laboratory.data.Teacher;
//could use laboratory.data.* to import all classes

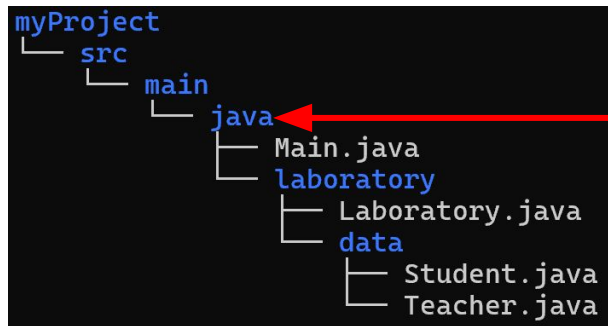
/**
 * <Class documentation>
 */
public class Laboratory {

    private ArrayList<Student> students;
    private Teacher teacher;
    private String name;

    /**
     * <Constructor documentation>
     */
    public Laboratory(String name, Teacher teacher) {
        //<implementation>
    }

    //More methods

    /**
     * <Method documentation>
     */
    public int numberOfStudents() {
        return students.size();
    }
}
```



We compile/run
while being in this
folder

*The src/main/java folder is the usual convention for
a Java project, it's one of many standard folders.*

Reviewing the structure of a class

```
package laboratory;

import java.util.ArrayList;
import laboratory.data.Student;
import laboratory.data.Teacher;
//could use laboratory.data.* to import all classes
```

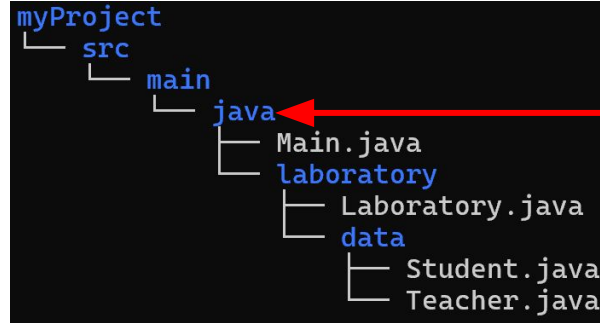
```
/**
 * <Class documentation>
 */
public class Laboratory {

    private ArrayList<Student> students;
    private Teacher teacher;
    private String name;

    /**
     * <Constructor documentation>
     */
    public Laboratory(String name, Teacher teacher) {
        //<implementation>
    }

    //More methods

    /**
     * <Method documentation>
     */
    public int numberOfStudents() {
        return students.size();
    }
}
```



We compile/run while being in this folder

The src/main/java folder is the usual convention for a Java project, it's one of many standard folders.

import in Java does not perform code insertion as in, for example, C.

Reviewing the structure of a class

```
package laboratory;

import java.util.ArrayList;
import laboratory.data.Student;
import laboratory.data.Teacher;
//could use laboratory.data.* to import all classes

/**
 * <Class documentation>
 */
public class Laboratory {

    private ArrayList<Student> students;
    private Teacher teacher;
    private String name;

    /**
     * <Constructor documentation>
     */
    public Laboratory(String name, Teacher teacher) {
        //<implementation>
    }

    //More methods

    /**
     * <Method documentation>
     */
    public int numberOfStudents() {
        return students.size();
    }
}
```

The interface of the class



```
package laboratory;

import java.util.ArrayList;
import laboratory.data.Student;
import laboratory.data.Teacher;

/**
 * <Class documentation>
 */
public class Laboratory {

    /**
     * <Constructor documentation>
     */
    public Laboratory(String name, Teacher teacher)

    //More methods (documentation and signatures)

    /**
     * <Method documentation>
     */
    public int numberOfStudents()
}
```

Visibility, class fields/methods, constants.

Accessibility of members declared by class A from different classes.

Visibility/Class	From class A (package pA)	From class A2 (package pA)	From class B2 (package pB, subclass of A)	From class B1 (package pB)
private	ACCESSIBLE	INACCESSIBLE	INACCESSIBLE	INACCESSIBLE
package	ACCESSIBLE	ACCESSIBLE	INACCESSIBLE	INACCESSIBLE
protected	ACCESSIBLE	ACCESSIBLE	ACCESSIBLE	INACCESSIBLE
public	ACCESSIBLE	ACCESSIBLE	ACCESSIBLE	ACCESSIBLE

Visibility, class fields/methods, constants.

Sometimes, we need members (fields or methods) that should not require an object to be used. For example, if we want to count how many objects of a class have been created; a math function that doesn't require a state, e.g.: fibonacci; sorting functions. In some cases we want a class to have at most one object (see Singleton pattern), constructors must be private and we must use a class method to the the one instance allowed.

To declare a class member we use the keyword **static**

Class members

Let's see a Singleton
class and an object
counter.

Visibility, class fields/methods, constants.

Sometimes, we need fields, and/or variables, to behave as constants. A constant MUST be initialized and one cannot change its value. We use the **final** keyword for this.

Introduction to Systems Programming

Parameterized classes (Generic classes)

Generic/Parameterized Classes

Sometimes, a class may have fields that may not have a specific type. We have seen this with `ArrayList`, where we have “ArrayList of T”, e.g.:
`ArrayList<String>` as an “ArrayList of String”; `ArrayList<Student>` as an “ArrayList of Student”.

So far we have used parameterized classes, let's see how to make them.

Generic/Parameterized Classes - Motivational example

We want to have a Book Library where we can search for a Book by a title, but that book may not exist.

```
/**
 * This class represents a personal library.
 */
public class Library {

    private ArrayList<Book> books;

    //Constructors and Methods

    public ? searchBook(String expression);

}
```

Generic/Parameterized Classes - Motivational example

We want to have a Book Library where we can search for a Book by a title, but that book may not exist.

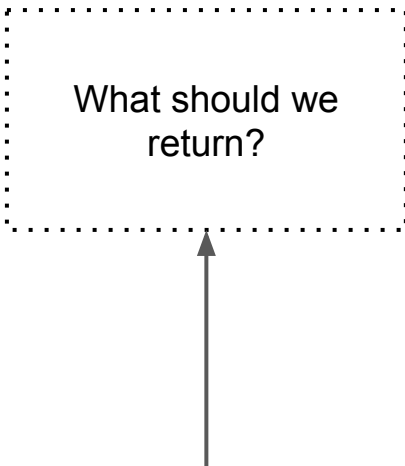
```
/**
 * This class represents a personal library.
 */
public class Library {

    private ArrayList<Book> books;

    //Constructors and Methods

    public ?searchBook(String expression);
}
```

What should we return?

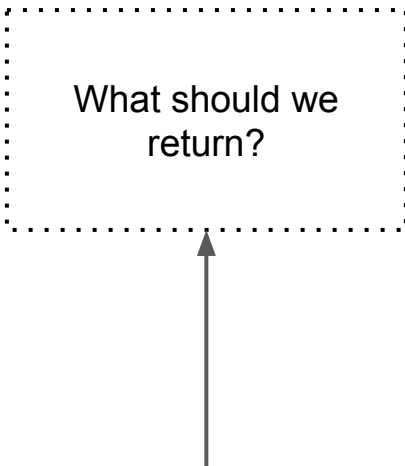


Generic/Parameterized Classes - Motivational example

We could make a class that represents an optional value, i.e.: “Maybe a value”. But making a class like this only for Book would not be reusable.

```
/**  
 * This class represents a personal library.  
 */  
public class Library {  
  
    private ArrayList<Book> books;  
  
    //Constructors and Methods  
  
    public ?searchBook(String expression);  
  
}
```

What should we
return?



Generic/Parameterized Classes - a Maybe class

Let's make a "Maybe a value" class.

Introduction to Systems Programming

Designing classes

Software need to change

Software is not a constant product, it evolves, it is modified. The phrase “change or die” is a very pertinent invariant in software development.

- A software that is continuously maintained will prevail in time.
- A software that is static will become obsolete and die.

From this we can infer that a software that is hard to maintain will be thrown away.

There are ways to increase the maintainability of our software.

Code quality

- Self documented code, i.e.: the code is easy to understand and follow.
- Proper documentation.
- Testability, i.e.: is easy to test the code.
- Low coupling.
- High cohesion.
- Others ...

Code quality

- Self documented code, i.e.: the code is easy to understand and follow.
- Proper documentation.
- Testability, i.e.: is easy to test the code.
- Low coupling.
- High cohesion.
- Others ...

Coupling and Cohesion

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are tightly coupled.
- We aim for loose coupling.

Coupling and Cohesion

Loose coupling makes it possible to:

- Understand one class without reading others.
- Change one class without affecting others.
- Thus: improves maintainability.

Coupling and Cohesion

- Cohesion refers to the the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has high cohesion.
- Cohesion applies to classes and methods.
- We aim for high cohesion.

Coupling and Cohesion

High cohesion makes it easier to:

- Understand what a class or method does.
- Use descriptive names
- Reuse classes or methods.

Coupling and Cohesion

- A method should be responsible for one and only one well defined task.
- Classes should represent one single, well defined entity.

Code duplication

Code duplication:

- Is an indicator of bad design.
- Makes maintenance harder.
- Can lead to introduction of errors during maintenance.

Responsibility-driven design (RDD)

- It focuses on the contracts between client (*) and provider (**).
- Each class should be responsible for manipulating its own data.
- RDD leads to low coupling.

() The one using the features (provided by a class).*

*(**) The one offering the features (the class).*

Localizing change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected *.

() This is a general rule, not only applied to RDD.*

Thinking ahead

- When designing a class, we try to think what changes are likely to be made in the future.
- We aim to make those changes easy.

But remember: do not overengineer your solutions.

Refactoring

- When classes are maintained, often code is added.
- Classes and methods tend to become longer.
- Every now and then, classes and methods should be refactored to maintain cohesion and low coupling.

Refactoring and testing

- When refactoring code, separate the refactoring from making other changes.
- First do the refactoring only, without changing the functionality.
- Test before and after refactoring to ensure that nothing was broken (*Regression testing is the usual way to validate this*).

Design guidelines

- A method is too long if it does more than one logical task.
- A class is too complex if it represents more than one logical entity.

Note: these are guidelines - they still leave much open to the designer.