1

Building strong objects

Noël PLOUZEAU
IRISA/ISTIC

1

2

In this part

- We will investigate good techniques to
  - ensure encapsulation
  - minimize dependencies
  - manage the lifecycle of objects

2

3

Encapsulation

- An object must master what happens to its own data (attributes)
- Every action on attributes (read, update) must go through the methods of the object
- In this part we will show how to ensure this

3

Rule #EJ13
Minimize accessibility

- Rule: make each type or attribute as unaccessible as possible
  - if using private does not break your code, use it!

4

Access levels in Java

- private
  - no access for subclasses, good idea for attributes
- (nothing: defaults to package private)
  - code in the same package has access
  - do not use: trust no one

5

Access levels in Java

- protected
  - subclasses and package have access
  - do not use for attributes
- public
  - **NEVER EVER** use it for attributes

6

API and accessibility

- Protected and public items (interfaces, classes, attributes, operations) are part of your API (application programming interface)
  - So be very careful
  - protected items are also part of implementation: **dangerous**
- private and package private are implementation details
  - They are not published in the API

7

Rule #EJ14

- attributes must be private
- therefore one needs accessors
  - getSomething(), setSomething(...)
- this way you keep control of your data

8

Accessors pitfalls

- What is wrong with this code?
  - class A {
    - private B b;
    - public B getB() { return this.b }

9

Careful with references

- if your return a reference of an attribute make sure the type is immutable (no setSomething operations)
- otherwise third party code can change the attribute state
- sometimes this is ok because it is planned carefully (eg using an Observer design pattern)

10

Examples

- int getSize() { return this.size; }
  - // **OK**, it's a copy (int values are not objects)
- String getName() { return this.name; }
  - // **OK**, it's a reference to an immutable class
- B getB() { return this.b; }
  - // **WARNING:** b may be modified surreptitiously elsewhere

11

The Collection problem

- Collections are a well-known problem for encapsulation
- It is easy to break encapsulation
  - This can lead to hair pulling bugs
- We will see several solutions for this

12

The bad

- What's wrong with the following code?
- class Broken {
  - private Collection myCol;
  - Collection getMyCol() {
    return this.mycol;}

13

Encapsulation is broken

- Client code example
  - Broken o = ...;
  - Collection c = o.getMyCol();
  - c.add(...); c.remove(...);
  - // o.myCol has changed without control from o

14

Ways to protect encapsulation

- Return a copy of the internal object
- Return an immutable object
- Add your own control operations to the returned object

15

Example for collections

```
class ColExample {
  private Collection<String> myCol;
  public Iterator<String> getIterator() {
    return this.myCol.iterator();
  }
  // No getMyCol operation => read only
```


16

Rule #EJ52
Use interfaces for object reference

- To use an object you don't need to know its concrete class

```
Collection<String> col = .....;
// col initialised from somewhere
col.add("Hello"); // We don't know the class and we are happy with that!
```


17

Use interfaces everywhere

- Which one is the best one?
  1. ArrayList l = new ArrayList();
  2. List<String> l = new ArrayList<String>();
  3. ArrayList<String> = new ArrayList<String>();

18

Use interfaces everywhere

- Which one is the best one?
  1. public doThis(ArrayList<String> l);
  2. public doThis(List<String> l);
  3. public doThis(Collection<String> l);


19

Rule #EJ18
Interfaces are better than abstract classes

- In Java you can do multiple inheritance with interfaces (operation inheritance)
- Single inheritance for methods
- If you provide an abstract class you force an implementation decision on your users


20

When to use abstract classes?

- To provide a partial implementation that users must finish
- This requires a very good documentation of how the subclasses must cooperate
  - or else everything can break easily (see rule #EJ16 later)


21

Rule #EJ16
avoid method inheritance

- We like method inheritance because it should provide powerful reuse means
  - yes but method inheritance breaks encapsulation

22

Example

```
class A {
  protected void doIt() { otherStuff(); }
  protected void otherStuff (){
    System.err.println("A::otherStuff() called");
    importantThing();
  }
```


23

Example (cont'd)

```
class B extends A {
  @Override
  protected void doIt() { super.doIt(); newThings()}
  @Override
  protected void otherStuff (){
    System.err.println("B::otherStuff() called");
  }
```


24

What will be printed?

- B b = new B(); b.faire();
- "B::otherStuff() called" will be printed
- Implementation of A::doIt() will break (no call to super.otherStuff in B)
- Now a perfectly alright A class can be broken by a simple extension

25

More problems

- Suppose that B is carefully written, A is carefully documented, both well tested
- Now A is modified (eg new operations, slight change in algorithm, etc)
  - B must be modified (*fragile base class* problem)
  - in the mean time B is silently broken!

26

Consequence of this

- Think twice before choosing method inheritance over composition (see later)
- If you are maintaining class A then you can extend A with B (carefully)
- Do not extend a library class (unless it has been designed for that)
- Document inheritance possibilities of a class (ie internal details!)
  - or else forbid inheritance completely

27

Use @Override

- Use @Override annotation for an operation
  - This helps the compiler to spot incorrect redefinition of operations
- Concretely, an operation definition that states @Override will not be mistaken for an overloading by the compiler.

28

Bad case

- We define a new type, we must define equals
- public boolean equals(Length l) {
- return (this.valueInMeters == l.getNumericalValue(LengthUnit.METER));
- }
- // The Object::equals method will be called for tests

29

Bad case (cont'd)

- This triggers a compile time error
  - equals not defined
  - adding an @Override in the front of define does not fix that
  - we need to redefine equals(Object)

30

Correct version

- @Override public boolean equals(Object o) {
  - if(! (o instanceof LengthImpl)) return false;
  - Length l = (Length) o;
  - return (this.valueInMeters == l.getNumericalValue(LengthUnit.METER));
- }

31

Using composition

- The idea is that you reuse code not by extending it but by calling it (delegation)
- therefore you only need to know an interface

32

Example

- interface List {
  - public void add(Value v);
- }
- // We need to declare an interface, see rule #EJ52
- class ListImpl implements List {...}

33

Method inheritance

- class MyClumsyList extends ListImpl {
  - public void add(Value v) {
    - checkValue(v);
    - super.add(v);
  - }

34

Composition

- class MyBetterList implements List {
  - private List internalList = new ListImpl();
  - public void add(Value v) {
    - checkValue(v);
    - internalList.add(v);
  - }

35

Comparison

- MyClumsyList depends on a library class (ListImpl)
  - this information is visible
  - if ListImpl is modified in the future MyClumsyList can break
  - choice of ListImpl versus other classes is done at compile time (not flexible!)

36

And the winner is...

- MyBetterList depends on
  - an interface List
  - one implementation of List
  - but the connection between implementations is through operations only (flexible)

37

Rule #EJ44
Document your API

- Your design has two parts
  - a public one (used by other designers and coders)
  - a private one (used by you only)
- Maintaining a well designed separation is critical for reliability and extensibility

38

What is the API

- This is the public part
  - public types (interfaces, some implementation classes)
  - public operations (signatures, constraints, exceptions)
- Writing the interfaces, operations, constraints, etc is the first step of design

39

Use Javadoc

```
/**
* Manages a phone book, as a set of PersonData items.
* Name duplicates are not allowed.
*
* @author nplouzeau
* @version 1.0
*/
public interface PhoneBook {
```


40


```
/**
        * Looks for a person by her name
        * @param name  name to look for
        * @return  a copy of the person data that matches the name,
        *     or null if no name matches
        *
        */
        public PersonData findPerson(String name);
```


41


```
/**
        * Adds a new person data object into the phone book.
        * @param name    name to add
        * @param phoneNumber  phone number to add for the name
        * @throws DuplicatePhoneDataException    if name already
 used
        */
        public void addPerson(String name, String phoneNumber);
```


42

Rule #EJ38
Check parameter validity

- This is in line with the contract-based design (CBD)
- For each operation provide
  - the preconditions
  - the postconditions

43

Preconditions

- An operation call is allowed only if all preconditions evaluate to true
- If at least one precondition is false than the operation can do anything
- This is a fault by the caller, not the operation implementer

44

Postconditions

- If all preconditions are true, then execution of the operation's method must bring the object to a state where all postconditions are true
- else this is a bug and it the implementer's fault

45

Design by contract

- A contract is the set of preconditions and postconditions for an operation
- Contract stake holders: caller and callee
- The caller is responsible for ensuring that the preconditions are satisfied
- The callee is responsible for ensuring that the postconditions are satisfied **if** the preconditions are satisfied


In Java

- Document the preconditions of each operation together with what happens when a precondition is not satisfied, for each precondition
- Document the postconditions & effects of each operation
- **Write code that checks for the preconditions** and signals problems to the caller



```
/**
        * Add o to the set of observers that will be notified when
        * the value of this changes.
        * pre : o is not already attached
        * @param o   the observer to attach
        * @throws IllegalArgumentException if o is already attached
        */
        @Override
        public void attach(Observer<T> o) {
                if (registeredObservers.contains(o))
                throw new IllegalArgumentException("o already attached");
                registeredObservers.add(o);
        }
```


Remark on checks

- If a precondition involves a parameter, make a copy of the parameter before checking
- The semantics of the precondition is that it must be evaluated instantaneously at the beginning of the method


49

Rule #EJ57
Exceptions must be exceptional

- An exception signals a problem, not a common possibility of execution
- For instance, reaching an end of file is **not** an exceptional situation


50

Exceptions for preconditions

- Exceptions are a good way to signal invalid preconditions
  - but you must document this: say in the Javadoc which exception is raised for each precondition


51

Example

```
/**
        * Removes o from the set of observers that will be notified when
        * the value of this changes.
        * pre : o must be in the attached observers set
        * otherwise an IllegalArgument  exception is thrown
        *
        * @param o the object to be removed
        * @throws IllegalArgumentException when o is not attached
        */
        @Override
        public void detach(Observer<T> o) {
                if (!registeredObservers.contains(o)) throw new IllegalArgumentException("o not attached");
                registeredObservers.remove(o);
        }
```


52

Checking for preconditions

- As the caller has to check for precondition before calling
  - you must provide means for checking them

53

Example

- interface Iterator<T> {
  - /**
  - * .....
  - @throws InvalidPosition if all items have be returned
  - public T next() throws InvalidPosition;

54

Example (cont'd)

- Without any means to check if there something left:
- Iterator<String> itString = ...;
- try {
  - s = itString.next();
- }
- catch (InvalidPosition e) {
  - // Reached end of iteration

DO NOT DO


55

The proper way

- // Add a retrieve operation
- /**
- * @returns true iff there are more items to read
- */
- public boolean hasNext();


56

Rule #EJ61
Manage exception abstraction

- Software is often organized in layers (low level services to high level services)
- If a low level service throws an exception, it must be caught in the middle level
  - either the problem is dealt with there
  - or the problem cannot be fixed at middle level
  - in this case catch and throw a middle level exception


57

Example

- Three layers L1 to L3
- L1 catches a file read error exception
- L1 throws an exception that describes the context of the read error using concepts at the L1 level (eg database index read error)
- L2 catches this and translates it into name search error, and so on...

58

Rule #EJ25
Prefer list to arrays

- Arrays have several problems
  - not dynamic in size (difficult to choose a size, overflows, unused memory space)
  - no proper iteration mechanism (back to integer indexes!)
  - poor compile time type checks

59

Concretely

- Which code is the most flexible?

1. String[] names = String[10];
2. Collection<String> names = new ArrayList<String>();

60