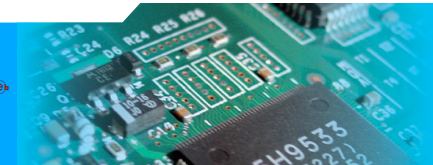


Best practices in Java

Noël PLOUZEAU

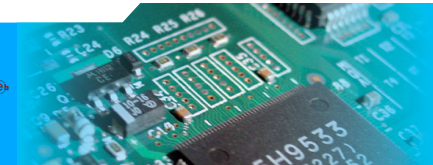
IRISA/ISTIC



References

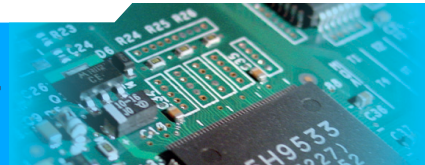
© Bloch

- © BLOCH, Joshua. Effective Java v3. Addison-Wesley Professional, 2018.
- © Note: The v2 edition is also quite useful (and slightly easier to understand compared with v3)



Use interfaces

- In Java we have two ways of defining operations without code
 - Interfaces
 - Abstract classes
- Java supports multiple inheritance for interfaces
 - but only single inheritance for classes
- Very different from C++



Advantages of interfaces

- Open for extensibility (**S****O**LID)
 - If a class implements a new interface it is easy to adapt it or build a subclass
 - A class can implements a set of different interfaces (mixin)
- Example
 - See [bestpractices.interfaces](https://bestpractices.interfaces.on.gitlab) on gitlab



Example

© See ACO2018/BestPracticesExamples interfaces



Key points of the example

- BadClass2 inherits from BadClass1 just to get operations, and no existing code...
- GoodClass1 can inherit from an implementation class if needed
- And this is what GoodClass2 is doing, plus implementing an other interface (this is a *mixin*)
- GoodClass3 is even better than GoodClass2 (see later)



Other key points

- Use of a pseudo PrintStream object
 - Either by creating one that one can read after and check its contents
 - Or by using Mockito to mock a PrintStream and check that the print operation was called with the proper argument



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)



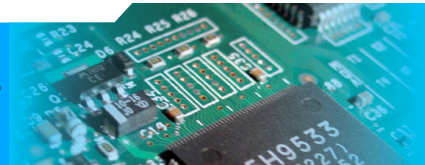
Encapsulation

- 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



Item #EJ15: Minimize accessibility

- ⦿ Rule: make each type or attribute as inaccessible as possible
 - ⦿ if using private does not break your code, use it!
- ⦿ Why this rule?
 - ⦿ To hide information and therefore to **decouple** architectural parts
 - ⦿ Faster design, reusable design
 - ⦿ No impact on performance, even better allows for tuning



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



Access levels in Java

- protected
 - subclasses and package have access
 - do not use for attributes
- public
 - NEVER EVER** use it for attributes (well almost never)



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



Rule #EJ16

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



Accessors pitfalls

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



Careful with references

- if you 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)



Examples

© `int getSize() { return this.size; }`

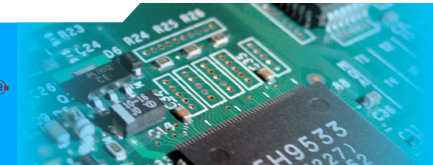
© `// OK`, it's a copy (int values are not objects)

© `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



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



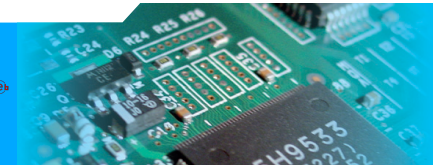
The bad

What's wrong with the following code?

```
class Broken {
```

```
    private Collection myCol;
```

```
    Collection getMyCol() {  
        return this.mycol;}  
}
```



Encapsulation is broken

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



Ways to protect encapsulation

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



Example for collections

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



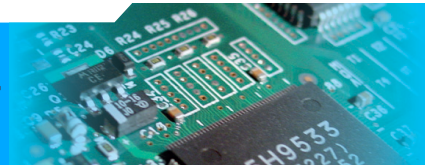
Item #EJ17: Minimize mutability

- What is an immutable object?
 - An object with a state that is defined at initialisation
 - And then that never changes
- Examples for the library
 - String
 - Immutable collections (no specific type, sadly)



Advantages of immutability

- No aliasing problems
- No concurrent modification problems: much easier to parallelise
- Much more secure regarding attacks (eg the check then use pattern)



Ways to provide class immutability

- Do not provide mutators (setters, etc)
- Prevent class extension (final class)
- Of course all attributes must be private
- Make defensive copies of external mutable objects



Item #EJ18: Avoid method inheritance

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



Example

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



Example (cont'd)

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



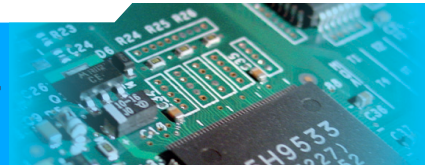
What will be printed?

- `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



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!



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



Example

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



Method inheritance

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



Composition

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



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!)



And the winner is...

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



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.



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



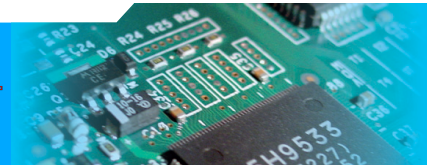
A word on version numbering

- © There should be a relation between version numbers and the different kinds of evolution
- © See example on blackboard



Item #EJ19: document for inheritance or else...

- We have seen that method inheritance has pitfalls
- If you intend your class to be extended by method inheritance
 - document how the overridable operations are used in you class (self used)
- If not, prohibit inheritance
 - final class



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



Use Javadoc

© /**

© * Manages a phone book, as a set of PersonData items.

© * Name duplicates are not allowed.

© *

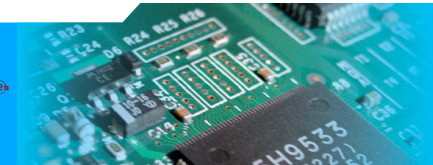
© * @author nplouzeau

© * @version 1.0

© */

© public interface PhoneBook {

©



©/**

©

* 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

©

*

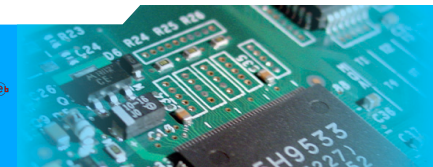
©

*/

©

public PersonData findPerson(String name);

©



©/**

© * Adds a new person data object into the phone book.

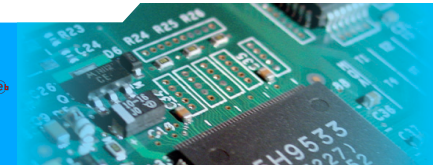
© * @param name name to add

© * @param phoneNumber phone number to add for the name

© * @throws DuplicatePhoneDataException if name already
used

© */

© public void addPerson(String name, String phoneNumber);



Rule #EJ49: Check parameter validity

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



CBD 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



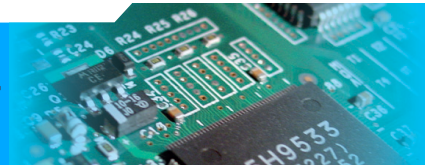
Example: Observer and Stack

- See the Stack and Observer examples on ACO2018
- Notice the use of `Objects.requireNonNull`
 - Throws NPE (null pointer exception) and returns its parameter
-



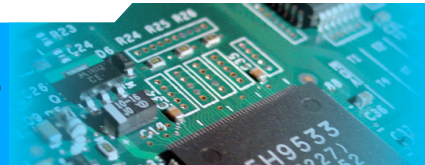
Why use `Objects.requireNonNull`?

- One may wonder whether checking for null pointer parameters early is useful
 - After all, null pointers will be caught
- Very good reasons
 - Detect problems as early as possible before partial execution of the method
 - Detection is guaranteed and problems are much easier to track down
- The choice of NPE by Oracle is a bit clumsy
 - A different exception would have been smarter (eg `NullPointerException`)
- Never ever** ignore problems silently!!!



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



Items #54 and #55: Don't return null

- Some people return null when there is no result
 - Bad idea, in fact null itself is a bad idea (emptiness is not nothingness)
 - Complicates the caller's code and makes it more fragile
- For collections and arrays return types
 - Return an empty collection or array
- For other object types T
 - Return `Optional<T>`



Support for emptiness in Java

- `Collections.emptyList()`
 - Returns an empty immutable list singleton
- `Optional<T> computeSomething();`
 - Return `Optional.empty()` when there is no result
 - Return `Optional.of(someUsefulResult)`
- Do not throw an exception if this is not an exceptional situation (see next slides)



An example of Optional and Collection

🕒 See the AddressBook example on the ACO2018 repository



Remarks on the AddressBook example

- All retrieve operations returns a non null value
 - a collection for findAll
 - an optional for findFirst
- Notice the use of a Stream object
 - More on that next year (for SE students)



Item #EJ69: 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



Exceptions for preconditions

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



Checking for preconditions

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



Example

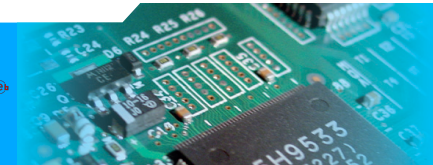
© interface Iterator<T> {

© /**

© *

© @throws InvalidPosition if all items have be returned

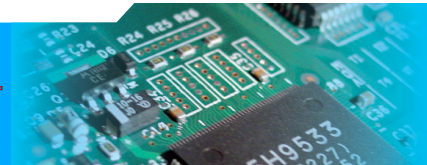
© public T next() throws InvalidPosition;



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 THIS!!!



The proper way

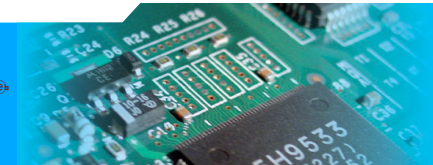
© // Add a retrieve operation

© /**

© * @returns true iff there are more items to read

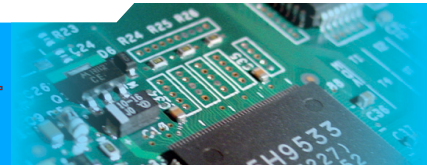
© */

© public boolean hasNext();



Item #EJ73: Match exception with abstraction level

- Software is often organised 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



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...



Item #EJ28: 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

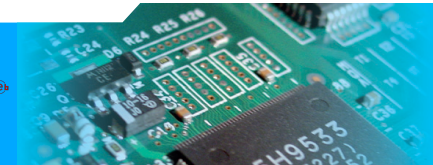


Concretely

© Which code is the most flexible?

1. `String[] names = String[10];`

2. `Collection<String> names = new ArrayList<String>();`



Item #42 to #44: On lambdas

- Three different ways to define operations that can be passed as parameters
 1. Method reference
 2. Lambdas
 3. Anonymous classes
- In the preferred order of use



Anonymous classes

- A way to provide an implementation of an interface without polluting the class namespace

- interface Algorithm {

- public double compute(double x);

- }

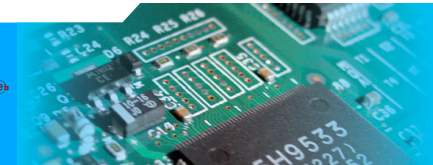
- Algorithm algo = new Algorithm() {

- @Override

- public int compute(double x) {

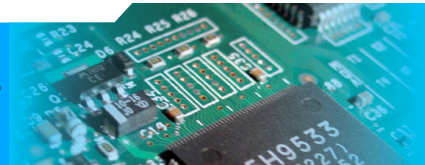
- return x - 1.0;

- }



Lambda expressions

- A special syntax to define an anonymous method and an anonymous class
- `MyListOfDouble mld = ...;`
- `mld.computeOnEach((d) -> d - 1.0);` // Type inference!
- The type of the parameter of `computeOnEach` must be an interface with **exactly one** operation
- To execute the lambda inside the `computeOnEach` method, call that operation



Method references

- Example of reference
 - `Integer::sum` // NOT evaluated here
 - This is different from
 - `Integer.sum(10,20)` // Evaluated here
- See the **LambdasAndStuff** example on ACO2018

