Practical **Immutability** in Java with *Immutables* and *Vavr*

What Object Oriented Programming is

- **Object Identity**
  - Uniquely identify an instance (pointer, reference, address ...)
- **Inheritance** and **polymorphism**
  - Classify and specialize behavior in classifications
- **Encapsulation**
  - Ensure integrity of object 👍
  - Essence of OOP

What Encapsulation is

- A **constructor** should either
  - 👍 construct a **consistent** instance from its parameters
  - 💣 or just fail if it cannot
- Applied on a consistent instance, a **method** should either
  - 👍 modify the object to another **consistent** state
  - 💣 or just fail if it cannot
- Protection of consistency by constructors and methods ensures integrity of object
- Consistency can be described by a set of integrity rules called **class invariant**

Setters == No Encapsulation at All == No OOP

```
public class Customer {
    private int id;
    private String firstName;
    private String lastName;
    public Customer() {}
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
}
```

- What are the integrity rules? How are they protected?
- This is structured programming, it works, but this is not OOP

OOP Revisited

- **Encapsulation is not optional in OOP**
- If you cannot describe (and protect) class invariant, there is no class encapsulation
- Sure, there exists **classes with very weak invariant**:
    - *Forms* which are never guaranteed to be consistent except after validation
    - JPA entity annotated with `@Entity` 💔
    - Or anything similar coming from an external system
- OOP does not require mutability and it works very well with immutability

Immutables

Java annotation processors to generate simple, safe and consistent value objects.

— *From https://immutables.github.io*

- Focused on **immutable classes** with minimum boilerplate
- Does not modify code but generates additional code
- Fully customizable
- Integrates with many **collection** and **option type** libraries
- May look similar to *Lombok* at first sight but is considerably more polished and feature complete

Vavr

Vavr core is a functional library for Java.

— *From http://www.vavr.io*

- Formerly known as *JavaSlang*
- Provides **immutable collections**
- Also provides functions and control structures (such as `Option`)
- Fully interoperable with Java collections and `Optional`
- Requires Java 8 or higher
- Integrates with *Immutables*

Immutable Classes

with *Immutables*

Immutable Class

- **Constructor** returns a new object
- **Methods** do not modify the object but return a **new object** with the modifications applied instead
- For an immutable class, *Immutables* generates
    - a `Builder` to create and modify instances 👍
    - a set of `.withXXX(xxx)` methods to modify instances 👍

Declaring an Immutable Class

```
@Value.Immutable
public abstract class Customer {
    public abstract int id();
    public abstract String firstName();
    public abstract String lastName();
}
```

Creating an Instance

```
final Customer customer =
    ImmutableCustomer.builder()
        .id(1)
        .firstName("John")
        .lastName("Doe")
        .build();
```

Modifying an Instance (one attribute)

```
final Customer modifiedCustomer =
    ImmutableCustomer.copyOf(customer).withLastName("Martin");
```

- Returns a **new instance** that is modified
- Previous instance remains unchanged
- Only **one attribute** modified

Modifying an Instance (multiple attributes)

```
final Customer modifiedCustomer =
    ImmutableCustomer.builder().from(customer)
        .firstName("Paul")
        .lastName("Martin")
        .build();
```

- Several attributes modified with no intermediary instances
- Also allows modifying **multiple attributes** that should remain **consistent** with each other

Calculating an Attribute from Other Attributes

```
@Value.Immutable
public abstract class Customer {
    // ...
    public String fullName() {
        return firstName() + " " + lastName();
    }
}
```

- From the outside, calculated attribute looks exactly the same as other attributes 👍
- **Uniform access principle**

Reminder on Comparing

- By **value**, comparing **attributes** of object
- By **reference**, comparing **object identity** (pointer, address, reference ...)

Comparing Immutable Instances

- Immutable class implies **comparison by value**
- *Immutables* generates consistent
  - `.equals(other)` 👍
  - `.hashCode()` 👍
- Can ultimately be customized by code
- Greatly simplifies unit test assertions 👍

Comparing Immutable Instances

```
final Customer customer1 = ImmutableCustomer.builder()
        .id(1).firstName("John").lastName("Doe").build();

final Customer customer2 = ImmutableCustomer.builder()
        .id(1).firstName("John").lastName("Doe").build();

assert customer1.equals(customer2); // Same attributes
assert customer1.hashCode() == customer2.hashCode();

final Customer customer3 = ImmutableCustomer.builder()
        .id(1).firstName("Paul").lastName("Martin").build();

assert !customer1.equals(customer3); // Different attributes
assert customer1.hashCode() != customer3.hashCode(); // Not a general property!
```

Printing Immutable Instance

- *Immutables* generates useful `.toString()` automatically 👍
- Confidential attributes can be hidden from `.toString()` using `@Redacted`
- Can ultimately be overridden by code
- Simplifies logging 👍
- Simplifies unit test debugging 👍
    - Compare with clipboard trick

Printing Immutable Instance

```
System.out.println(customer.toString());
```

Will output something like

```
Customer{id=1, firstName=John, lastName=Doe}
```

Preventing `null` attributes

- Attributes should never be `null`
  - `null` is evil! 😈
- *Immutables* will reject `null` by default 👍
- Optional attribute should be explicit using an **option type**
  - *Vavr* `Option` is a good ... option 😉
  - More later

Immutables prevents absence of attributes at creation

```
ImmutableCustomer.builder().id(1).build()
```

Will fail with an exception

```
java.lang.IllegalStateException: Cannot build Customer,
some of required attributes are not set [firstName,
lastName]
```

Immutables prevents `null` attributes

```
ImmutableCustomer.builder()
    .id(1).firstName(null).lastName("Martin")
    .build()

ImmutableCustomer.copyOf(customer).withFirstName(null)

ImmutableCustomer.builder().from(customer)
    .firstName(null).lastName("Martin")
    .build()
```

Will all fail with an exception

```
java.lang.NullPointerException: firstName
```

Ensuring Consistency

- Proper encapsulation requires explicit **class invariant**
  - A set of rules that applies to attributes of class
  - and with which all instances must comply
- *Immutables* allows to write a class invariant and will enforce it automatically 👍
- *Guava* also provides `Preconditions` to help

Expressing Class Invariant

```java
@Value.Immutable
public abstract class Customer {
    // ...
    @Value.Check
    protected void check() {
        Preconditions.checkState(
                id() >= 1,
                "ID should be a least 1 (" + id() + ")");

        Preconditions.checkState(
                StringValidation.isTrimmedAndNonEmpty(firstName()),
                "First Name should be trimmed and non empty (" + firstName() + ")");

        Preconditions.checkState(
                StringValidation.isTrimmedAndNonEmpty(lastName()),
                "Last Name should be trimmed and non empty (" + lastName() + ")");
    }
}
```

Immutables ensures invariant at creation

```
final Customer customer =
        ImmutableCustomer.builder()
                .id(-1)
                .firstName("Paul")
                .lastName("Simpson")
                .build();
```

Will fail with an exception

```
java.lang.IllegalStateException: ID should be a least 1
(-1)
```

Immutables ensures invariant at modification

```
final Customer modifiedCustomer =
        ImmutableCustomer.copyOf(customer).withFirstName(" Paul ");
```

Will fail with an exception

```
java.lang.IllegalStateException: First Name should be
trimmed and non empty ( Paul )
```

Immutables ensures invariant at modification

```
final Customer modifiedCustomer =
        ImmutableCustomer.builder()
                .from(customer)
                .lastName("")
                .build();
```

Will fail with an exception

```
java.lang.IllegalStateException: Last Name should be
trimmed and non empty ()
```

Immutable Collections

with *Vavr*

Immutable Collections

- A method that transforms an immutable collection
  - always return a **new collection** with the transformation applied
  - and keep the **original collection unchanged**
- Immutable collections **compare by value**
  - *Vavr* implements `.equals(other)` and `.hashCode()` consistently 👍
- In principle, they **should not accept `null`** as element
  - but *Vavr* does 👿
- Immutable collections are special efficient data structures called **persistent data structures**

Vavr Immutable Collections

Mutable (Java)	**Immutable (*Vavr*)**
`Collection`	`Seq`
`List`	`IndexedSeq`
`Set`	`Set`
`Map`	`Map`

- Collections can be wrapped
  - from Java to *Vavr* using `.ofAll(...)` methods
  - and from *Vavr* to Java using `.toJavaXXX()` methods

Immutable Sequence

```
final Seq<Integer> ids = List.of(1, 2, 3, 4, 5);

final Seq<String> availableIds = ids
        .prepend(0) // Add 0 at head of list
        .append(6) // Add 6 as last element of list
        .filter(i -> i % 2 == 0) // Keep only even numbers
        .map(i -> "#" + i); // Transform to rank
```

`availableIds` will print as

```
List(#0, #2, #4, #6)
```

Immutable Indexed Sequence

```
final IndexedSeq<String> commands = Vector.of(
       "command", "ls", "pwd", "cd", "man");

final IndexedSeq<String> availableCommands = commands
       .tail() // Drop head of list keeping only tail
       .remove("man"); // Remove man command
```

`availableCommands` will print as

```
Vector(ls, pwd, cd)
```

Immutable Set

```
final Set<String> greetings = HashSet.of("hello", "goodbye");

final Set<String> availableGreetings = greetings
        .addAll(List.of("hi", "bye", "hello")); // Add more greetings
```

`availableGreetings` will print as

```
HashSet(hi, bye, goodbye, hello)
```

Immutable Map

```
final Map<Integer, String> idToName = HashMap.ofEntries(
        Map.entry(1, "Peter"),
        Map.entry(2, "John"),
        Map.entry(3, "Mary"),
        Map.entry(4, "Kate"));

final Map<Integer, String> updatedIdToName = idToName
        .remove(1) // Remove entry with key 1
        .put(5, "Bart") // Add entry
        .mapValues(String::toUpperCase);
```

`updatedIdToName` will print as

```
HashMap((2, JOHN), (3, MARY), (4, KATE), (5, BART))
```

Immutable Option Type

with *Vavr*

Option Type

- An option type is a generic type such as *Vavr* `Option<T>` that models the **presence** or the **absence** of a value of type `T`.
- Options **compare by value** 👍
- In principle, options **should not accept** `null` as present value
  - but *Vavr* does 👿

Present Value (some)

```
final Option<String> maybeTitle = Option.some("Mister");

final String displayedTitle = maybeTitle
        .map(String::toUpperCase) // Transform value, as present
        .getOrElse("<No Title>"); // Get value, as present
```

`displayedTitle` will print as

```
MISTER
```

Absent Value (none)

```
final Option<String> maybeTitle = Option.none();

final String displayedTitle = maybeTitle
        .map(String::toUpperCase) // Does nothing, as absent
        .getOrElse("<No Title>"); // Return parameter, as absent
```

`displayedTitle` will print as

```
<No Title>
```

Bridging with Nullable

From nullable to `Option`

```
final Option<String> maybeTitle =
        Option.of(nullableTitle);
```

From `Option` to nullable

```
final String nullableTitle =
        maybeTitle.getOrNull();
```

Immutable from Classes to Collections

with *Immutables* and *Vavr*

`Customer` with an Optional Title

```java
@Value.Immutable
public abstract class Customer {
    public abstract Option<String> title();
    public abstract int id();
    public abstract String firstName();
    public abstract String lastName();
    // ...
}
```

Preventing `null` in Title `Option`

```
@Value.Immutable
public abstract class Customer {
    // ...
    @Value.Check
    protected void check() {
        Preconditions.checkState(
                title().forAll(Objects::nonNull), // Fix Vavr :-)
                "Title should not contain null");
        // ...
    }
}
```

Creating a `Customer` without a Title

```
ImmutableCustomer.builder()
        .id(1)
        // Does no set optional attribute
        .firstName("Paul")
        .lastName("Simpson")
        .build();
```

- Assigns `Option.none()` as title
- Will print as

```
Customer{title=None, id=1, firstName=Paul, lastName=Simpson}
```

Creating a `Customer` with a Title

```
ImmutableCustomer.builder()
        .id(1)
        .title("Mister") // Sets optional attribute
        .firstName("Paul")
        .lastName("Simpson")
        .build();
```

- Assigns `Option.some("Mister")` as title
- Will print as

```
Customer{title=Some(Mister), id=1, firstName=Paul, lastName=Simpson}
```

Unsetting Optional Title

```
ImmutableCustomer.copyOf(customer).withTitle(Option.none());
```

Or

```
ImmutableCustomer.builder().from(customer)
         .unsetTitle()
         .build();
```

Setting Optional Title

```
ImmutableCustomer.copyOf(customer).withTitle("Mister");
```

Or

```
ImmutableCustomer.builder().from(customer)
        .title("Miss")
        .firstName("Paula")
        .build();
```

TodoList class

```
@Value.Immutable
public abstract class TodoList {
    @Value.Parameter public abstract String name();
    public abstract Seq<Todo> todos();

    public static TodoList of(final String name) {
        return ImmutableTodoList.of(name);
    }
    // ...
}
```

TodoList Invariant

```
@Value.Immutable
public abstract class TodoList {
    //...
    @Value.Check
    protected void check() {
        Preconditions.checkState(
                StringValidation.isTrimmedAndNonEmpty(name()),
                "Name should be trimmed and non empty (" + name() + ")");

        Preconditions.checkState(
                todos().forAll(Objects::nonNull), // Fix Vavr :-)
                "Todos should all be non-null");
    }
    //...
}
```

Todo class

```
@Value.Immutable
public abstract class Todo {
    @Value.Parameter public abstract int id();
    @Value.Parameter public abstract String name();
    @Value.Default public boolean isDone() { return false; };

    public Todo markAsDone() { return ImmutableTodo.copyOf(this).withIsDone(true); }

    public static Todo of(final int id, final String name) {
        return ImmutableTodo.of(id, name);
    }
    // ...
}
```

Todo Invariant

```
@Value.Immutable
public abstract class Todo {
    // ...
    @Value.Check
    public void check() {
        Preconditions.checkState(
                id() >= 1,
                "ID should be a least 1 (" + id() + ")");

        Preconditions.checkState(
                StringValidation.isTrimmedAndNonEmpty(name()),
                "Name should be trimmed and non empty (" + name() + ")");
    }
}
```

Adding and Removing Todo

```
@Value.Immutable
public abstract class TodoList {
    // ...
    public TodoList addTodo(final Todo todo) {
        return ImmutableTodoList.builder().from(this).addTodo(todo).build();
    }

    public TodoList removeTodo(final int todoId) {
        final Seq<Todo> modifiedTodos =
            this.todos().removeFirst(todo -> todo.id() == todoId);

        return ImmutableTodoList.copyOf(this).withTodos(modifiedTodos);
    }
    // ...
}
```

Marking Todo as Done

```
@Value.Immutable
public abstract class TodoList {
    // ...
    public TodoList markTodoAsDone(final int todoId) {
        final int todoIndex = todos().indexWhere(todo -> todo.id() == todoId);

        if (todoIndex >= 0) {
            final Seq<Todo> modifiedTodos = todos().update(todoIndex, Todo::markAsDone);
            return ImmutableTodoList.copyOf(this).withTodos(modifiedTodos);
        } else {
            return this;
        }
    }
    //...
}
```

Counting Pending and Done Todos

```
@Value.Immutable
public abstract class TodoList {
    // ...
    public int pendingCount() {
        return todos().count(todo -> !todo.isDone());
    }

    public int doneCount() {
        return todos().count(todo -> todo.isDone());
    }
}
```

Creating and Manipulating `TodoList`

```
final TodoList todoList = TodoList.of("Food")
        .addTodo(Todo.of(1, "Leek"))
        .addTodo(Todo.of(2, "Turnip"))
        .addTodo(Todo.of(3, "Cabbage"));

final TodoList modifiedTodoList = todoList
        .markTodoAsDone(3)
        .removeTodo(2);
```

In Everyday Life

What about *Spring MVC*, *Jackson*, *Hibernate* ...

Support for Common Technologies

	Immutables	**Vavr**
Spring MVC	😄	😄
Jackson	😄	😄 `vavr-jackson`
Bean Validation	😐 `getXXX`, custom style	😄 `vavr-beanvalidation2`
Spring Data	😐	😄
Hibernate	😟	😟
jOOQ	😄	😄

Hibernate, or not Hibernate, that is the question

- **Hibernate** requires absence of encapsulation 👿
  - Mutable classes
  - Mutable collections
- Facade Hibernate!
- Or use **jOOQ**