Type-driven development

The joy of types
Hallmarks of a good API

• Typesafe
  • Errors are caught as early as possible

• Simply typed
  • No complex generic types

• Friendly
  • “What’s common should be easy and simple, what’s uncommon should be possible”

• Fluent and Readable

• Discoverable
  • Easy to understand how to use
Typesafety

- Typesafety means catching as many errors as possible during compile time
  - And failing as early as possible during run time
- The prime nemesis of typesafety are:
  - Down-casting
    - Inherently not typesafe, but sometimes necessary when interfacing with legacy APIs
  - Primitive obsession and String Typing
    - Using primitives when a more appropriate type is appropriate
    - Lifting primitives to a higher level of abstraction than they deserve
  - Stringly typed
    - More specific, but more common than primitive obsession
Down-casting

- e.g.,

```java
Object o = new Object();
String s = (String) o;
```

- Down-casting errors are usually more **subtle** than that...
- Unless you absolutely, positively, "know what you're doing", avoid down-casts
  - Examples of "I Know what I'm doing": you're the owner of the object you're down-casting, and you have to use down-casting due to **legacy/external API**, e.g., serialization
  - The code is **exactly below** an `instanceof` check (still not as typesafe as other languages, e.g., Kotlin, Ceylon)
- Use other `patterns` to get around this, e.g., `Visitor`, or more advanced language constructs like `generics`
Primitive Obsession

- Examples include age as `int`, currency as `double`, ID as `long`...
- For example, why age isn't an `int`
  - Age is much more **restricted** than `int`
    - Age can’t be **negative**
    - Age probably can’t surpass a relatively small number (130? 200? Definitely not $2E9$)
    - Arithmatic on ages usually doesn't make sense (what's the semantic meaning of $age1 * age2$?)
- Every piece of code has to re-validate its input
- Where do you place methods related to age? (`canBuyAlcohol()`, `canVote()`, etc.)
- Can pass the wrong type as age (`timeout`, `numberOfChildren`, `flags`, `arrayIndex`)
- We allow buggy code, e.g., negating age
- What happens if you want to **redesign** age to use `date of birth` instead of hard coding the number?
Primitive Obsession: Pros

There are some lures to using primitives

• Primitive literals
  • You can’t write in Java `Age age = 4`
  • This can cause your API to become extremely verbose

```java
Person p = new Person(new Age(18),
  new FirstName(new Name("John")),
  new LastName(new Name("Smith")));
```

• No need to reinvent the wheel
  • Comparing ages, getting the difference between ages
  • But we also don’t have needless functionality, such as multiplication

• The outside world does not know our DSL
  • We have to convert to and from a more universal datatype
(no) Value classes in Java

- Languages without **value classes** make for very **verbose** class definitions
  - Tools such as **FreeBuilder**, **AutoValue** and **Project Lombok** can help

```java
public class Age {
    private final int age;
    public Age(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        Age age1 = (Age) o;
        return age == age1.age;
    }
    @Override
    public int hashCode() {
        return age;
    }
    @Override
    public String toString() {
        return "Age{age=" + age + "}';
    }
}
```

```java
@FreeBuilder
public interface Age {
    int age();
}
```

```java
@Data // Lombok
public class Age {
    private final int age;
}
```

```java
// other languages…
// Scala
case class Age(age: Int) extends AnyVal
// Kotlin
data class Age(age: Int)
// Haskell
data Age = Age Int deriving (Show, Eq, Ord)
```
Typesafety: Stringly typed

A more specific, and much more common, example of primitive obsession is using strings when a more appropriate type exists

- The following things are really strings:
  - User input – *unparsed* arguments, e-mail contents, Facebook posts, etc.
  - The contents of a text file
  - Raw data passed between different layers of code e.g., *unparsed* HTTP requests

- Unfortunately, our code often fails to parse and convert the above

- The following things are *not* strings:
  - IDs, names, file paths, enums, business objects, addresses, URLs, XMLs/JSONs
  - Whenever your function accept a string, ask yourself: "Is the entire catalogue of Shakespeare sonnets a valid input?"
    - If it's not, then it's not a String
  - If you use strings for Intra-process messages, or get them from a user, parse them as soon as possible
Primitive parameters

- Primitives not only hurt Typesafety, but they also reduce readability

```java
void connect(int timeoutInMillis) { ... }
connect(500); // what does this number mean again?
```

- If we used types, it would be much clearer:

```java
connect(Timeout.inMillis(500));
```

- Especially egregious is the use of boolean parameters

```java
void foo(boolean doTheThing) { ... }
// in a far away place in the code
foo(true) // what's true?
```

- Alternatives: enums and new methods (not overloads!)

```java
void foo() { ... }
void fooWithTheThing() { ... }
// Or use enums (yes, even if it has only two instances!)
void foo(ThingChoice tc) { ... }
foo(ThingChoice.DO_THE_THING);
```
Generic names and code smells

- Prefer to use **generic names** whenever possible
  ```java
  void Foo(String s, int n, double x, ...);
  ```
- Shorter to read and write
  - `String` string provides **no more information** than `String` `s`
- This is not possible when you have to **qualify** your parameters with additional information
  ```java
  void Foo(String url, int timeIntMillis, double ratio);
  ```
- Whenever you have to use non-generic names, you are relying on "name enforcement" instead of type enforcement
  - And names don’t enforce
  - Shorter names help convey **meaning**: we have no **additional requirements**
- Comments and descriptive parameter names are **code smells**
- Not only applicable to **primitives**
  ```java
  void Foo(User adminUser); // what if it's not an admin?
  void Foo(AdminUser adminUser); // better, safer
  ```
Failing at runtime

If you **have to defer** checks to run time, do it **as early as possible**

- Ideally at *construction* time
  - Prevent illegal objects from **ever existing**!
- If not, then when accepting parameters
  - Don't delay checks until you actually *need* the object
  - E.g., check for null parameters if you don't use them **right away**
- Don't fail silently!
  - If you can, **crash** the program
  - If you can't, **log** the error (and make sure you check the logs!)
- Garbage-in-Garbage-out is a **horrible** policy
  - Especially the garbage-out part
  - Either **fail** (preferable), or **fix** the input (if you have to)
- Fail **early**, fail **often**
Unrepresentable parameters

Make certain kinds of errors fail at compile time

• For example:

```java
class NonEmptyList<T> {
    public static <T> NonEmptyList<T> of(T... ts) {
        if (ts.length == 0)
            throw new IllegalArgumentException("List must be non-empty!");
        // rest of code
    }
}
```

• Making empty lists illegal

```java
class NonEmptyList<T> {
    public static <T> NonEmptyList<T> of(T first, T... ts) {
        // NonEmptyList.of() doesn't compile anymore!
    }
}
```
Unrepresentable states

Make illegal states unrepresentable, i.e., can't be instantiated

• For example:

class Connection {
    boolean isConnected;
    public void connectTo(ConnectionAddress address) {
        if (isConnected)
            throw new IllegalStateException("Already connected; disconnect first!");
        // ... connection logic
    }
    public void disconnect() {
        if (!isConnected)
            throw new IllegalStateException("Nothing to disconnect from!");
        // ... disconnection logic
    }
}

• Encapsulating state with types:

class OpenConnection {
    // no need keep and check state!
    ClosedConnection disconnect() { ... }
}
class ClosedConnection {
    OpenConnection connectTo(ConnectionAddress address) { ... }
}
Typesafety: Consequences

We get the same benefits we get from using a static programming language, but boosted

- Faster development time
  - There’s a world of difference between a compiler error and a test failure
- Confidence in code
  - Easier to refactor
  - Less brittle code
- Can get by with less unit tests
  - No need to test what the compiler guarantees
  - Tests only show that your code works on specific input, types prove that they work on all inputs

Cons?
- Increased verbosity
- Compiler can “get in the way”
  - But it's probably better than failing in run time anyway...
Simple(r) types

• Composed generic types can quickly get out of hand
  • List<Double> is probably fine
  • Map<Pair<Long, Double>, List<List<Double>>> on the other hand...

• Using nested generic types is a subtle form of primitive obsession
  • Can’t use overload because of type erasure
    • Can’t have two methods named
      foo(List<String> strings) and foo(List<Integer>)
  • A List<Person> can be any list of Person
  • Can’t promise: non-empty lists, non-null elements, all children have the same parent...

• So are we missing typedefs?
  • Yes and no: typedefs aren’t branding, i.e., not typesafe
Simple(r) types (cont.)

Children, Students, Employees ensure safety, convey semantic meaning, and are (much) shorter to write

- We can add useful methods to each type
  - `getAverage()` on Students, `getParent()` on Children

- We can use inheritance to give us access to all the original Collection’s methods
  - Children `extends` ArrayList<Person>

- But if we use inheritance, it might be impossible to change implementation later (HashMap vs. TreeMap, ArrayList vs. LinkedList)
  - Can usually use the IDE to provide us free delegation code
Constructors

• It's hard to differentiate between overloaded constructors, since they have no names
  • Even if you only use overloading for emulating default parameters, its semantics still aren't clear

• **Static factory methods** are better than `new`, since they are named
  • In general, they are more declarative than `new` (for a full list of advantages, see [Effective Java Item #1](https://www.amazon.com/Effective-Java-2nd-Edition-Programming/dp/0321356481))

• But **builders** are better than factory methods
  • Static factory method give names to constructors
  • Builder objects give explicit names to parameters (no named parameters in Java)
Friendliness

• Avoid boilerplate, make the common things easy
  • Essence over ceremony, convention over configuration

• How much code do clients have to write in order to use your API?
  • If your API parses XMLs, how many lines of code does it take to print them?
  • If you need to send an email, how many lines are needed to actually send a simple email to a single address?

• Consider exposing an external primitive interface, and using a more type-safe API internally
  • Similar to checked exceptions: useful when used internally, annoying when exposed
Fluency and Readability

- Your types should be at the level of abstraction of your application
  - Nouns are classes, verbs are methods, yada yada yada...
  - Code is more readable and more declarative
    - Your type and method names match the problem domain
    - Build higher level abstractions from lower level ones
- Your code should read like prose, not like instructions to a computer
  - Use prepositions in your method names
    - buildFrom, applyTo, andThen, assertThat
  - "Programs must be written for people to read, and only incidentally for machines to execute"
Typesafety and Discoverability

• Discoverability: the ease of using and discovering features of our API
  • One of the powerful advantages of joining OOP, a static type system, and IDEs
• Types increase the **discoverability** of your code
• Example: configuration

```java
config(Map<String, String> config)
```

• What is the set of allowed keys (configuration options)? What is the set of allowed values for each key?
• Typos cause runtime errors (at best)
• Maps are mutable, what if someone changes it after it's passed from the outside?
• We have to resort to the (hopefully existing) documentation to figure this out

```java
config(Configuration config)
```

• Configuration options are methods; no need to read documentation, everything is available from the IDE
• Typos aren't possible
• Can be immutable
Discoverability (cont.)

- With the correct design, and a powerful IDE, clients can almost get **self-writing code**!
  - Methods, constructors, and parameters can be auto-completed
  - Looking functionality related to `Age`?
    Type "`Age. <C-Space>`" to see all applicable methods
  - Method `Foo` requires parameter `Bar`?
    Type "`Bar. <C-Space>`" to see all the static factories of `Bar`

- Although type-oriented code generally has **more boilerplate** code, the actual **development time** is usually **reduced**
  - Typos and errors are caught in compile time
    - usually while typing them!
  - No need to write as many tests
    - If it compiles, it probably works