Coding Idioms

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand what code idioms are and what they describe
- Understand which idioms apply to java coding and practice
- Use these idioms in your development practice

Inspiration

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." – C.A.R. Hoare

Introduction

- Programming idioms are “smarter/better” ways to get something done in your favorite language.
- Idioms often amount to some fixed pattern of syntax; low-level idioms are always syntactic.
- Your goal in becoming a code guru is to **expand your own library of programming idioms.**

For Python, Jeff Knupp’s **Writing Idiomatic Python** is a good resource for low-level Python idioms. The Andela Way has a summary of a few of these; The Python Programming Wikibook lists a few as well.

For Java, a good book describing both high- and low-level programming idioms is **Effective Java** by Bloch.

We will super briefly go over some of Bloch’s high-level idioms.

Be Immutable Whenever Possible

(Bloch's Item 15)

- An immutable object is one which doesn't change.
- Many objects are just naturally immutable, they don't change over time. Coffee cups. Pens. etc.
- Immutable = Static = Declarative = Mathematical = Good
  - Put another way, you always know what an immutable object is by reading its definition.
  - No change could have altered its meaning, Just like a definition in a mathematics textbook.
  - The technical term for this property is referential transparency
- Key Corollary: immutable objects can be shared freely without risk. Thus they are always thread-safe.
- So, if possible be immutable. Even being mostly immutable is better than not being immutable at all.
- Related point: initialize the object in the constructor, not afterwards – objects in the "limbo" state in-between are bug magnets.

Bloch example 1

Be Immutable Whenever Possible

There is strong precedence for this concept in programming languages.

- Functional programming languages (e.g. Haskell): all objects in core are immutable.
- Extended functional programming languages (ML): objects are immutable unless you declare them mutable via `ref`.
- Smalltalk et al: method arguments are always immutable.
- Java: make only the fields that will change mutable, make the rest `final`. Recall that `final` fields cannot change after initialization.
- In general, use as many `final` field declarations as you can, a part-immutable object is better than a fully mutable one.

Immutable Objects in Java

- Make all fields private and final, and not providing any mutating methods.
- Be careful about how shallow/deep the immutability is: a Java `final` field itself can't change, but objects in the field can have components change if they have non-`final` fields (that's the point of the example above).
- Deep immutability is the one with good mathematical properties.

A nice webpage on final and immutability: the final word on the final keyword.

Minimize Accessibility

(Bloch Item 13.)

We've "talked the talk" about how good encapsulation is; this is one important way you need to "walk the walk" to implement encapsulation.

- Avoid public static fields like the plague: everyone can see them
- Don't forget to get the protected/private modifiers set appropriately on your code.
- Start out with things private and loosen up as needed. Yes those compile errors are annoying when you can't access the field/method you want, but there is a reward later.
- Loosen up in increments: private -> default -> protected -> public
  (default differs from public in that default only allows access within the package and protected offers access to subclasses outside the package)

Array Access Issues

```
// Potential security hole!
public static final Type[] VALUES = { ... };
```

The public array should be replaced by a private array and a public immutable list:

```
private static final Type[] PRIVATE_VALUES = { ... };

public static final List VALUES =
    collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Inheritance

When Using Inheritance be Clear and Robust or … Don't Inherit

(Bloch item 17)

Since inheritance breaks class encapsulation, it can be a very good rope to hang yourself. This is closely related to the Open-Closed Principle (OCP).

- Inheritance breaks the normal class encapsulation boundary: you are mucking with the superclass' code when you override methods. (Think of it this way: the superclass is in effect importing the overridden method from the subclass to use it)
- Only use inheritance when there is really an is-a relationship between the sub and super-class.
  - Example: don't subclass from `HashSet` when your class is e.g. a set of car parts. Car parts aren't hash sets!

Inheritance

- Be explicit on what methods depend on methods that can can be overridden via putting `final` or not on methods.
  - Example: `remove` from collection iterates over the collection so if `iterator` was overridden, its behavior will change.
- When overriding a method, don't change its original purpose. A more significant design refactoring is needed if you feel the purpose is changing.
- Constructors shouldn't invoke overridable methods: the subclass' constructor will not have been run before the overriding method would get run and things could blow up.
- If you aren't keeping the above things in mind, make your class `final`—if you need to inherit you can later clean up the class and remove final.

Bloch Example 2

Favor Composition Over Inheritance

(Bloch Item 16)

This is also very much related to The Open-Closed Principle (OCP).

- Composition is a whole-object holding on to (in a field) a part-object.
- Observe that by delegating some methods sent to the whole to the part (forwarding them) we achieve something similar to inheritance minus ability of overriding to change part's behavior.
- Composition is a weaker relationship than inheritance which doesn't violate class encapsulation – it obeys the OCP
- You can explicitly pick what parts of the part-object that outsiders can see, by forwarding only some messages sent to the whole to the part.
- Composition is an "advanced" OO style – several design patterns use composition in their implementation, including Composite, Decorator, State, ...

Bloch Example 3

Avoid switch whenever possible

(same as Bloch item 20, replace unions with class hierarchies)

- Switches branch on an aspect of an object or objects: the object is passive, and the switch is active – this is data-centric and opposed to the principle of active objects.
- Let the object do the work instead, as a method
- Any `enum` type can be replaced with an inheritance hierarchy, with an abstract superclass (or interface) and a subclass for each discriminant in the enum.
- We are going to cover this one in detail in the Refactoring lectures.

Bloch Example 4

Factory Methods vs. Constructors

Consider using static factory methods in place of class constructors

(Bloch Item 1)

Why? Flexibility!

- Gives constructor a name for its purpose (Smalltalk always does this: all constructors are static methods!)
- Could opt to not make an object in special cases
- Could also make a subtype instead of current type

Bloch Example 5

Java Singletons

(Bloch's Item 2)

Its good to implement a singleton in a manner that guarantees the singleton behavior only.

- Use a static final field whose initializer creates the single instance
- Make a the default constructor private – then outsiders can't make any more of the singleton.

Bloch example

```
//Singleton with static factory
public class Elvis {
  private static final Elvis INSTANCE = new Elvis();

  private Elvis() {
    ...
  }

  public static Elvis getInstance() {
    return INSTANCE;
  }

  ... // Remainder omitted
}
```

And Many More!

The Bloch book has many more. Here is a highlight.

- (Items 8 and 9) Overriding `equals` can be good, but you need to make sure your method satisfies all the things equals needs
  - See the API documentation for what equals needs to obey: it must for instance be transitive, reflexive, symmetric, and have a faithful `hashCode` implemented for it.
- (Item 30) Use enums instead of `int` constants
  - `int` constants are for low-level C programmers only
- (Item 45) Minimize the scope of local variables.
  - If a variable is declared only in the neighborhood where its used, it won't be abused elsewhere in the method. But, your methods should be small enough anyway that this should not be a problem.
- (Item 58) Use checked exceptions for things the caller can recover from and unchecked exceptions for things they can't recover from.
  - (unchecked exceptions are the ones subclassing `RuntimeException` which you don't have to declare `throws` on in method headers).

And Many More!

The Bloch book has many more. Here is a highlight.

- (Item 60) Reuse existing exceptions built into Java whenever possible.
  - Example: for a bad parameter passed to your method use `IllegalArgumentException`. If its null use `NullPointerException`.
- (Item 61) Throw exceptions understandable by the caller only
  - Thus, if there is some low-level exception arising (`SocketClosedException`), catch it and rethrow it as (`PlayerOfflineException`) to keep the grubby low-level thing from leaking up.
- (Item 65) Don't just catch and ignore exceptions
  - This sort of code is "on the edge" and will easily fall apart—the exception indicates a problem and you need to track it down to the source. (In Fowler's terms, this is a bad smell in code)

Are there any questions?