Encapsulation

FIT2099: SEMESTER 1 2018

Where were we?

- Connascence – wanted to minimise
- Somehow associated with *encapsulation*

Encapsulation - some definitions

1. a software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module
2. the concept that access to the names, meanings, and values of the responsibilities of a class is entirely separated from access to their realization.
3. the idea that a module has an outside that is distinct from its inside, that it has an external interface and an internal implementation

cf. data abstraction, information hiding

--IEEE Software Engineering Vocabulary

Page-Jones on encapsulation

1. *Encapsulation* is the grouping of ideas into one unit, which can thereafter be referred to by a single name.
2. *Object-oriented* encapsulation is the packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only via the interface provided by the encapsulation.

   Meilir Page-Jones, Fundamentals of Object-Oriented Design in UML

Information hiding

Parnas (1972): (Describing a second, improved modular design for a system compared with standard techniques used at the time) *: Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings… A data structure, its internal linkings, accessing procedures and modifying procedures are part of a single module.*

Information/implementation hiding is the use of encapsulation to restrict from external visibility certain information or implementation decisions that are internal to the encapsulation structure.
-- Meilir Page-Jones

Mechanisms for encapsulation

- Java was *made* to encapsulate
  - as are many other OO languages – and non-OO ones too!
- Basic unit of Java programs is the class.
- Can restrict access to *anything* in the class to:
  - Within the class only (`private`)
  - Within the package only (no access modifier - default)
  - Only to subclasses and within the package (`protected`)
  - No restrictions (`public`)

Encapsulation boundaries

- An encapsulation boundary is simply something across which visibility can be restricted
  - the class
  - the package
  - Even the scope defined within methods by curly braces {}
- Any calls to methods (or attribute accesses) to anything not in a class (or package) crosses an *encapsulation boundary.*
- You want to minimise these.

Controlling access to classes

- To date, you’ve made classes `public`
- You can also make them *package-private* by omitting `public` from the declaration.
- This is another way to prevent unintended connascence.

Encapsulation in Watch

- The final abstract Watch had the following public methods:
  - Two constructors
  - `tick()`
  - `display()`
  - `testWatch()`
- External code had no access to *anything* else.
  - So those five methods were the *only* opportunities for connascence
- Pretty well encapsulated!

Encapsulation in Java Collections Framework

```
class TreeNode<K extends Comparable<K>, V> {

private TreeNode leftChild;
private TreeNode rightChild;

private K key;
private V data;

...

}
```

Reducing connescence using encapsulation

- Using Java is not enough!
  - Nor is any other language
- Design your system carefully
- Take advantage of feature provided by the language
  - Access control modifiers (`private`, `protected`, etc)
  - Classes, packages

Simple things

- Avoid public attributes
- Only make methods public where necessary
  - Consider a policy of making everything private when you first create it
- Keep the class package-private if not needed!
- Use *protected* sparingly, consider using methods rather than attributes
  - Remember that protected things are accessible to subclasses in *other packages*.

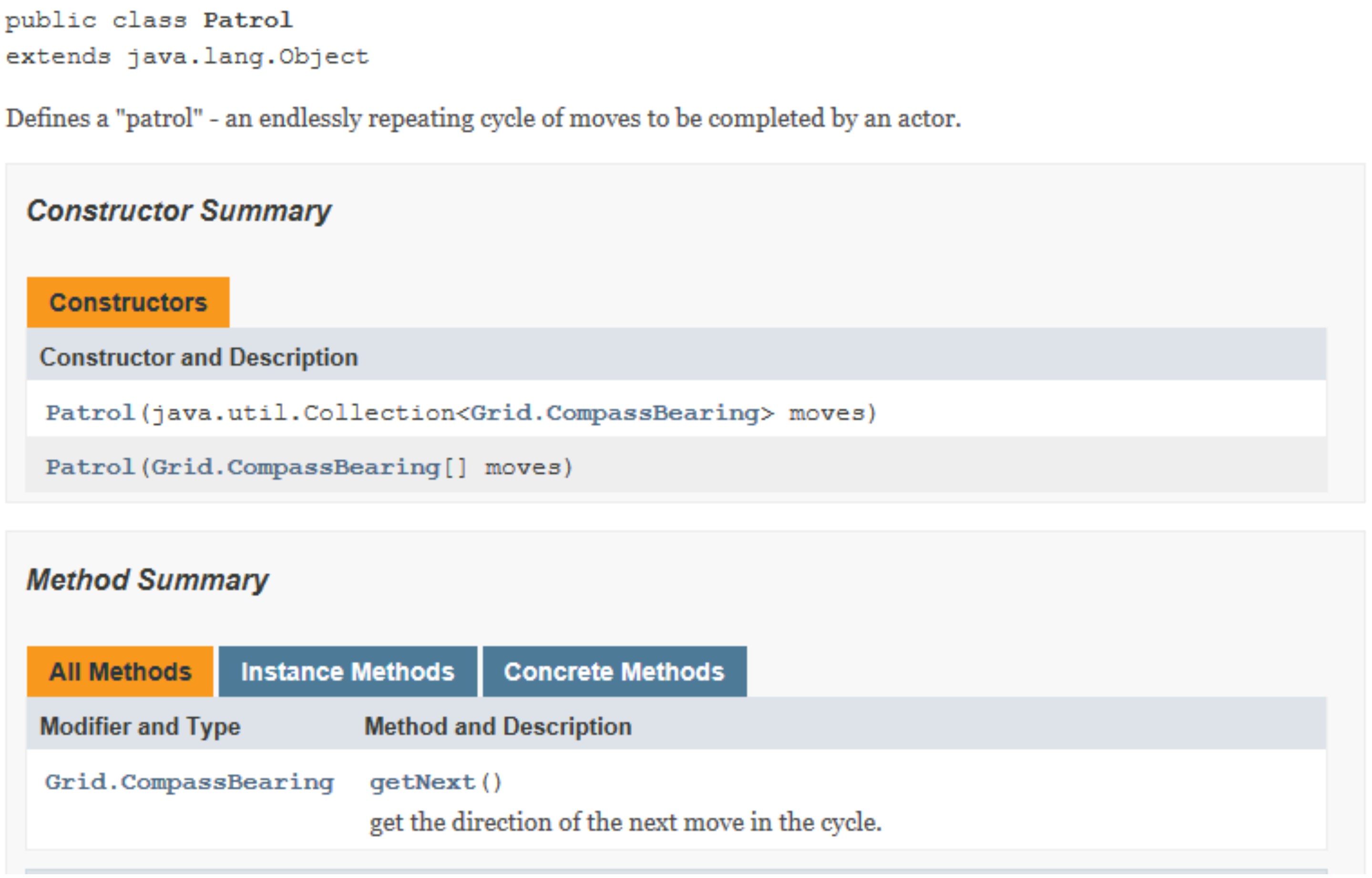
Methods in the right place

```
public class AnnualReport {

    public AnnualReport() {
    …
    }

    public String formatDirector(Director d) {
    return "Name: " + d.getName() + "\n" +
    "Years on Board" + getcurrentYear - d.getStartYear() +
    …
    }
}
```

Minimise interfaces



```
public class Patrol
extends java.lang.Object
```

Defines a "patrol" - an endlessly repeating cycle of moves to be completed by an actor.

Constructor Summary

Constructors

Constructor and Description
`Patrol(java.util.Collection<Grid.CompassBearing> moves)`
`Patrol(Grid.CompassBearing[] moves)`

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`Grid.CompassBearing`	`getNext()` get the direction of the next move in the cycle.

Defensively copy

- When getters return a reference to a private object that is *mutable*
  - i.e. with public attributes or mutator methods other than constructor
- Generally, make a copy and return that
- Otherwise, lose benefit of encapsulation
  - This is called a *privacy leak*
- Lose control of connascence

```
rt java.util.ArrayList;

ic class Stack <T> {
rivate ArrayList<T> items;
ublic Stack() {
    items = new ArrayList<T>();


ublic void push(T newitem) {
    items.add(newitem);


ublic T pop() {
    T last = items.get(items.size() -1);
    items.remove(items.size() - 1);
    return last;


ublic ArrayList<T> getAll() {
    return items;
```

```
public class StackUser {

    public static void main(String [] args) {
        Stack<String> s = new Stack<String>();
        s.push("fred");
        s.push("ginger");
        s.push("john");
        s.push("olivia");
        System.out.println(s.pop());
        StackBreaker whome = new StackBreaker(
        System.out.println(whome.bad(s.getAll(
    }
}
public class StackBreaker {

    public String bad(ArrayList<String>
safeWithMe) {
        String first = safeWithMe.get(0);
        safeWithMe.remove(0);
        return first;
    }
}
```

(Not) exposing implementation details

```
public class TelephoneDirectory {

    private TreeMap<String, String> entries;


    public TreeMap<String, String> getEntries() {
        return new TreeMap<String,
String>(entries);
    }
}
```

Algorithm quirks

```
/**
 * return a Collection containing all of the phone numbers
 * @return the phone numbers
 */
public Collection<String> getNumbers() {
        return new ArrayList<String>(entries.values());
    }
```

```java
public class Couple {

    private Person member1 = null;
    private Person member2 = null;
    public Couple() {
        // TODO Auto-generated constructor stub
    }

    public void setPerson1(Person p1) {
        member1 = p1;
    }

    public void setPerson2(Person p2) {
        member2 = p2 ;
    }

    public String toString() {
        return member1.description() + " " + member2.description();
    }
}
```

Downsides

- Yes, more work initially
- Requires careful thought.
- Payoff later
  - And it is a **BIG** payoff

Summary

- Encapsulation
  - Bundling methods with data in objects.
  - Restricting access to object to the interface.
- Java has powerful access control methods.
  - To entire classes
  - To elements within a class.
- Can and should be used to minimise connascence
  - But it doesn’t just happen!