Using abstraction in Java

FIT2099: SEMESTER 1 2019

Recap: abstraction

We want to make programming easier. To do this, we:

- bundle related pieces of code together into modules
  - not too many, or the modules themselves become hard to maintain
- work out how each module should look from the point of view of other modules
  - minimize the amount of thought that other programmers have to put into using our module
- hide everything else within the encapsulation boundary

Using abstraction at code level

- Abstraction is a design principle rather than a programming technique
  - caution: don't mistake these programming language features for principles of abstraction!
- But it's useful, so most programming languages offer features that support it
- This lecture covers some of the features of Java that can help you write code that makes good use of abstraction
  - you've seen all of them before if you've been keeping up with lectures and readings
  - today we'll do a deeper dive into some of the more complex language features and relate them specifically to abstraction

Classes

The class is the most important mechanism for abstraction in most OO languages, and that includes Java. A well-designed class should

- represent a single concept within your system (with a responsibility that is easy to express)
- expose a public interface that allows it to respond to messages in order to fulfil its responsibility
- hide any implementation details that don't directly fulfil that responsibility
- ensure that its attributes are in a valid condition rather than relying on client code to maintain its state

Visibility modifiers

You've seen these before: `private`, `protected`, `public`

Deciding what to hide and what to expose is a big part of applying abstraction in your code!

Take great care over what is and is not visible from outside your class

- rule of thumb: if in doubt, make it private
- only provide getters and setters if you're sure that external classes need to directly manipulate

Remember, if you leave out the visibility modifier, your class/attribute/method will be visible within the package in which it is declared

- annoyingly, there is no explicit `package` modifier

Abstract classes

As you've seen, abstract classes can't be instantiated

- they may lack important components, such as method bodies
- so they are only useful as base classes
- you might be wondering what they're for

In Java, a subclass inherits all the non-private methods declared in the base class

- so the compiler will let you assign an instance of the subclass to a base class reference:
  `AbstractBaseClass a = new ConcreteSubclass();`
- legal because all methods declared in `AbstractBaseClass` must exist in `ConcreteSubclass`

So client code can be passed an instance of some concrete subclass without ever needing to know its exact type. All it needs to know is that it does everything the abstract class says it can.

- very useful, e.g. for Dependency Injection

Hinge points

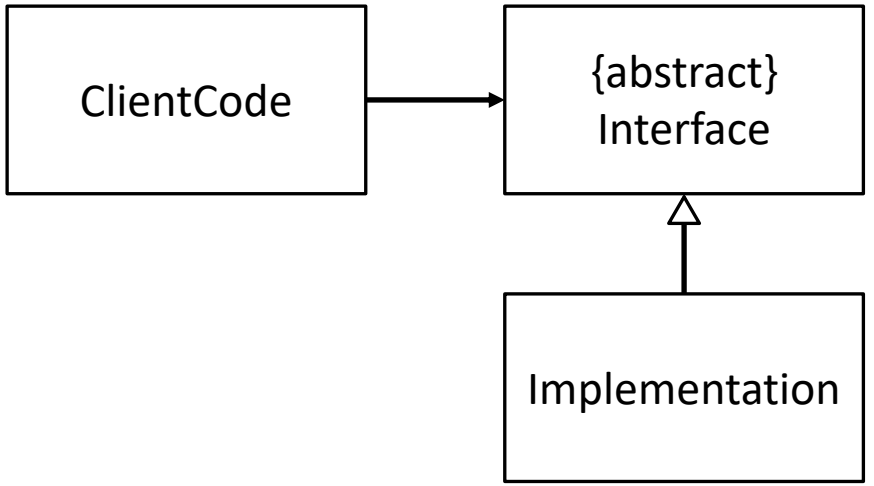
Applying dependency inversion to a single relationship gives you a design that looks like this

This is a powerful technique!

- it means that ClientCode doesn't have to care about anything that happens in the Implementation
- similarly, Implementation doesn't have to care about ClientCode

ClientCode and Implementation can be changed as much as you like provided they both respect the Interface

- this gives your design a kind of "hinge"
- the parts can move around freely except where they are joined
- design is constrained, but as lightly as possible



Packages

As we saw in the last lecture, we don't want our classes to be too large

- they become unmaintainable

But we may need a lot of code in order to implement the features we need

Solution: packages

- group related classes together into a subsystem
- you have seen these in the Watch example from lectures
- the assignment code also uses them extensively – now you know why

Declaring a package is straightforward

- come up with a good descriptive name for it (that's usually the hardest part)
- put `package` *`packagename`* at the top of each class in the package
- move the .java files into a directory called *`packagename`* (your IDE can do this for you)

Nesting packages

Unfortunately, you *can't* put a package inside another package in Java

- `java.util.jar` is not inside `java.util`
- they are separate packages

This means that package-visibility attributes in classes in `java.util` are not visible to methods in `java.util.jar`

- also means that classes in `java.util.jar` must explicitly import classes in `java.util` if they want to use them

But you can still use this `a.b.c` package naming notation to group packages together!

- it still lets you group packages together to simplify interactions in your code
- Oracle uses this notation extensively in its libraries despite the fact that it there is no real nesting going on – ease of use for third-party programming is absolutely critical for libraries

Abstraction layers

An abstraction layer is the publicly-accessible interface to a class, package, or subsystem.

You can create an abstraction layer by restricting visibility as much as possible

- ideally, make methods and attributes private
- if not private, then package (or protected if they may be used in subclasses outside the package)
- one possibility: use dependency inversion to surround anything that might need to change with hinge points; make the interfaces public and keep all other classes at default visibility

Common problem: making too much public

- when programming, it's easy to think of this as making functionality accessible to client code
- but consider: it's harder for developers who want to use your code to understand it if there's hundreds of methods that need to be fiddled with to get anything to work

Well-designed abstraction layers simplify the use of your classes and packages.

Simple and complex abstraction layers

There is a tradeoff between ease of use and power.
The Airbus A380 on the left can take you around the world but qualifying to fly it takes years.
The Holden Colorado in the centre only needs a driver's license.
The switch on the right can be operated by anybody.

Make sure that the complexity of your software interface (API) does not exceed the likely benefit to client code: nobody would buy a car with controls that looked like the Airbus cockpit.

Interfaces

Used extensively in both Java and C#

- they separate the publicly-accessible interface from their implementations
- many interfaces in the Java libraries
- they are Java's only mechanism for multiple inheritance

From a design perspective, an interface can be thought of as an abstract class taken to extremes

There are perfectly respectable OO languages that don't support interfaces at all

- example: C++
- even in C++ you can declare an abstract base class that has no method bodies
- C++ has multiple inheritance so Java-style interfaces are not necessary

Main thing to bear in mind: everything were about to say about the use of abstract classes to define an abstract interface to a component applies to interfaces as well.

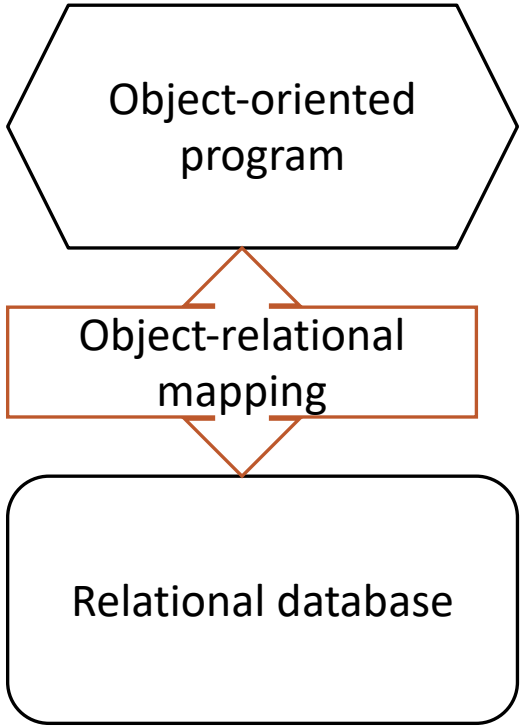
Example: ORMs

It is notoriously painful to combine a relational database with an object-oriented program

- this is due to the so-called object-relational impedance mismatch
- RDBMSes model the world in terms of tables; OOP uses objects that respond to messages
- RDBMS languages such as SQL are typically declarative (i.e. they make statements that define how the world is); all the major OO languages are imperative (i.e. lists of orders about what to do)

Solution: introduce an abstraction layer between the OO side and the DB

- this is called an ORM (Object-Relational Mapping)
- provides classes and objects to the OOP
- maps changes to OO objects to RDBMS commands
- can develop the OO components of the system without having to consider



Generics

Sometimes Java's strongly typed nature gets in the way of good use of abstraction

- example: suppose you want to write a program that requires a set of integers and a set of strings
- the set operations and logic behind these sets would be the same, so we would like to be able to define a Set class to generalize these
- but we'd need to be able to store both Integers and Strings in it…
- would also be nice to be able to re-use this Set code in other projects

Java addresses this problem using generics

- you have seen these in the Collection framework (e.g. `ArrayList<>`)
- essentially, we supply the name of the class as a parameter inside angle brackets
- let's take a closer look at how they developed and how they work

Before generics

One possibility: store as Objects
- every class in Java has java.lang.Object as an ancestor

Can store Integers (or Strings) in an array of Object:

```
Object[] oarr = new Object[100];
oarr[0] = new Integer(3);
```

But can't easily get them out:

```
Integer n = oar[0];  // can't assign an Object to an Integer
```

Can force the compiler to accept this assignment with a typecast

```
Integer n = (Integer) oarr[0];  // tells the compiler oar[0] is an Integer
```

- but typecasts are risky
- and would make our Set harder to use

This is awkward! However, in early versions of Java it was all that was available
- Collections framework appeared in Java 1.2
- current generics introduced in Java 5

Your own generic class

```
public class Set<T> {
    private ArrayList<T> contents;

    public Set() {
        contents = new ArrayList<T>();
    }

    public void add(T newElement) {
        if (contents.contains(newElement))
            contents.add(newElement);
    }

    public boolean isMember(T element) {
        return contents.contains(newElement);
    }

    public Set<T> union(Set<T> otherSet) {
        Set<T> newSet = new Set();
        for (T item: contents)
            newSet.add(item);
        for (T item: otherSet.contents)
            newSet.add(item);
        return newSet;
    }
}
```

Generics and abstraction

With a Set class defined this way, client code can use Set's functionality (e.g. the `add()`, `isMember()`, and `union()` methods shown) to store any datatype without needing typecasts

- can use Set operations without having to consider side effects or implementation details
- if you designed your Set class's public interface well, should be easy to use

But what if you ever want to change the way the Set is implemented?

- combine generics with dependency inversion: make Set an interface
- then implement ArraySet<T>, TreeSet<T>, etc.
- this should sound familiar to you: it's how Java's collection classes are structured!

Bound type parameters

Suppose you want to write a generic collection class that places a restriction on the classes of objects it can store

The problem doesn't arise with a Set

- they don't need to do call any methods on their elements other than `equals()`, which is defined in `Object`

But what if you want to implement a Binary Search Tree?

- need to be able to compare instances of the items you want to store:

```
public void insert(T item) {
        if (item.compareTo(this.key) < 0)
            ...
```

- but the compiler will complain: it can't be certain that T has a `compareTo()` method defined in it
- how can we tell it that it does?

Bound type parameters

We can restrict the kinds of T that can be used to realize a generic:

```
public class BinarySearchTree<T extends Comparable<T>> {
```

This says that in order for objects of class T to be able to be stored in our BinarySearchTree, class T must implement the `Comparable<T>` interface.

- for convenience, you use the `extends` keyword whether you're talking about a class or an interface

The `Comparable<T>` interface has been part of the standard Java library since Java 1.2, and includes only one method signature:

```
int compareTo(T otherObject);
```

So it boils down to "T can be any type, as long as you can compare its instances."

This is similar to the use of abstract base classes and interfaces in polymorphism.

Summary

Java language features that support abstraction

- classes and visibility modifiers
- packages
- abstract classes and interfaces
- generics