Abstraction and Encapsulation in Java

Recap: The Role of Objects

An *object* is a component of a program that represents some entity of interest

- In a banking program: customers, their accounts, cheques, etc.
- In an air traffic control program: planes, airports, beacons, etc.
- In the College database: students, lecturers, courses, etc.

An object can be thought of as a *model* of some real world entity

Recap: The Role of Classes

A *class* is a description of a set of objects that represent the same kind of entity

Each class describes a particular *type* of object

e.g. class Person, class Car, class Airport

Every object is an *instance* of some class

A class can be thought of as a *template* from which different objects can be created

Abstraction 1

An *abstraction* is a description of the essential properties of an entity that are of interest

- Abstractions are *relative* to the perspective of the viewer
- Abstractions focus on the perceived *behaviour* of the entity rather than its implementation
- Abstractions provide an *external* view of the entity

Abstraction 2

Owner thinks of:
- furry
- cuddly
- purrs
- food preferences

Vet thinks of:
- kidney
- stomach
- intestine
- heart
- lung
- liver
- ...

Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Abstraction 3

Classes define *abstractions* of the entities of interest to some program

- A class describes the essential *properties* of the type of entity that its instances represent
- Different classes might be required to describe the same type of entity in different circumstances
- The *behaviour* of the entities is captured in the set of methods provided by the class

Abstraction 4

When designing a class we must provide a set of methods that accurately captures the behaviour of entities of that type

- Initially, we must focus on what methods are required and what role each method plays (rather than how it works)
- Next, we decide on the name, formal parameters and return type of each method - i.e., the method's *signature*
- The result is a complete list of the methods to be provided by the class - i.e., its *interface* or *protocol*

Abstraction 5

Let's define a complex number abstraction

A complex number is a number …

… with a real part, an imaginary part, …

… and a conjugate…

… that can be added to, subtracted from, multiplied by, divided by, or compared with another complex number

(But I don't care how!)

Abstraction 6

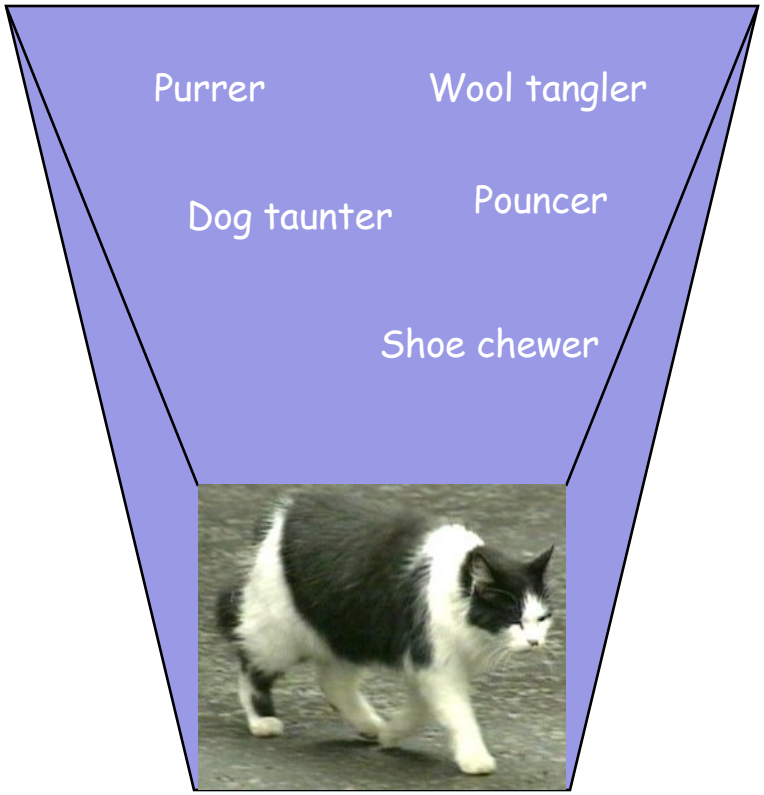
- The interface of a class should provide all the information required to use the class
- Other classes and programs that use this class may now be written (reasonably) independently as they don't need to know how the methods are implemented
- Signatures tell us how to invoke each method individually, but not about the *order* they should be invoked
- Often some methods only make sense after other methods have been invoked previously
- The basic Java language provides no means for specifying meaningful sequences of method invocations, and so comments should be meaningful and helpful in this regard

Encapsulation 1

Encapsulation is the process of hiding all the details of an entity that do not contribute to its essential characteristics

- Encapsulation hides the implementation of an abstraction from its users (or *clients*)
- Encapsulation is often referred to as *information hiding*
- Only the interface to an abstraction should be known to its clients
- How that interface is implemented is hidden from clients

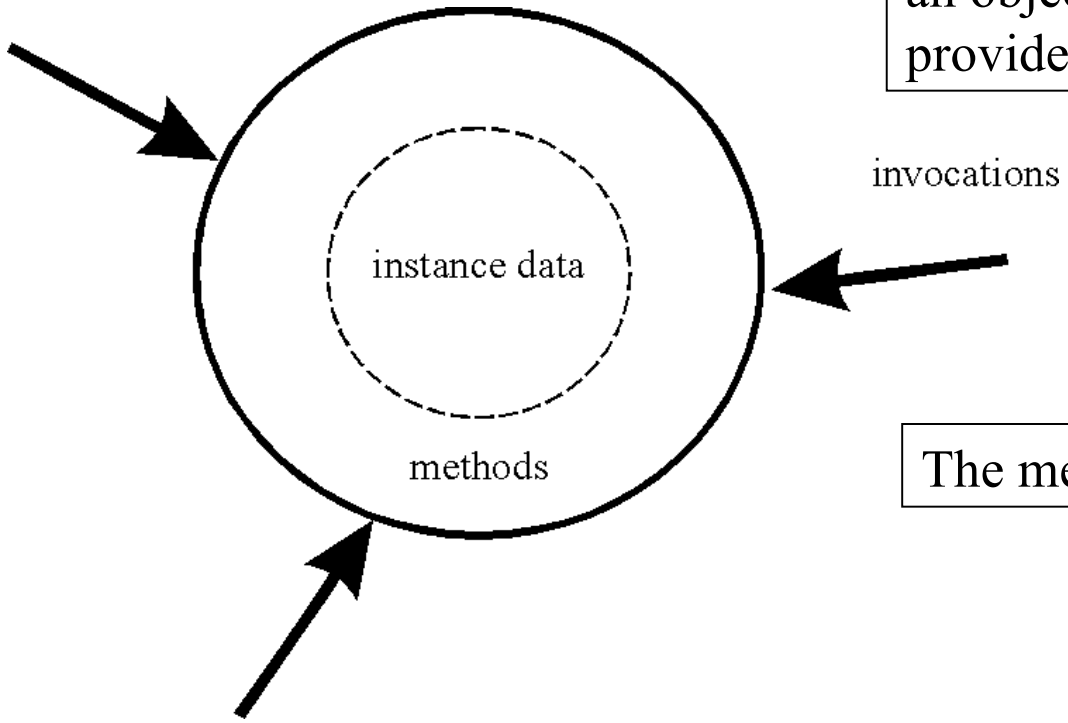
Encapsulation 2



Encapsulation hides the details of the implementation of an object

Encapsulation 3

Encapsulation allows us to hide both how the state of an object is represented and how the object's methods work



The *only* way to access the state of an object should be via the methods provided in its interface

invocations

The methods *encapsulate* the state

Encapsulation 4

Let's define a date abstraction

Some considerations:

- only allow valid dates to be represented:

  31/8/1996 but not 31/9/1996

  29/2/1996 but not 29/2/1997

- define meaningful operations on dates that are likely to be generally useful

Encapsulation 5

Now, we might implement `Date` using three instance variables

```
private int day;   // 1..31
private int month; // 1..12
private int year;  // year AD
```

or using just a single instance variable

```
private int numSeconds; // since 1/1/1900
```

In any case, this choice is invisible to clients of our `Date` class!

Encapsulation 6

Encapsulation:

- separates interface from implementation
- protects validity of abstractions
- ensures that clients of an abstraction only need to know its interface
- allows implementations to be changed at will
- reduces dependencies between parts of a program

Access modifiers

Java allows to distinguish properties of a class that are intended to be visible to its clients from those that intended to be hidden

- So far, all of our objects have been *fully encapsulated*
- I.e., `public` methods and `private` instance variables only
- We can also have `public` instance variables and `private` methods
- `public` and `private` are called *access modifiers*

`public` properties

The keyword `public` is used to designate those properties of a class that constitute the *interface* to the class

- The `public` properties of a class are *visible* to its clients
- The declaration of a `public` property cannot be changed without affecting clients of the class
- Normally, only methods should be made `public`

`private` properties

The keyword `private` is used to designate those properties of a class that constitute *implementation details*

- Clients of a class cannot access its `private` properties directly
- `private` properties are *fully encapsulated*
- In general, all instance variables should be `private`
- This allows the implementation of a class to be changed without exposing the change to its clients

Using `private` instance variables

`private` instance variables can only be accessed within the *class* in which they are declared

- Instances of the same class can access each other's `private` instance variables
- Instances of different classes cannot
- For example, consider class `ComplexNumber`

Using `public` instance variables 1

Using `public` instance variables exposes the implementation details of the class

- Consider another version of class `ComplexNumber`
- Since `re` and `im` are now `public`, they can be accessed directly by any other class
- There is no need for the `getRe` and `getIm` methods
- Other classes can *independently* and *arbitrarily* change the real and imaginary parts of any complex number

Using `public` instance variables 2

- Consider a version of class `Date` that uses `public` instance variables
- Now, we have no choice as to how to represent a date
- Moreover, the `day`, `month`, and `year` variables of any instance of `Date` can *independently* and *arbitrarily* be changed by any other class
- Dates like 31/2/1997, 56/13/1894, or even -34/45/0 are possible!

Using `public` instance data *invalidates* our abstraction!

Summary

- An *abstraction* is a description of the essential properties of an entity that are of interest
- Classes define *abstractions* of the entities of interest to some program
- *Encapsulation* is the process of hiding all the details of an entity that do not contribute to its essential characteristics
- The keyword `public` is used to designate those properties of a class that constitute the *interface* of the class
- The keyword `private` is used to designate those properties of a class that constitute *implementation details*
- Encapsulation protects the *validity* of the abstractions that we define