Lecture 14

We change the way data is stored in the `Time` class definition (specifically by using different instance variables). What we're most interested in are the knock-on effects of the change.

From this, we can continue our discussion of program *design*. We look again at *encapsulation, interface versus implementation* and *cohesion & coupling.* But this time we are looking at these concepts in the context of *data abstraction:* the way we represent data and organise access to the data.

Some of the `Time` class definition

```
public class Time
{
   public Time(int theHour, int theMins,
      int theSecs)
   {  timeInSecs =
         theHour * SECS_PER_HOUR +
         theMins * MINS_PER_HOUR +
         theSecs;
   }

   public static final int
      HOURS_PER_DAY = 24;
   public static final int
      SECS_PER_MIN = 60;
   public static final int
      MINS_PER_HOUR = 60;
   public static final int
      SECS_PER_HOUR =
         SECS_PER_MIN * MINS_PER_HOUR
```



```
public boolean isEqualsTo(Time aTime)
{  return this.timeInSecs ==
      aTime.timeInSecs;
}
```

```
public String getClockTime()
{  return getHours() + ":" +
      getMins() + ":" + getSecs();
}
```



```
public String getMilitaryTime()
{  String milTime;
   String partOfDay = getPartOfDay();
   if (partOfDay.equals(MIDNIGHT_TEXT) ||
      partOfDay.equals(MIDDAY_TEXT))
   {  milTime = partOfDay;
   }
   else
   {  int milTimeHour = getHours();
      if (partOfDay.equals(AFTERNOON_TEXT))
      {  milTimeHour -= MIDDAY_IN_HOURS;
      }
      milTime = milTimeHour + ":" +
         getMins() + ":" + getSecs() +
         " " + partOfDay;
   }
   return milTime;
}
```



```
private int getHours()
{  return timeInSecs / SECS_PER_HOUR;
}
```

```
private int getMins()
{  return timeInSecs % SECS_PER_HOUR /
      SECS_PER_MIN;
}
```

```
private int getSecs()
{  return timeInSecs % SECS_PER_HOUR %
      SECS_PER_MIN;
}
```



```
private int timeInSecs;


private static final String MIDNIGHT_TEXT
   = "midnight";
private static final String MORNING_TEXT
   = "a.m.";
private static final String MIDDAY_TEXT
   = "midday";
private static final String AFTERNOON_TEXT
   = "p.m.";
private static final int MIDDAY_IN_HOURS
   = HOURS_PER_DAY / 2;
private static final int MIDNIGHT_IN_SECS
   = 0;
private static final int MIDDAY_IN_SECS =
         MIDDAY_IN_HOURS * SECS_PER_HOUR;
}
```

(Some of the methods that I omitted would also need changing.)

Changing `Time`'s data representation

- Presently, time data is stored in a single instance variable (seconds since midnight). Suppose, instead, we now decide to store time data in three instance variables (hours, minutes, seconds)
- Such changes affect the *implementation* of the class definition:
  - the way data is represented (more/less memory)
  - the computation that is done in the method bodies (faster/slower execution)
- But the *interface* remains unchanged
  - client code does not need to be changed
  - it does not even have to be recompiled!

Data abstraction

- *Data abstraction*: organising access to a program's data
- Good data abstraction gives readability, re-usablility and maintainability
- *Encapsulation*: 'information hiding'
  - the way the data is represented is hidden (`private` variables)
  - all access to the data comes through the public interface

  This is the distinction between *interface* and *implementation* again (the *what* versus the *how*)

Cohesion and Coupling

- High cohesion: the variables and methods within a class definition should all be directed at a single purpose (i.e. defining some class of object)
  - convoluted or vague class names are a sign of low cohesion
- Low coupling: a class definition should not make details of its implementation (e.g. data representation) visible to its clients
  - variables should be `private`
  - all access to variables should be indirect, via methods in the public interface
  - the alternative, i.e. making variables `public` and allowing clients to directly access them means that changes to the variables might have knock-on effects in the clients

Some advantages of only allowing indirect access

- Setters can check the legitimacy of arguments
- Getters & setters can carry out conversions
- They can keep an object's state consistent, e.g. if updating a variable requires updating another
- They can give the illusion that certain things are stored, when, in fact, they are only computed from what is stored, while giving uniform access to both

Some advantages of only allowing indirect access

- They may be good places to (temporarily) place code that monitors patterns of data access; this may reveal which data structures are causing bottlenecks and need re-implementing
- In concurrent programs, direct concurrent access may lead to inconsistent data, whereas concurrent access through public methods can be synchronised.
- The main advantage: *clients of a class will be unaffected by changes to the class definition's implementation.*