Lecture 13

We write another method for our `Time` class definition.

This introduces the idea of *helper methods*.

We can then discuss some key program *design* concepts: *encapsulation*, the distinction between *interface* and *implementation*, and *cohesion* & *coupling*. We discuss them in the context of *procedural abstraction:* the way we organise the *computation* that a program is to do.

Instance variables & named constants

```
private int timeInSecs;

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;
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;
```


The goal: to write another getter

We wrote this one:

```
public String getClockTime()
{ return timeInSecs / SECS_PER_HOUR + ":" +
     timeInSecs % SECS_PER_HOUR
                / SECS_PER_MIN + ":" +
     timeInSecs % SECS_PER_HOUR
                % SECS_PER_MIN;
}
```

We want to write this one:

```
public String getMilitaryTime()
```

which returns times as phrases such as

midnight
9:30 a.m.
midday
9:45 p.m. (N.B. Not 21:45 p.m.)

Helper methods

Helper methods help other methods get their work done. Typically, they are `private`

```
/* ========================================
       HELPER METHODS
   ========================================
*/

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 String getPartOfDay()
{  String partOfDay;
   if (timeInSecs == MIDNIGHT_IN_SECS)
   {  partOfDay = MIDNIGHT_TEXT;
   }
   else if (timeInSecs < MIDDAY_IN_SECS)
   {  partOfDay = MORNING_TEXT;
   }
   else if (timeInSecs == MIDDAY_IN_SECS)
   {  partOfDay = MIDDAY_TEXT;
   }
   else // timeInSecs > MIDDAY_IN_SECS
   {  partOfDay = AFTERNOON_TEXT;
   }
   return partOfDay;
}
```


Using these helper methods

We can rewrite `getClockTime`:

```
public String getClockTime()
{


}
```

And we can write `getMilitaryTime`:

```
public String getMilitaryTime()
{









}
```

Procedural abstraction

- *Procedural abstraction*: organising your program's computation (the statements) into methods
- Techniques:
  - *Factorization*: identify recurring segments of code and capture them *once* in a single method
  - *Top-down decomposition*: design & code a complex method by successively breaking it down into simpler methods
  - *Bottom-up composition*: design & code a complex method by writing some simple methods and then constructing it out of these

Encapsulation

- Encapsulation: hiding implementational details behind an interface
- Consider electronic consumer goods: sealed boxes
- In software,

  Class definition = interface + implementation

- Interface: the headers of the `public` methods
- Implementation (hidden): method bodies and `private` methods (e.g. helper methods) — clients need know nothing about them
- Use of `private` hides implementation

Cohesion and Coupling

- *The cohesion of a method is the degree to which the statements are directed at a single purpose.*
- High cohesion is desirable: single-purpose methods
  - convoluted or vague method names are a sign of low cohesion
- *Coupling is the degree to which methods depend on each other*
- Low coupling is desirable: methods should communicate with other methods to a minimal degree
  - don't pass unnecessary parameters
  - don't communicate unnecessarily through shared variables

The benefits of good procedural abstraction

- *Greater readability*:
  - detail is encapsulated and hidden within well-named helper methods
  - the whole program may be shorter (if a lot of repetition is now avoided); individual methods are shorter; the contribution of each method is summarised at a high level of abstraction in the method's signature
  - readability aids correctness, debugging, maintainability, etc.
- *Greater maintainability*:
  - due to greater readability
  - with reduced repetition, there is less risk of inconsistent changes

The benefits of good procedural abstraction, cont'd

- *Greater testability*:
  - the program now comprises small, independently testable and debuggable pieces
  - elimination of repetition decreases the amount of code to be tested
- *Greater re-usability*:
  - each smaller method does some single, well-defined task that may crop up in future programs

A possible benefit of good procedural abstraction

- *Possibly even the prospect of better performance!*
  - although we now have the overhead of extra method invocations, this overhead is small (and sometimes compilers can in-line method bodies — though this can be problematic in object-oriented languages)
  - but it may be easier to find bottle-necks now that the code is better modularised
  - and then, because we have eliminated repetition, if I improve the performance of a much-invoked method, all clients of that method will reap the benefit