How to THINK like a Programmer

Problem Solving for the Bewildered

paul vickers

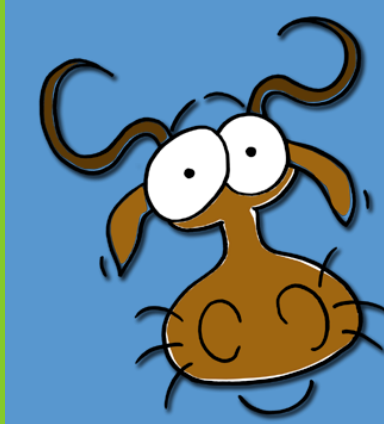
chapter 6

data & control abstractions

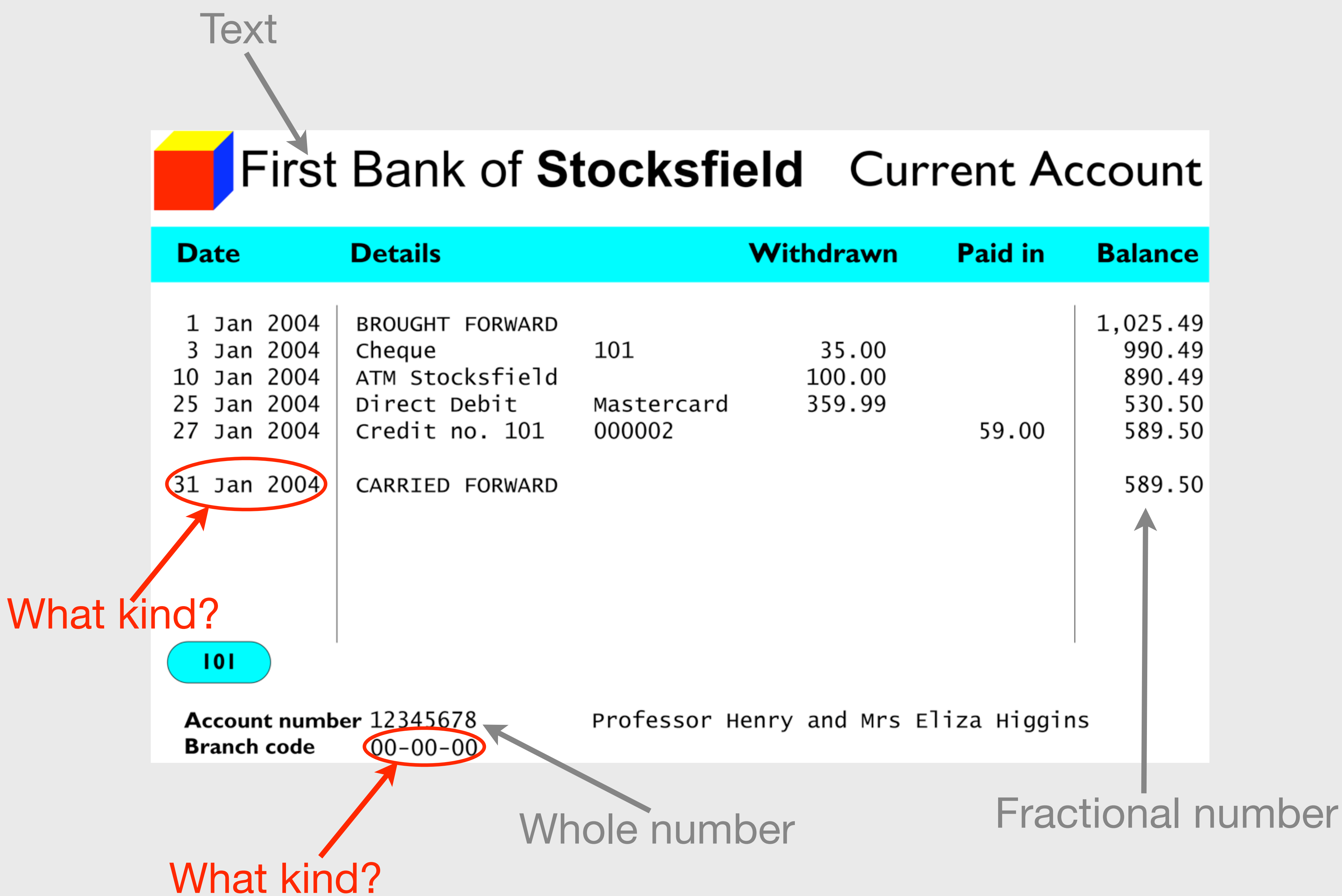
Purpose

- Building a repertoire of standard solutions
- This chapter is concerned with abstraction, specifically:
  - Data abstraction: different kinds of data
  - Control abstraction: some new constructs--
    - Selection: `IF` & `IF...ELSE`
    - Iteration: `WHILE`, `FOR`, and `DO...WHILE`

Data: different kinds



how to think like a programmer



First Bank of **Stocksfield** Current Account

Date	Details		Withdrawn	Paid in	Balance
1 Jan 2004	BROUGHT FORWARD				1,025.49
3 Jan 2004	Cheque	101	35.00		990.49
10 Jan 2004	ATM Stocksfield		100.00		890.49
25 Jan 2004	Direct Debit	Mastercard	359.99		530.50
27 Jan 2004	Credit no. 101	000002		59.00	589.50
31 Jan 2004	CARRIED FORWARD				589.50

101

Account number 12345678
Branch code 00-00-00

Professor Henry and Mrs Eliza Higgins

Data

- The bank statement shows that data comes in different kinds, e.g.
  - Textual data
    - Words, characters
  - Numeric data
    - Whole numbers (integers)
    - Fractional numbers (real numbers, floating point numbers)
- Operating on numeric data
  - Addition +
  - Subtraction -
  - Multiplication ×
  - Division ÷
  - etc ?

Numeric operators

- Different number types respond differently to the some numeric operators
  - +, -, × no problem
  - What about ÷ ?
- The division problem
  - 18 ÷ 9 is easy, the answer is 2
  - What about 17 ÷ 9?

ACTIVITY

Think about the expression 17 ÷ 9. What are some possible answers to it?

17 ÷ 9

- If you reached for your calculator you might have got 1.888888889
- Perhaps you rounded up to get 1.89
- But another answer is 1. 9 goes into 17 one time leaving a remainder of 8.
- The answer depends on whether we wish to deal with fractional numbers (1.89) or whole numbers (1, remainder 8).
- The **type** of number determines what actions can be performed up on it
- Consider the number 7. Four simple things we can do to it are: add, subtract, multiply, divide

Abstract data type

- Defining a data item in terms of the range of values it can take and the actions that can be performed upon it is called an **abstract data type** (ADT)
- ADT for positive whole numbers:

Whole numbers	
Range	0…∞
Operations	+, -, ×, ÷

- The operations are defined in terms of the type
  - multiply and divide make sense for whole numbers but not for textual data
  - What might be the operations allowed for text?
- Abstract because we are not concerned (yet) with how the operations are performed

Why?

- Computers store different data in different ways
  - A whole number may use only 4 bytes of memory
  - A fractional number may use 8 bytes of memory
  - A single character, e.g., ‘Z’ may need only 1 or 2 bytes
- Different types work in different ways
  - 17 ÷ 9 = 1 (dividing one whole number by another)
  - 17 ÷ 9.0 = 1.888888889 (dividing a whole number by a fractional number)
- Data **typing** very important. Programming languages enforce typing rules (some more conservatively than others -- e.g. javascript vs Ada)
- Data typing gives meaning to the operations

ACTIVITY

We know (possibly) what 8 + 8 means, but what does XX + YY mean? Give some possible answers (it depends on what you define '+' to mean in this context)

XX + YY

- Possible solutions:
  - ZZ
  - XXYY
  - YYXX
  - …
- What about ‘A’ + ‘B’?
  - If ‘A’ and ‘B’ are text, then + might mean ‘concatenate’ (add one piece of text to the end of another) to give ‘AB’
  - What if A and B are numbers? In the hexadecimal number system (base 16) A = 10 and B = 11, so A + B = 15
    - 15 in base 16 = 1×16 + 5 units = 21 decimal = 10 +11

Data types

- So, it is important to start identifying the different types the data in our solutions belong to: we will need this information when it comes to writing the solution in a real programming language

Algorithmic building blocks

- Algorithms can be built from only three fundamental (abstract) building blocks
  - Sequence
  - Selection
  - Iteration
- We saw an iteration and selection construct in chapter 4 (`IF` and `WHILE`)

Sequence

- The sequence is the heart of the algorithm for it determines the overall order in which steps are carried out, e.g.
  - A. Get up
    B. Get dressed
    C. Eat breakfast
- Identifying the overall sequence is the first step in the HTTLAP strategy
- Recognizing that there may be more than one valid sequence (ordering) of the actions is important as it can affect the way the algorithm works
  - We might eat breakfast before getting dressed

Selections

- We saw in Chapter 4 how to make basic decisions with the `IF` construct

```
IF (milk required)
   Add milk ;
ENDIF
```

- This is fine for simple decisions: either add milk or do nothing at all (action or *null* action)
- What if we need to do something else if the condition is false?

Extended selections

- Consider an ATM transaction: if money in the account dispense required cash otherwise issue a ‘no funds’ message. Could do this messily:

```
IF (funds available)
   Dispense cash ;
ENDIF
IF (funds not available)
   Display ‘No funds’ message ;
ENDIF
```

- Doesn’t look good though. There is a better way: the `IF...ELSE` construct

```
IF (condition)
   Action 1 ;
ELSE
   Action 2 ;
ENDIF
```

Anatomy of IF...ELSE

```
IF (condition)
   Action 1 ;
ELSE
   Action 2 ;
ENDIF
```

- This time, if the condition is true then Action 1 is carried out otherwise Action 2 is carried out.
- Only **one** of the actions is performed: it’s an *either/or* situation
- Hence:

```
IF (funds available)
   Dispense cash ;
ELSE
   Display ‘No funds’ message ;
ENDIF
```

ACTIVITY

Write an `IF...ELSE` statement that chooses between wearing sandals or shoes depending on whether it is raining

Multi part selections

- What if we need more than just an either/or? What if we have multiple conditional actions? Consider the scenario:
    - Parcels 5 kg and under should be labelled ‘light’, over 5 kg but lighter than 10 kg is ‘medium’, and 10 kg or over is ‘heavy’. Could do this:

```
IF (parcelWeight up to (and including) 5 kilos)
   Add 'light' sticker ;
ENDIF
IF (parcelWeight more than 5 and less than 10 kilos)
   Add 'medium' sticker ;
ENDIF
IF (parcelWeight 10 kilos or over)
   Add 'heavy' sticker ;
ENDIF
Load parcel on van ;
```

- But...

Multi part selections

- ...it looks bad again. What we do is extend the `IF...ELSE`

```
IF (parcelWeight up to 5 kilos)
   Add 'light' sticker ;
ELSE IF (parcelWeight less than 10 kilos)
   Add 'medium' sticker ;
ELSE
   Add 'heavy' sticker ;
ENDIF
Load parcel on van ;
```

- Notice how each condition makes use of the results of the previous one, and the last `ELSE` has no condition

Selection conditions

- Writing things like `parcelWeight up to 5 kilos` is quite verbose
- We can use the **relational operators** to make it neater

Operator	Pseudo-code
Less than	$<$
Less than or equal to	$\leq$
Equals (equality)	$=$
Greater than or equal to	$\geq$
Greater than	$>$
Not equal to (inequality)	$\neq$

Parcels again

- Using the relational operators we get

```
IF (parcelWeight ≤ 5 kilos)
   Add 'light' sticker ;
ELSE IF (parcelWeight < 10 kilos)
   Add 'medium' sticker ;
ELSE
   Add 'heavy' sticker ;
ENDIF
Load parcel on van ;
```

- And

```
IF (age ≥ 18)
   Issue voting card
ELSE
   Display 'Sorry, too young' message ;
ELSE
```

ACTIVITY

What's wrong with the following selection to assign grades?

```
IF (mark ≥ 80)
   grade ← 'A' ;
ELSE IF (mark ≥ 70) AND (mark ≤ 79)
   grade ← 'B' ;
ELSE IF (mark ≥ 60) AND (mark ≤ 69)
   grade ← 'C' ;
ELSE IF (mark ≥ 50) AND (mark ≤ 59)
   grade ← 'D' ;
ELSE IF (mark ≥ 40) AND (mark ≤ 49)
   grade ← 'E' ;
ELSE IF (mark < 40)
   grade ← 'F' ;
ENDIF
```

Fixing the IF...ELSE

- The selection in the last activity did not make use of previous conditions. It should be:

```
IF (mark ≥ 80)
   grade ← ‘A’ ;
ELSE IF (mark ≥ 70)
   grade ← ‘B’ ;
ELSE IF (mark ≥ 60)
   grade ← ‘C’ ;
ELSE IF (mark ≥ 50)
   grade ← ‘D’ ;
ELSE IF (mark ≥ 40)
   grade ← ‘E’ ;
ELSE
   grade ← ‘F’ ;
ENDIF
```

- Why no `IF` for the final `ELSE`?

Nested selections

- Consider the following algorithm for deciding the number of days in a month

```
IF (month = 4) OR (month = 6) OR (month = 9) OR
                                 (month = 11)
   daysInMonth ← 30 ;
ELSE IF (month = 2)
   IF (isLeapYear)
      daysInMonth ← 29 ;
   ELSE
      daysInMonth ← 28 ;
   ENDIF
ELSE
  daysInMonth ← 31 ;
ENDIF
```

The shaded selection is nested within the ELSE part

Iteration

- Iterations (looping structures) come in two flavours:
  - **Determinate** — one in which the number of times the action block is to be repeated is known in advance or can be calculated at the time the loop is executed
  - **Indeterminate** — one in which the number of iterations is unknown in advance and cannot be calculated

Determinate loops

- Examples of determinate loops
  - An algorithm that reports the average rainfall for each month of the year would use a determinate loop: there are 12 months, so the loop will iterate 12 times
  - An algorithm that calculates the number of leap years between two years chosen by the user would also use a determinate loop. The years are not known in advance, but once the user has selected the start and end years, the number of iterations of the loop to process them can be calculated
- Here's one

```
sugarsAdded ← 0 ;
Find out how many sugarsRequired ;
WHILE (sugarsAdded < sugarsRequired)
   Add spoonful of sugar ;
   sugarsAdded ← sugarsAdded + 1 ;
ENDWHILE
```

ACTIVITY

Explain why the loop below is **determinate**

```
sugarsAdded ← 0 ;
Find out how many sugarsRequired ;
WHILE (sugarsAdded < sugarsRequired)
   Add spoonful of sugar ;
   sugarsAdded ← sugarsAdded + 1 ;
ENDWHILE
```

Count-controlled

- These iterations we have just seen are **count-controlled** because a counter variable is used to determine when to terminate
- General pattern:

```
Initialize counter to starting value ;
WHILE (counter < finishing value)
   Actions for loop body ;
   Add increment to counter ;
ENDWHILE
```

Indeterminate iterations

- Number of executions cannot be calculated before entering the loop
- An order processing program for an online retailer might use an indeterminate/indeterminate loop of the form

```
WHILE (orders to process)
   Fetch details of order ;
   Calculate order value ;
   Calculate shipping costs ;
   Print invoice ;
ENDWHILE
```

Read ahead

- The indeterminate loop used in the van loading solution has the general form:

```
Get first item ;
WHILE (continuation condition)
   Process item ;
   Get next item ;
ENDWHILE
```

- Known as a **read-ahead** loop
  - first item to be processed is fetched before entering the loop
  - item replenished as the last action in the loop
  - Used when information about the items to be processed is needed in order to test the loop's controlling condition

Read and process

- Where we do not need to know anything about the items in advance we can use the read-and-process structure

```
WHILE (continuation condition)
   Get item ;
   Process item ;
ENDWHILE
```

Zero or more

- Because the `WHILE` loop tests its condition at the start, the condition can be **false** before any actions are carried out resulting in zero iterations of the loop

```
sugarsAdded ← 0 ;
sugarsRequired ← 0 ;
WHILE (sugarsAdded < sugarsRequired)
   Add sugar ;
   sugarsAdded ← sugarsAdded + 1 ;
ENDWHILE
```

- Thus the `WHILE` is a **zero-or-more** iteration construct
- What about an ATM where we need to enter the ‘enter-your-PIN’ validation loop at least once?

At-least-once iterations

- For the ATM we need an **at-least-once** iteration

```
PIN ← -1 ;
WHILE (PIN ≠ storedPIN)
   Display ‘Enter your PIN’ ;
   Type in PIN ;
ENDWHILE
```

- This solution looks clumsy. Is there a neater way?
  - Yes, but it comes later

A count-controlled construct

- Because count-controlled loops are so common most languages have a dedicated iteration construct for them

Counter variable. Also known as the **loop invariant**

Counter starting value

Counter finishing value

```
FOR variable GOES FROM initial TO final
   Action block ;
ENDFOR
```

FOR example

- Here's an example

```
FOR month GOES FROM 1 TO 12
   Display 'Please enter the month's rainfall' ;
   Get monthRainfall ;
   totalRainfall ← totalRainfall + monthRainfall ;
ENDFOR
```

- And another with calculated start and finish values

```
FOR year GOES FROM age TO age + 10
   Action block ;
ENDFOR
```

- For letters:

```
FOR letter GOES FROM 'A' TO 'Z'
   Display letter ;
ENDFOR
```

ACTIVITY

Consider the following loop

```
FOR year GOES FROM age TO age + 10
   Action block ;
ENDFOR
```

If age = 38, how many times does the loop iterate?

At-least-once construct

- At-least-once loops are also very common so there is another specialized iteration construct, the `DO...WHILE`

Keyword DO denotes start of loop

```
DO
   Action block ;
WHILE (conditional expression) ;
```

Action block is iterated through while (as long as) the conditional expression remains true

The PIN algorithm

- Now we can do the ATM PIN checking properly

```
DO
   Display 'Please enter your PIN' ;
   Get PIN ;
WHILE (PIN ≠ storedPIN) ;
```

Sentinels

- Now read up on sentinels in section 6.5 (p. 152 onwards)

end of chapter 6