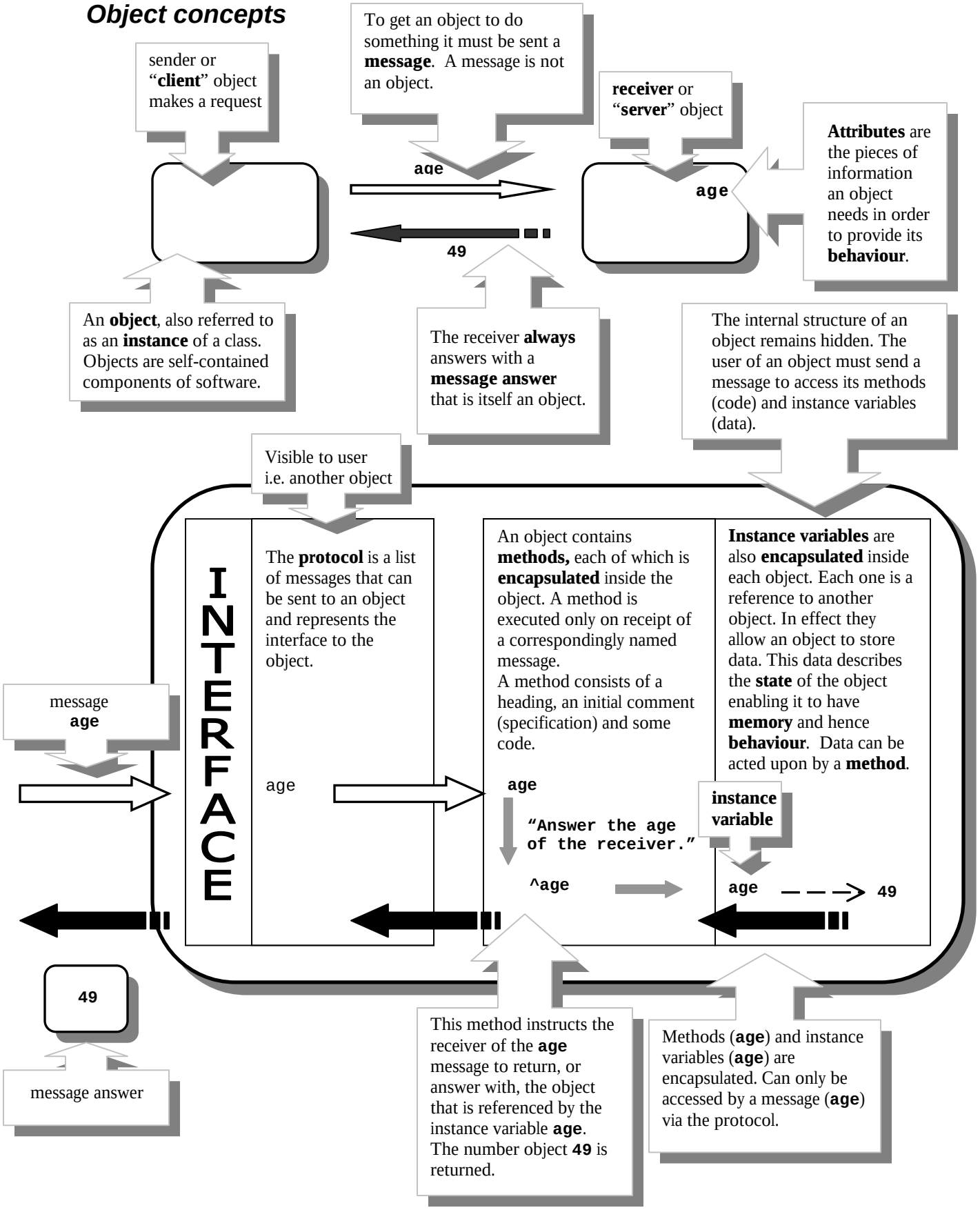
Basic Smalltalk

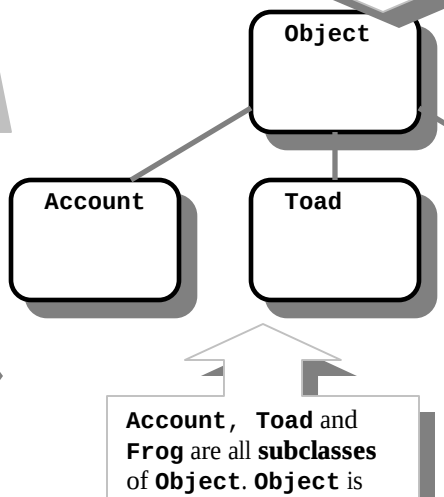
Object concepts



Class and hierarchy

All objects **inherit** the protocol of the class **`Object`**

A class is a **template** for creating instances of that class. All classes are objects.



As subclasses are created, the behaviour of instances of those classes becomes more specific.

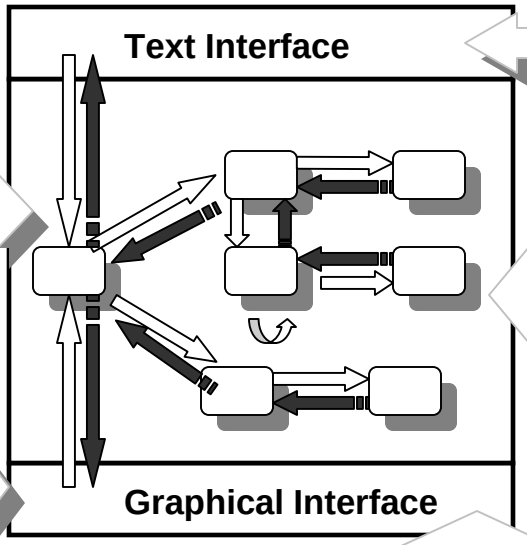
Degree of abstraction increases. Objects become more generalised in their behaviour.

`Account`, **`Toad`** and **`Frog`** are all **subclasses** of **`Object`**. **`Object`** is their **superclass**.

Objects are organised into classes. Objects belonging to the same class (instances of the class) have the same attributes and respond to the same set of messages, responding to each message in an identical manner. Instances of a class are initialised in the same way. Objects of a subclass may or may not be initialised differently from the objects of its superclass.

`HoverFrog` is a subclass of **`Frog`**. Instances of **`HoverFrog`** respond to all the messages for **`Frog`** and will possess further messages in order to implement a more specific behaviour i.e. **`HoverFrog`** instances possess the additional attribute of height

Domain models and user interfaces



Text commands entered at the keyboard.

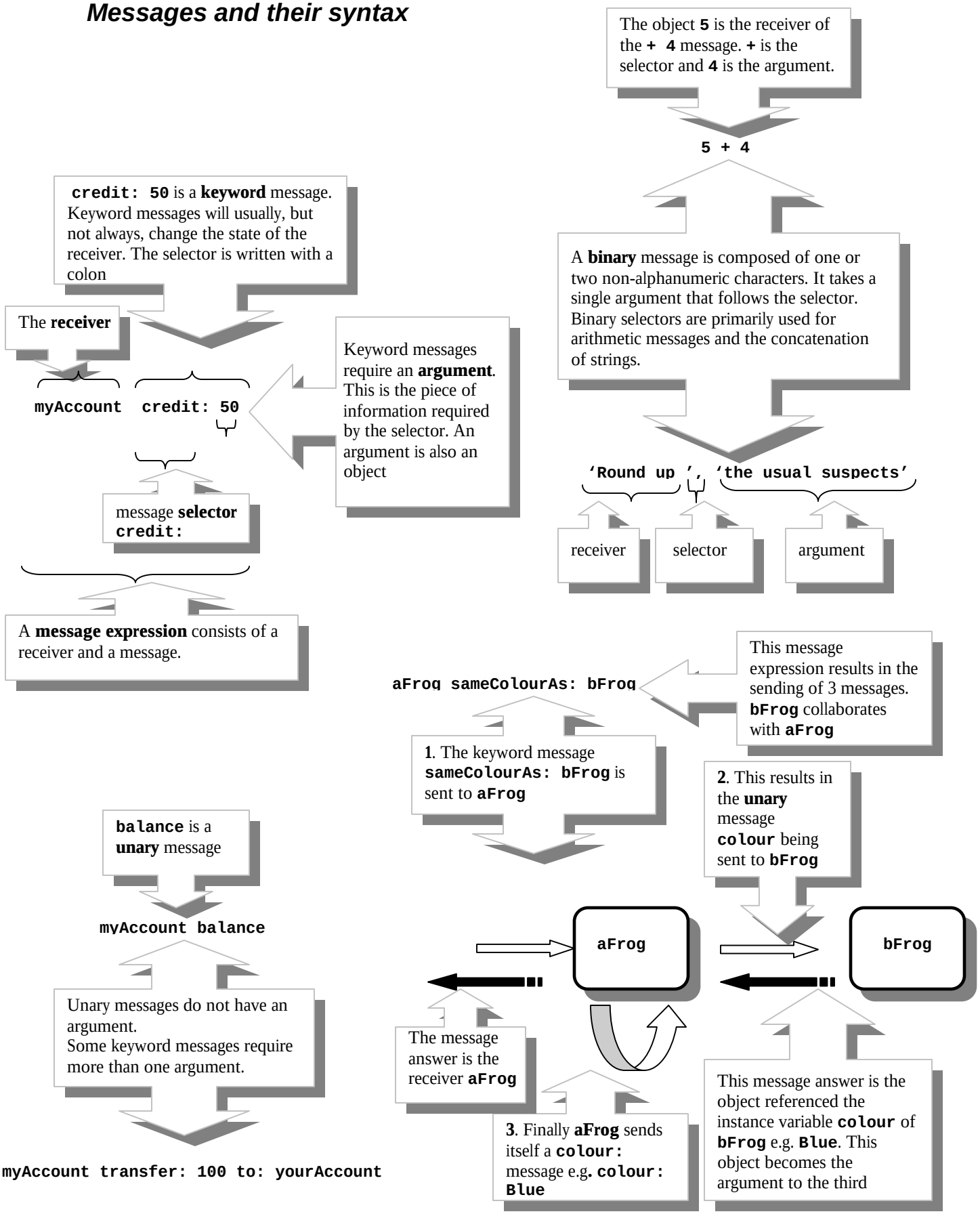
Software **models** some aspect of the real world.

Domain model (Ch5) The software objects and messages passing between them that makes up the program. These are separate from the user interface.

Each user interface is independent of the domain model. It can be changed without affecting the underlying objects and messages. An interface is the means by which humans interact with objects.

Employs widgets (buttons, drop-down boxes, text-fields etc), and graphics.

Messages and their syntax



Precedence

Smalltalk precedence rules (Ch 14 p12)

1. **Bracketed** expressions are evaluated first**, work from left to right.**
2. **unary** expressions are evaluated **left to right before (take precedence over) binary** expressions;
3. **binary** expressions are **evaluated left to right before (take precedence over) keyword** expressions;
4. **keyword** expressions are evaluated **last**, again working **left to right**.
5. **Warning!** Although usually sufficient for most purposes, these rules occasionally lead us astray. See below.

`new` is a class message (unary) (Ch 22 p6). It is sent to the class **`Account`** to create an **`Account`** instance

3. The string **`'William Gates '`** receives the binary message **`, '3'`** and returns the string **`'William Gates  3'`**

```
Account new holder: 'William Gates ' , 3 printString
```

`printString` is a unary message that every object inherits from **`Object`**. It returns a string whose characters are a description of the receiver. (Ch 31 p26)

1. There are no brackets so **`Account new`** is evaluated first. It returns a new instance of **`Account`**.

4. The new **`Account`** instance receives the keyword message **`holder: 'William Gates 3'`** The message expression finally returns the receiver of this keyword message.

2. The object **`3`** receives the **`printString`** message and returns the string **`'3'`**

The precedence rules do not always accurately reflect the order of evaluation as implemented by the Smalltalk **parser**, the software that 'reads' the code. The parser works strictly left to right but before evaluating a message it looks ahead to the next message to check if it is of a higher precedence than the current message. If the next message is of higher precedence then it gets evaluated before the current message. The **precedence tool** (LB- 14 Practical 14) uses the same rules as the parser and works well as long as brackets are not used. It works reliably for evaluating expressions that are combinations of binary, unary and keywords but you should **avoid using it with bracketed expressions**.

```
aFrog right position:(aFrog position)
```

If you follow the precedence rules you will evaluate the bracketed message first. This will lead you to a wrong evaluation. The parser evaluates **`aFrog right`** first - it reads left to right – this is the first unary message. If **`aFrog`** is a newly created instance with **`position`** of **`1`** then the above expression will set its position to **`2`** not **`1`**

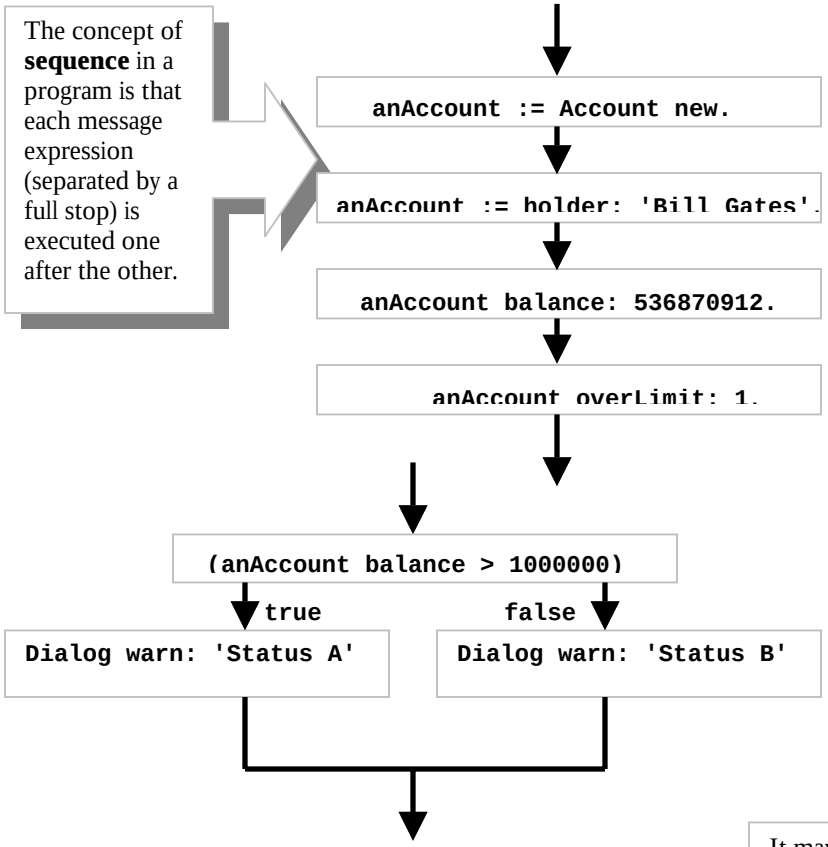
The precedence tool finds **`+`** realises it must look ahead one for a message of higher precedence. In the look ahead it instead finds the brackets and correctly evaluates the contents of those brackets. It then 'forgets' that it still hasn't completed the original search for a message of higher precedence and incorrectly evaluates the binary message **`(10 + -10)`** before sending **`negated`** to **`0`**. The tool leads you to an incorrect answer of **`0`**. The correct result of the message expression should of course be **`20`**.

```
10 + (10 negated) negated
```

Structuring Smalltalk

Branching

The concept of **sequence** in a program is that each message expression (separated by a full stop) is executed one after the other.



There are two mechanisms by which this linear progression through a message series can be changed. These are called **branching/selection** and **looping/iteration**.
By employing the concept of branching we can follow different paths through a message expression series. In branching a boolean **condition** (ie an expression which will return either the object **`true`** or **`false`**) determines which path is taken (see Ch.16 p3). If there are two possible paths, the Smalltalk message **`ifTrue: ifFalse:`** is appropriate.

E.g

```
(anAccount balance > 1000000)
   ifTrue: [Dialog warn: 'Status A']
   ifFalse: [Dialog warn: 'Status B']
```

It may be the case that the situation is not an "either or" situation but that a message expression(s) should or should not be evaluated depending on the Boolean condition. In these cases a simple **`ifTrue:`** or **`ifFalse:`** message is sufficient. E.g.

```
(self balance isNil)
    ifTrue: [self balance: 0].
```

Looping

The terms iteration and loop are interchangeable. There are two main parts to a loop construct. The loop body is the message expression, or expressions that are to be repeated. There must also be a way of controlling how many times the loop body is executed. Loops generally fall into two categories. In a count-controlled loop it is determined how many times the loop body will execute before the first iteration takes place.

```
myFrog := Frog new.

3 timesRepeat: [myFrog right]
```

In Smalltalk sending the **`timesRepeat:`** message results in a count-controlled loop. The receiver, an integer object, determines the fixed number of times that the loop body will execute. The loop body is contained in the argument which will be a Block (since it is code that will be repeatedly evaluated.) (see Ch.20 p10).
In this example the code inside the block is repeated 3 times.

Looping continued

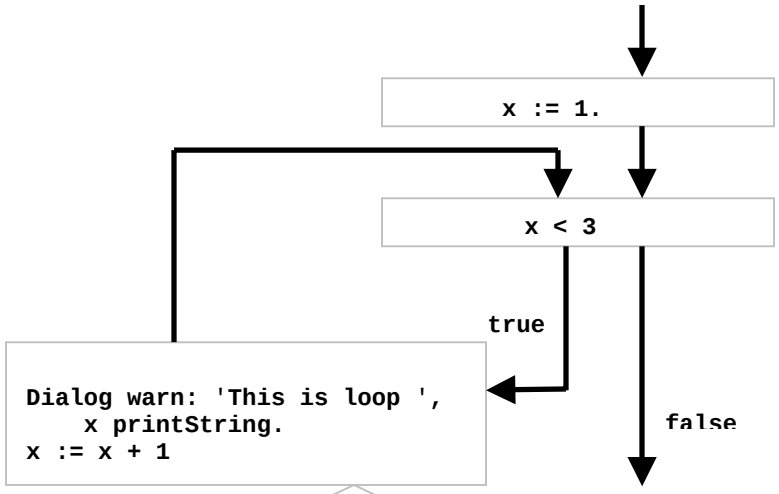
The second kind of loop is event-driven, whereby an event happens inside the loop that leads to the halting of execution. In Smalltalk the messages **`whileTrue:`** (Ch 20 p11) and **`whileFalse:`** (Ch 31 p19) cause the loop body inside the argument block to be repeatedly executed whilst the condition in the receiver block returns **`true`**, in the case of **`whileTrue:`** , or returns **`false`** in the case of **`whileFalse:`** .

```
|x|

x := 1.

[x < 3] whileTrue: [Dialog warn:
       'This is loop ', x
         printString.

 x := x + 1]
```



Notice that the receiver, a Block that contains a Boolean condition expression, is evaluated at least once to decide if the loop body is to be executed. At the end of each iteration it is evaluated again to see if another pass through the loop body is required. When using this kind of loop it is important to always ensure that within the loop body there is an expression, that will eventually change the boolean condition so that the loop body will stop executing. In the example above, the expression **`x:= x + 1`** will ensure that regardless of value to which **`x`** is initialised it will eventually be incremented to a value of 3, at which point the receiver will return false and the loop will exit. If you inadvertently write a loop where there is no provision made for the boolean condition to change, then you get an infinite loop, and your machine will appear to freeze as it repeats the loop body ad infinitum. Pressing Ctrl+Q together may solve this problem.

This will loop twice before the condition returns **`false`**. Once this happens the next message expression in the sequence after the **`whileTrue:`** message will be evaluated.

Nested Structures
In both branching and looping you may have 'nested' structures. That is loops and branches that are themselves within loop and branch bodies. In these cases, proper indentation is helpful in signposting the structure of the code.

```
(anAccount balance >= chequeAmount)
     ifTrue: [
               Dialog warn: 'Pay cheque']
     ifFalse: [
      ((anAccount overLimit + anAccount balance) >= chequeAmount)
            ifTrue:[
               Dialog warn: 'Overdraft needed']
            ifFalse:[
               Dialog warn: 'Bounce cheque']]
```