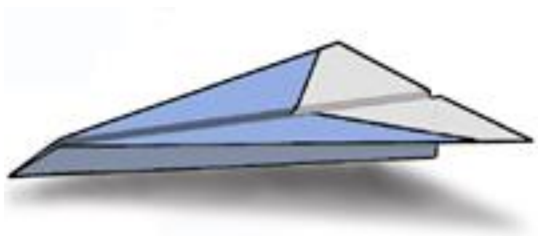
# Imperative Programming 3

## Objects

Peter Jeavons

Trinity Term 2019

# Why another programming course?



Toy programs are:

- small
- simple
- solitary
- short-lived

Real programs are:

- large
- complex
- designed by teams
- long-lasting

**Question:** What is the hardest thing about writing real software? Why?

# Outline (details on web)

- Fundamental principles of OOP
- Design patterns and case studies
- Designing out errors, test-driven development
- Generics and collections
- Graphics, events and threads
- Code organisation/software development

A function to delete a character
(from the Nano editor, written in C)

```
void do_delete(void)
{
    bool do_refresh = FALSE;
        /* Do we have to call edit_refresh(), or can we get away with
         * update_line()? */

    assert(current != NULL && current->data != NULL
                && current_x <= strlen(current->data));

    placewewant = xplustabs();

    if (current->data[current_x] != '\0') {
        int char_buf_len = parse_mbchar(current->data + current_x,
                NULL, NULL, NULL);
        size_t line_len = strlen(current->data + current_x);

        assert(current_x < strlen(current->data));

        /* Let's get dangerous. */
        charmove(&current->data[current_x],
                &current->data[current_x + char_buf_len],
                line_len - char_buf_len + 1);

        null_at(&current->data, current_x + line_len - char_buf_len);
        if (current_x < mark_beginx && mark_beginbuf == current)
            mark_beginx -= char_buf_len;
        totsize--;
    } else if (current != filebot && (current->next != filebot ||
        current->data[0] == '\0')) {
        /* We can delete the line before filebot only if it is blank: it
         * becomes the new magicline then. */
        filestruct *foo = current->next;

        assert(current_x == strlen(current->data));

        /* If we're deleting at the end of a line, we need to call
         * edit_refresh(). */
        if (current->data[current_x] == '\0')
            do_refresh = TRUE;

        current->data = charealloc(current->data,
                current_x + strlen(foo->data) + 1);
        strcpy(current->data + current_x, foo->data);
        if (mark_beginbuf == current->next) {
            mark_beginx += current_x;
            mark_beginbuf = current;
        }
        if (filebot == foo)
            filebot = current;

        unlink_node(foo);
        delete_node(foo);
        renumber(current);
        totlines--;
        totsize--;
        wrap_reset();
    } else
        return;

    set_modified();

    if (do_refresh)
        edit_refresh();
    else
        update_line(current, current_x);
}
```

# Recap: Abstract Data Types

- Part 2 introduced the idea of simplifying programs by using **abstract data types (ADTs)** to store data and provide operations on it.
- Each ADT has a specification that describes what its operations do, and an implementation that provides code to do it.
- To use an ADT we only need to know the specification; different implementations can be used interchangeably.

# Objects

- In Part 3 we extend this idea to build programs out of **objects** that store data and provide operations on it.
- Each object has a specification that describes what its operations do, and an implementation that provides code to do it.
- To use an object we only need to know the specification or public interface; different implementations can be used interchangeably.

# Classes

- Most objects are instances of **classes** that define the data and operations for a whole family of similar objects.
- The power of OOP comes from:
  - the ability to use many powerful and general *pre-defined classes*
  - the ability to *define new classes*…

# Defining a new class - example

- If our program manipulates text, we might want to define a `Text` class, which would allow us to create and manipulate objects representing a sequence of characters.
- The interface might specify methods to: get the length of a text, read the character at a given position, and insert a character at a given position

# Using the new class

- Without knowing anything about how the new class is *implemented*, we can already write code that uses it:

```
buffer = new Text
for (i <- 1 to buffer.length())
  if (buffer.charAt(i-1) == ’a’)
              buffer.insert(i, ’b’)
```

- This is one way that OOP achieves:

# Note the benefits…

- The example code deals only with the logic of the task (*not* with how the data is stored), so it's easier to *write*.
- It's easier to *understand* what it does
- It's easier to *modify* when necessary (for example, to also delete all the "c"s)
- The *same code* can deal with different kinds of text (for example, text that was too big to all be stored in memory), so it's easier to *reuse*.

# Providing a new class

- To gain all the benefits of using our new class we have to be able to *implement* it (if it isn't pre-defined in our language);
- This might be done by a different person;
- This way of breaking a large programming task into separate pieces is one way that OOP achieves:

# Big Idea 1:

Object Identity

# Object identity

- Multiple instances of a class have separate *identities*: updates to temperature in London should not change Paris


- Object identities can be copied without making a copy of the object: this is known as *aliasing* or *sharing*

# Example (object identity)

```
var a = new ArrayIntSet(100) // a is one object
var b = new ArrayIntSet(100) // b is separate object
        new constructs the object and returns its identity

a.insert(23)
a.contains(23) /* --> true  */
b.contains(23) /* --> false */

b = a     // a is now aliased to b

// a & b share identity. (Original b is lost)
a.insert(34)
b.contains(34) /* --> true */
```

# Big Idea 2:

Encapsulation

# Encapsulation

Separation between *external* interface of component and its *internal* implementation

“Separation of concerns”

- Methods not in the interface should be *hidden* by marking them as “private”
- Instance data also (usually) private

# Example

```
class ArrayIntSet(MAX: Int) {
  private var elems = new Array[Int](MAX)
  private var size = 0

  def isEmpty = (size == 0)
  def contains(x: Int) = { ... }
  private def find(x: Int): Int = { ... }
}
```

- All data is private (hidden & inaccessible)...
- ...so the caller can’t break *concrete invariant*
- Helper method is private (not part of interface)

# Encapsulation is good

- **user** and **implementer** only need to understand the interface and its specification (abstraction and decomposition)
- Debugging (“Who broke their promise?”)
- Program *evolution*:
  - Re-implement with confidence that behaviour doesn’t change
  - Enhance components by extending the specification (which informs what else needs to change)

# Guidelines for encapsulation

OOP buzzwords and fashionable methodologies build on the central usefulness of encapsulation

Examples:

- "DESIGN BY CONTRACT"
- "*Loose coupling*" and "*High cohesion*"
- "**Law of Demeter**" aka "**Principle of least knowledge**"

# Loose coupling & high cohesion

Loose coupling means that different classes should not depend closely on the details of each other

- Can *understand* one class without reading the other
- Can *change* one class without affecting the other
- Improves maintainability

# Loose coupling & high cohesion

High cohesion means that the variables and methods within one class are closely associated (ideally *one task* per method)

- Can easily *understand* what class/methods do
- Can use simple *descriptive names*
- Can easily *reuse* classes and methods

# Law of Demeter

A specific way to encourage loose coupling…

“Principle of least knowledge”

“Don’t talk to strangers”

“USE ONLY ONE DOT”

# Breaking the Law of Demeter

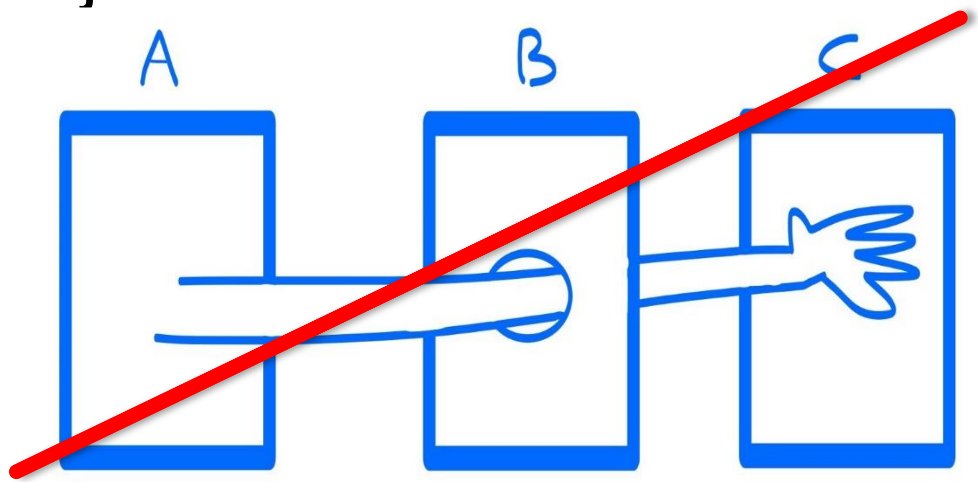
- You don’t control your dog’s legs—you just take the dog for walk
- You don’t ask for money by taking control of wallets.  Customers keep control

```
payment = 2.00;
paidAmount = myCustomer.getPayment(payment);
if (paidAmount == payment) {
    // Thank you!
}
```

# Enforcing the Law of Demeter

A method of an object should invoke only the following kinds of objects:

- The object itself
- The method's parameters
- Objects the method creates/instantiates
- Direct component objects



# Big Idea 3:

Polymorphism

# Polymorphism

- Literally means “many shapes”
- The idea is that some constructs in a programming language can process *objects of different data types* in appropriate ways

# Polymorphism

- For example, the same operator or method can be explicitly defined for several different argument datatypes

  (“overloading” or “ad hoc polymorphism”)

- Some code can be written *generically* so that it can handle argument values *identically* without depending on their type

  (“parametric polymorphism” or “generics”)

C. Strachey, *Fundamental concepts in programming languages*. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967

# Polymorphism

- A key form of polymorphism in OOP is that multiple components can implement the *same interface* and can therefore be used interchangeably

"Program to the interface"

- Using only the interface leads to loose coupling – no specific details of class used

"Stability under change"

# Example (polymorphism)

```
/** IntSet is an interface */
trait IntSet {
  def isEmpty: Boolean
  def contains(x: Int): Boolean
  def insert(x: Int)
  def delete(x: Int)
}
```

```
/* Different implementations extend it */
class ArrayIntSet(MAX: Int) extends IntSet
{..}
class BitmapIntSet(MAX: Int) extends IntSet
{..}
```

```
// Use it
val a: IntSet = new BitmapIntSet(100)
```

# Example (polymorphism)

```
// Use it
val a: IntSet = new BitmapIntSet(100)
```

- A constant value, `a`, of static type `IntSet` has been initialised with a newly constructed object from a concrete class
- Code can invoke any methods from the `IntSet` interface on the object a (but can’t use any additional features provided by `BitmapIntSet`)

# Runtime polymorphism

```
/** This method makes an IntSet. The
 * concrete type is chosen at run time
 */
def makeSet(bound:Int): IntSet = {
   if (bound <= 1000)
      new BitmapIntSet(bound)
   else
      new ArrayIntSet(100)
```

Note: polymorphism is also useful for debugging.

(For example, temporarily replace the class responsible for updating live data with one which does not.)

# We’ve not covered inheritance

- It’s not strictly necessary: we can write programs in OOP style *without inheritance*
- Inheritance relationships may not be *stable under change,* so need careful thought
- Interfaces between a class and its subclasses may be hard to specify

More in the next lecture…

# Summary

- Some slogans and three big ideas from object-oriented programming:
  - Object identity
  - Encapsulation
  - Polymorphism…
  - …but not inheritance
- General principles for good design
  - Loose coupling
  - High cohesion

See also *Programming in Scala*: Chapter 4