Imperative Languages

Java is an *imperative* object-oriented language.

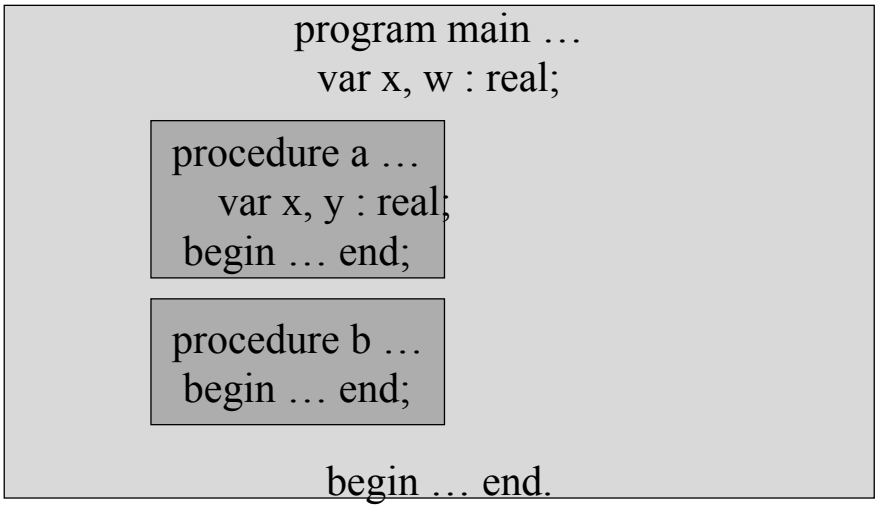
What is the difference in the organisation of a program in a ***procedural*** and an ***object-oriented*** language?


```
class BankAccount {
  private int balance;
  private String accNum;
  public BankAccount(String a) {
    accNum = a;
    balance = 0;
  } //constructor
  public void deposit(int amount) {
    balance = balance + amount;
  } // deposit
  public int getBalance() {
    return balance;
  } // getBalance
} // BankAccount
```

```
class BankClient {
   private BankAccount bk1
        = new BankAccount("a123");
   ...
   public void cutDebt(int m) {
        int am1 = bk1.getBalance();
        if (am1 < 0)
            bk1.deposit(m);
   } // cutDebt
} // BankClient
```


Procedural language

In a procedural language such as Pascal, the main construct is the procedure or function.



We have a ***main program*** within which ***types***, ***variables***, ***procedures*** and ***functions*** are defined.

BankAccount in Pascal

```
program BankAccountEx(input, output);
  type BankAccount =
    record
      balance: Integer;
      accNum: String;
    end;
  var bk1, bk2: BankAccount;
      am1: Integer;
  procedure mkBankAccount
   (var b: BankAccount; a: String);
  begin
    b.balance := 0;
    b.accNum := a;
  end {mkBankAccount};
```



```
  procedure deposit(
    var b: BankAccount; amount: Integer);
  begin
    b.balance := b.balance + amount;
  end {deposit};
  function getBalance(b:BankAccount):Integer;
  begin
    getBalance := b.balance;
  end {getBalance};
begin
  mkBankAccount(bk1, "1234");
  ... deposit(bk1, 6);
  ... am1 := getBalance(bk1); ...
end.
```

Procedural language

Blocks

- Program or subprogram (procedure or function).
- Blocks can contain the definition of types, variables, procedures and functions.
- Can be nested ***(block-structured language)***

Procedural language: data

Procedural languages have records.

Pascal

```
type BankAccount =
        record
    balance: Integer;
     accNum: String;
        end;
```

Java

```
class BankAccount {
 public int balance;
public String accNum;
 } // BankAccount
```

Java and C++ classes are just records in which we can define operations as well as attributes.

In Pascal, operations are separate from types.

Method/Procedure calls

In Java, calls have the form:

```
bk1.deposit(m);
```

In Pascal, calls have the form:

```
deposit(bk1, m);
```

In Pascal, we require a parameter of type **`BankAccount`** to determine which **`BankAccount`** object is being referred to.

In Java, the **`bk1`** is the focus of our attention while in Pascal it is just another parameter.

In general, in Pascal, we have one more parameter in a procedure call than in the equivalent call of a Java method.

Problems with Pascal approach

There is no constructor in Pascal. We need an extra procedure **`mkBankAccount`** to initialise a **`BankAccount`** variable.

In a large Pascal program, the definitions of **`mkBankAccount`**, **`deposit`** and **`getBalance`** can be far away from the definition of type **`BankAccount`**.

This makes it difficult for the reader to recognise that they logically belong together.

Problems with Pascal approach

In Pascal, the fields of a record are visible (public) and can be accessed as in:

```
bk1.balance
bk1.accNum
```

As the structure of **`BankAccount`** is visible, it is possible to directly access and change **`balance`** and **`accNum`**.

Only when the implementation of a type is ***hidden*** (private) can we ***guarantee*** that it is only manipulated through public methods and that users cannot take “efficient” short cuts by directly manipulating internal details.

Modules

Modules :

- have a visible interface part,
- have a hidden implementation part,
- allow types, variables and subprograms to be defined together as a group,
- should allow full type checking across module boundaries,
- allow libraries of pre-compiled modules to be built up.

Definitions

Data abstraction: we think of a type as a whole and in terms of what we can do with it through its public operations.

Encapsulation: attributes and methods declared together in a single unit.

Information hiding: the internal details of a type are hidden.

Encapsulation

Encapsulation is provided by Ada 83:

```
package BankAccounts is
  type BankAccount is private;
  procedure mkBankAccount(b: out BankAccount;
                a: in Integer);
  procedure deposit(b: in out BankAccount;
                    amount: in Integer);
  function getBalance(b: BankAccount)
                return Integer;
private
  type BankAccount is
    record
      balance: Integer;
      accNum: Integer;
    end record;
end BankAccounts;
```

Encapsulation

We also need a ***package body***.

```
package body BankAccounts is
   -- definitions of mkBankAccount,
   -- deposit and getBalance
end BankAccounts;
```

The signatures of the operations given in the package specification and are visible while their bodies are hidden in the package body.

`BankAccount` is a ***private type***; its details are given in the ***private part*** of the package specification.

Information Hiding

We therefore get information hiding.

The identifier **`BankAccount`** is visible, but no information about the structure of **`BankAccount`** is available to users of the package.

A **`BankAccount`** object can only be accessed or modified through the operations **`mkBankAccount`**, **`deposit`** and **`getBalance`** (plus equality and assignment).

Encapsulation: Ada

A package allows us to group together the definition of a type together with the operations that operate on the type.

However, we declare variables of type **`BankAccount`** and make calls of **`mkBankAccount`**, **`deposit`** and **`getBalance`** in the same way as in Pascal because these operations are not declared as part of type **`BankAccount`**.

In Java a class ***is*** a type definition while an Ada package can ***contain*** a type definition.

Ada is an ***object-based language***.

A hybrid language: C++

C++ has both function and class definitions.

```
class Cost {
private:
  int cents, dollars;
public:
  Cost(int d, int c) {
    dollars = d;
    cents = c;
  } //constructor
  void add(int d, int c) {
    dollars += d;
    cents += c;
  } //add
  int getDollars() const {return dollars;}
  int getCents() const {return cents;}
};
```

C++

```
void main() {
  Cost dress(45, 95);
  Cost* book = new Cost(15, 50);
  ...
  dress.add(5, 0);
  book->add(3, 15);
  ...
}
```

`main` is a void function.

It corresponds to the main program, i.e. the place where program execution starts.

Does C++ have

encapsulation?

information hiding?

C++

In C++, methods are called **`member functions`**.

There are also ordinary functions, as in a procedural language.

The syntax of a C++ function definition is the same as a C++ method.

C++ was developed from C. C has **`structs`**, same as Pascal or Ada records.

A C++ class is a **`struct`** which has methods as well as attributes.

Java equivalent

```
class Cost {
  private int cents, dollars;
  public Cost(int d, int c){
    dollars = d;
    cents = c;
  } //constructor
  public void add(int d, int c) {
    dollars += d;
    cents += c;
  } //add
  public int getDollars() {return dollars;}
  public int getCents() {return cents;}
} // Cost
```



```
public class Example {
    private Cost dress = new Cost(45, 95);
    private Cost book = new Cost(15, 50);

    public static void main(String [] args) {
        Example ex = new Example();
    } // main

    public Example() {
        ...
        dress.add(5, 0);
        book.add(3, 15);
        ...
    } // constructor
    ...
} // Example
```

Side-effects

We will use the term ***procedure*** as a generic term to cover C and C++ **`void`** functions, as well as Pascal procedures.

We will use the term ***method*** as a generic term to cover C++ member functions, as well as Java methods.

In procedures, functions and methods, non-local variables may be accessed and may have their values changed.

This is known as a ***side-effect***.

Side-effects

Ideally, in a procedural language, procedures should be self-contained entities.

Reasoning about a procedure is much easier if there are no side-effects, i.e. non-local variables are not modified.

In an object-oriented language, the purpose of a **`void`** method is usually to modify one or more attributes.

Hence, we have a side-effect.

Not a problem as the side-effects are restricted to being within the object.

Side-effects

Often, in a procedural language like Pascal, the effect of calling a procedure is to modify the value of one of its parameters.

An example is the **`deposit`** procedure in our **`BankAccount`** example which has a **`var`** parameter.

In an object-oriented language like Java, the effect of calling a **`void`** method is to modify the value of one of the object's attributes.

An example is the **`deposit`** method in our **`BankAccount`** example.

Pure Functions

Do these two expressions give the same result?

```
f(x) + g(y)
g(y) + f(x)
```

Avoiding side-effects is very important for functions and for value returning methods.

C makes heavy use of side-effects in functions: this can lead to very tricky programming.

A ***pure function*** returns a value and *does nothing else*.