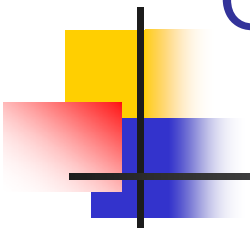
Calcul en précision arbitraire
Séquence pédagogique

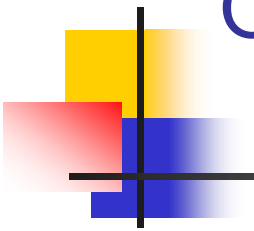
Luc Bougé
Professeur ENS Cachan
Antenne de Bretagne

Objectifs: aspects mathématiques



- Reformuler les connaissances et techniques en arithmétique
  - Opérations élémentaires
  - Procédures de calcul
  - Calcul numérique de tête
- Réfléchir à la notion de calcul
  - Machine de Turing

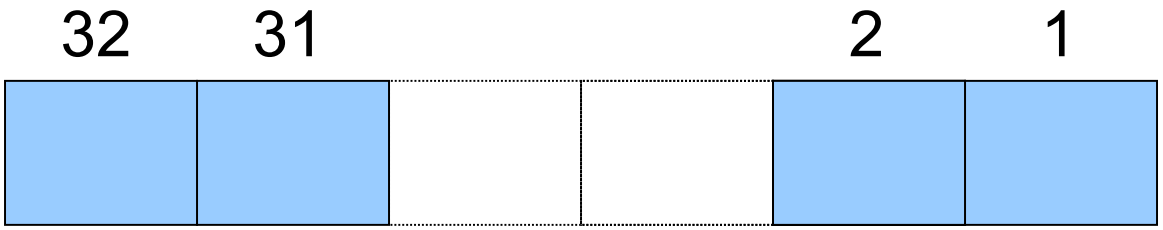
Objectifs: aspects informatiques



- Mettre en place la notion d'interface
  - *Application Programming Interface*
  - Notion d'objet et de méthode
  - Conception hiérarchique
- Itérations bornées
  - Boucle *for*
- Structure de donnée abstraite
  - Notion de *vecteur*

Le problème

- Limites de calcul des machines
  - En fonction de la base B?
  - Cas B=2
    - Combien font 2^{32}? 2^{64}? 2^{100}?



Combien peut-on écrire de nombres en base B?

Comment coder les nombres signés?

- Codage par complément à B
  - Cas B = 2
  - Cas B = 10
- Cas B = 2^{32}

3 = 011

.....

1 = 001

0 = 000

? = ???

.....

-4 = 100

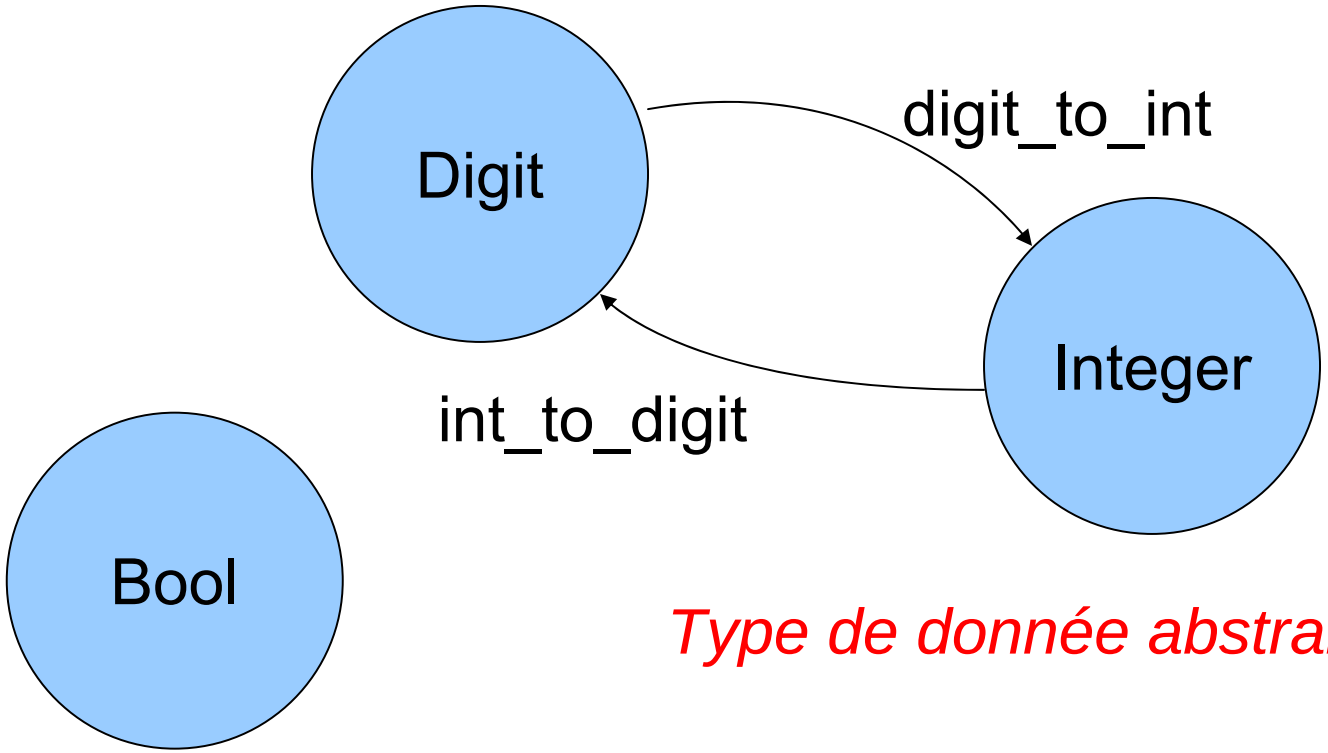
Bit de signe!

Et les réels?

- Codage IEEE 754 64 bits
  - 1 bit de signe
  - 52 bits de mantisse non signée
  - 11 bit d'exposant signé
- Plus grand nombre positif? $2^{52} \times 2^{1024} = 10^{325}$
- Plus petit nombre positif? $2^{-1024} = 10^{-300}$
- Précision maximale? $2^{52} = 10^{18}$
  - Environ 18 chiffres après la virgule

Qu'est-ce qu'un chiffre?

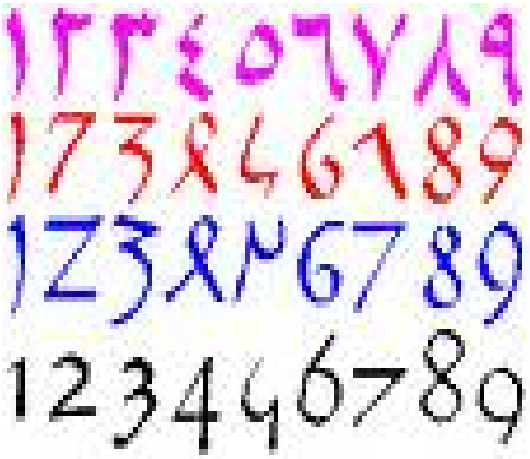
Equality: Digit x Digit -> Bool
Base: Unit -> Int
Print: Digit -> Unit

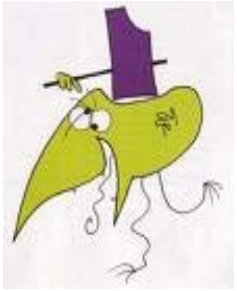


Type de donnée abstrait

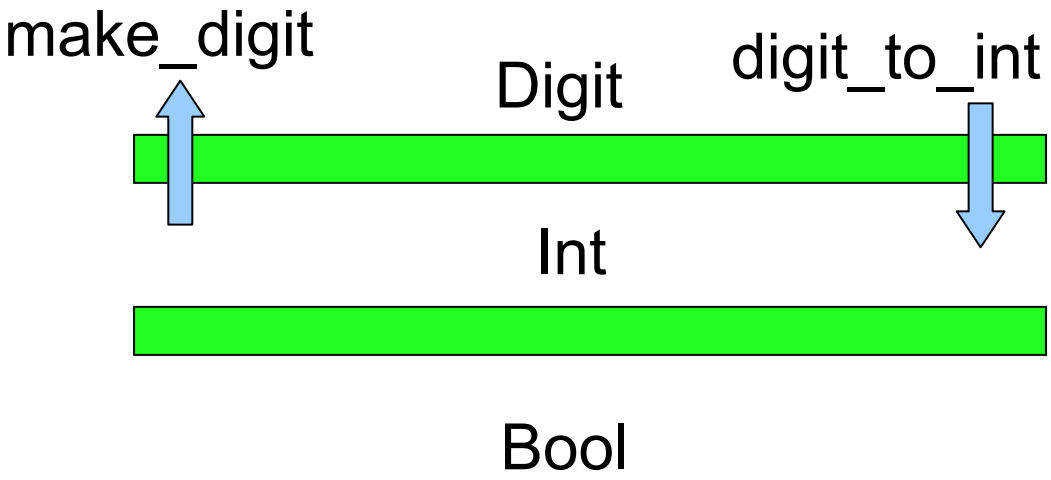
Qu'est-ce qu'un chiffre? (2)

- Entiers
  - 0, 1, 2, 3, 4
- Chiffres romains
  - "", "I", "II", "III", "IV"
- Chiffres anglais
  - "zero", "one", two", "three", "four"
- Chiffres Shadoks
  - 



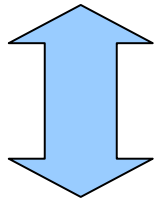


Qu'est-ce qu'un chiffre? (3)



Application Programming Interface

Programme
Application
Abstrait
Haut niveau



Système
Implémentation
Concret
Bas niveau

Qu'est-ce qu'un chiffre? (4)

```
#define BASE 10

class Digit {
private:
  int contents;
public:
  Digit(void);
  explicit Digit(int n);
  int to_integer() const;

  friend ostream& operator <<(ostream& s, Digit a);
};

extern bool operator ==(Digit a, Digit b);
extern bool operator !=(Digit a, Digit b);
```

Qu'est-ce qu'un chiffre? (5)

```
Digit::Digit(void): contents(0) {};

Digit::Digit(int n): contents(n) {
  assert((0 <= n) && (n < BASE));
}

int Digit::to_integer() const {
  return contents;
}

bool operator ==(Digit a, Digit b) {
  return a.to_integer() ==
b.to_integer();
}
```

Assertion

Fonction partielle: Integer -> Digit

Maintenant, les nombres naturels!

- Un nombre est un tableau de chiffres de taille fixée N
  - Cas N = 32, N = 64, N = 1000

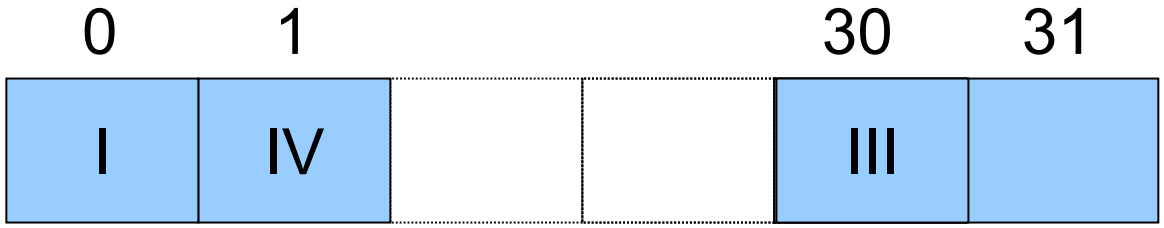
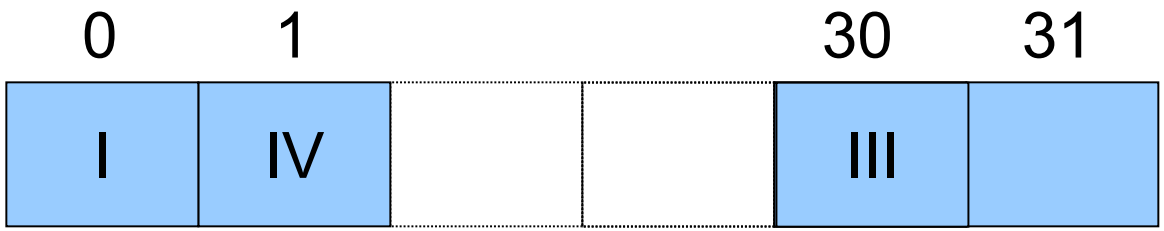


Tableau comme type abstrait

- Accéder à l'élément à un index
  - a = get_digit(i), set_digit(i, a)
- Parcourir dans le sens direct des indices
  - for (i = 0; i < N; i = i+1)
- Parcourir dans le sens inverse des indices
  - for (i = N-1; i >= 0; i = i-1)

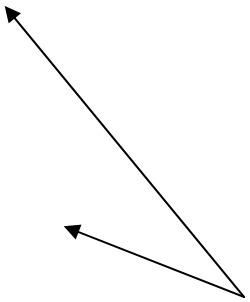


Qu'est-ce qu'un nombre?

```
#define N 1024

class Number {
private:
  Digit contents[N];

public:
  Number(void);
  explicit Number(Digit a);

  void set_digit(int i, Digit a);
  Digit get_digit(int i);

  friend ostream& operator <<(ostream& s, Number x);
};
```

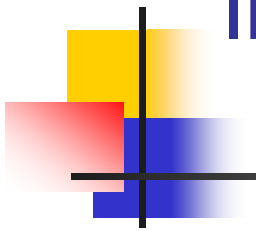
Qu'est-ce qu'un nombre? (2)

```
Number::Number(void) {
  for (int i = 0; i < N; i++)
    contents[i] = Digit(0);
}

Number::Number(Digit a) {
  for (int i = 0; i < N; i++) {
    contents[i] = Digit(0);
  }
  contents[0] = a;
}

void Number::set_digit(int i, Digit a) {
  assert((0 <= i) && (i < N));
  contents[i] = a;
}
Digit Number::get_digit(int i) {
  assert((0 <= i) && (i < N));
  return contents[i];
}
```



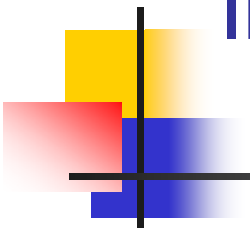
Itérateurs

Imprimer un nombre



- Tableau: 0012345000000
  - -> 5432100
- Itérer dans le sens direct (i)
  - Impression du chiffre courant

Imprimer un nombre



- Tableau: 0012345000000
  - -> 5432100
- Itérer dans le sens direct (i)
  - Impression du chiffre courant
- ***Sauf pour 0000000000!***

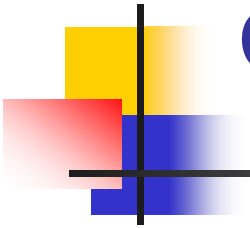
Imprimer un nombre (2)

- first_non_zero := NONE
- Itérer dans le sens inverse (i)
  - Si l'élément à l'index i est non nul
    - Alors first_non_zero := i; **sortir**
- Si first_non_zero = NONE
  - Alors imprimer le chiffre "0"
  - Sinon, continuer l'itération en imprimant les chiffres

Échappement

Imprimer un nombre (3)

```
int first_non_zero = NONE;

for (int i = N-1; i >= 0; i--) {
    if (x.get_digit(i) != Digit(0)) {
      first_non_zero = i;
      break;
    }
  }

  if (first_non_zero == NONE)
    return s << Digit(0);

  for (int i = first_non_zero; i >= 0; i--)
    s << x.get_digit(i);
```



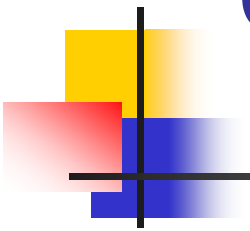
Comparer deux nombres: égalité

- Deux nombres x et y: sont-ils égaux?
- Itérer dans le sens direct (i)
  - si x[i] != y[i] alors rendre faux
- rendre vrai

Comparer deux nombres (2)

```
bool operator ==(Number x, Number y) {
  for (int i = 0; i < N; i++) {
    if (x.get_digit(i) ! = y.get_digit(i))
      return false;
  }
  return true;
}
```

Comparer deux nombres: inférieur



- Nombres x et y: x < y?
- Itérer dans le sens inverse (i)
  - si y[i] > x[i] alors rendre vrai
  - si y[i] < x[i] alors rendre faux
- Rendre faux

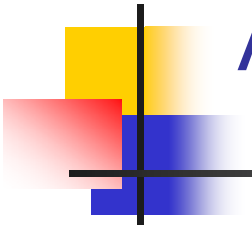
- x >= y
  - !(x < y)
- x > y
  - (x >= y) && (x != y)

Additionner deux nombres

Retenue

```
   1
  1234
+ 5678
------
  6912
```

Additionner deux nombres (2)



- Nombres x et y: calculer z := x+y
- carry := 0
- Itérer dans le sens direct (i)
  - xd := digit_to_int(x[i])
  - yd := digit_to_int(y[i])
  - s := carry + xd + yd
  - carry := 0
  - si s >= BASE
    - alors s := s – BASE; carry := 1
  - z[i] := int_to_digit(s)

Additionner deux nombres (3)

```
int carry = 0;

for (int i = 0; i < N; i++) {
  int xd = (x.get_digit(i)).to_integer();
  int yd = (y.get_digit(i)).to_integer();

  int s = carry + xd + yd;
  carry = 0;
  if (s >= BASE) {
    carry = 1;
    s = s - BASE;
  }
  z.set_digit(i, Digit(s));
}
assert(carry == 0);
```

Assertion

Multiplier deux nombres

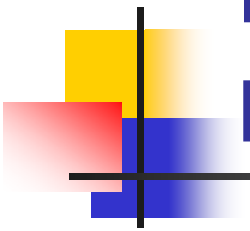
```
   1234
 x   45
------
   6170
+ 4936
------
  55530
```

Décalage

```
1234
x   5
```

```
1234
x   4
```

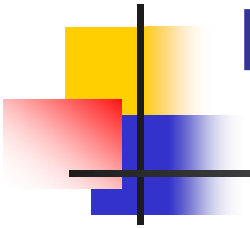
Multiplier un nombre par un entier-chiffre



- Nombre x et entier n: calculer y := x*n
- Assert 0 <= n < B
- carry := 0
- itérer dans le sens direct (i)
  - xd := digit_to_int(x[i])
  - p = **xd * n** + carry
  - quotient := **p / BASE** (division entière)
  - remainder := p % BASE (modulo)
  - y[i] := int_to_digit(remainder)
  - carry := quotient
- assert (carry = 0)

Multiplier un nombre par un entier-chiffre (2)

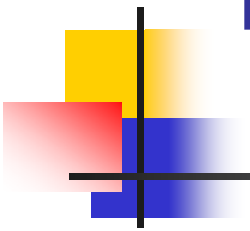
```
assert((0 <= n) && (n < BASE));
int carry = 0;

for (int i = 0; i < N; i++) {
  int xd = (x.get_digit(i)).to_integer();
  int p = (xd * n) + carry;

  int quotient = p / BASE;
  int remainder = p % BASE;

  assert((0 <= remainder) && (remainder < BASE));
  assert(p == (quotient * BASE) + remainder);

  y.set_digit(i, Digit(remainder));
  carry = quotient;
}
assert(carry == 0);
```



Décaler un nombre

- nombre x: décaler x d'un chiffre vers la gauche
- carry = int_to_digit(0)
- Itérer dans le sens direct (i)
  - tmp := x[i]
  - x[i] := carry
  - carry := tmp
- assert (carry = int_to_digit(0))

Multiplier deux nombres (2)



- Nombres x et y: calculer x*y
- result = le nombre 0
- Itérer dans le sens direct
  - n := digit_to_number(y[i])
  - z := x * n
  - shift_left(z, i) (décaler de i positions le résultat)
  - result := result + z

- Debordements possibles
  - multiplication par un chiffre
  - décalage
  - accumulation

Soustraire deux nombres

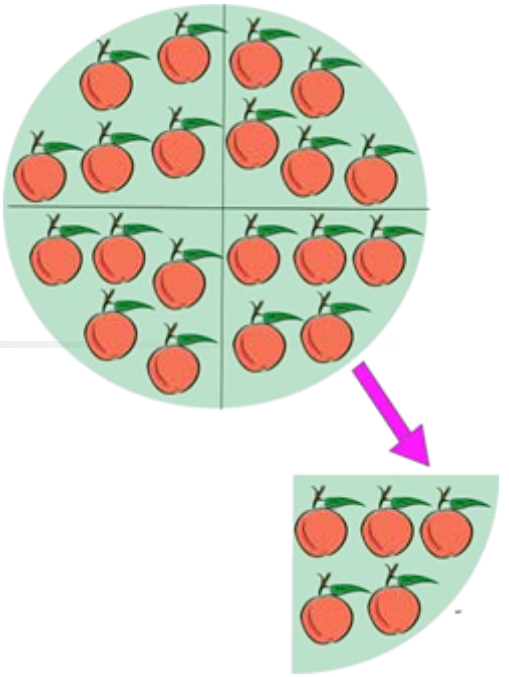
- Nombres x et y: calculer z: = x-y
- carry := 0
- Itérer dans le sens direct (i)
  - xd := digit_to_number(x[i])
  - yd := digit_to_number(y[i])
  - yd := yd + carry
  - si yd > xd
    - alors xd := xd + BASE; carry = 1
  - assert (xd >= yd)
  - z[i] := number_to_digit(xd – yd)
- assert (carry = 0)

Soustraire deux nombres (2)

```
int carry = 0;

for (int i = 0; i < N; i++) {
  int xd = (x.get_digit(i)).to_integer();
  int yd = (y.get_digit(i)).to_integer();

  yd = yd + carry;
  carry = 0;

  if (xd < yd) {
    xd = xd + BASE;
    carry = 1;
  }

  assert(xd >= yd);
  z.set_digit(i, Digit(xd - yd));
}
assert(carry == 0);

assert(x == y + z);
```

Programme auto-validant

Diviser deux nombres



- Nombres x et y: calculer z := x/y
- assert (y != zero)
- dividend := zero
- quotient := zero
- Itérer dans le sens inverse (i)
  - dividend := dividend*base + digit_to_number(x[i])
  - assert (dividend < y*base)
  - Trouver le plus grand entier n tel que y*n <= dividende
  - assert (0 <= n < BASE)
  - **assert(0 <= dividend – y*n < y)**

Diviser deux nombres (2)

- d := digit_to_number(int_to_digit(n))
- quotient := quotient * base + d
- dividend := dividend – (y * d)
- assert (0 <= dividend < y)

- remainder := dividend
- assert (0 <= remainder < y)
- assert (x = (quotient * y) + remainder)

Trouver le chiffre suivant du quotient

- Nombres x et y tel que x < y * BASE: trouver le plus grand n tel que y * n <= x
- Itérer dans le sens direct i := 0..BASE-1
  - assert (y*i <= x)
  - si **x – y*i < y** alors rendre i
- assert (faux)

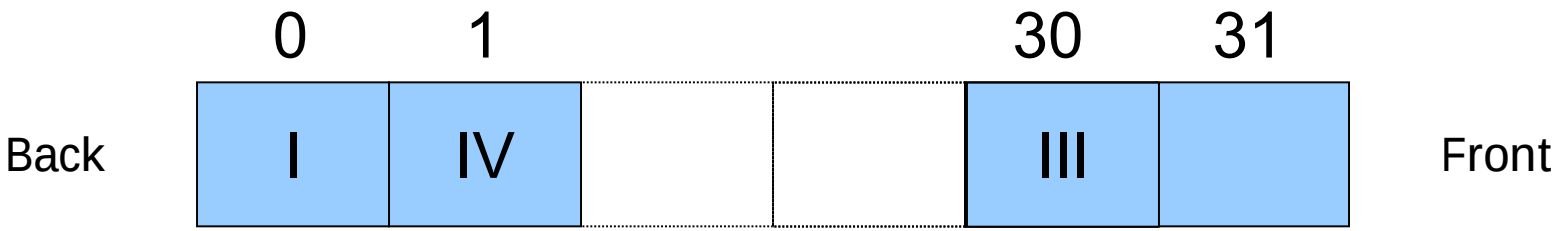
Ordre de grandeur de la base

Attention aux débordements!

Programmation défensive

Notion de "vecteur"

- Interface abstraite de tableau sans limite de taille
  - a = get(i), set(a, i)
  - a = pop_front(), push_front(a)
  - a = pop_back(), push_back(a)
  - etc.
- Gestion de l'allocation dynamique (**new**)



Conclusion: principes

- Méthodologie de la programmation
  - Algorithme + structure de données
    - Définir les interfaces avant tout
  - Structure de contrôle: itération bornée
    - Utilisation des échappements
    - Se méfier des itérations non bornées!
- Types de données abstraits
  - Interfaces bien définie
  - Test de cohérence à chaque appel

Conclusion: méthodologie

- Stratégie défensive
  - Utilisation intensive des assertions
  - Code auto-vérifiant
- Construction incrémentale
  - *Bottom-up*
  - *Top-down*
  - Utilisation cruciale de la notion d'interface
- Choix du langage
  - Notion d'interface abstraite
  - Itération + échappements