Object-oriented programming

Teréz A. Várkonyi

http://people.inf.elte.hu/treszka

Procedural vs. Object-oriented programming

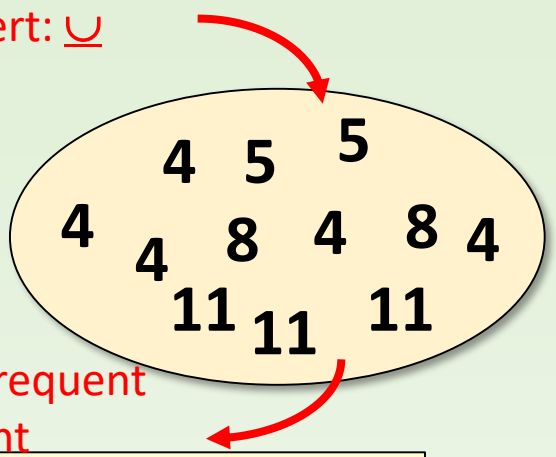
- Procedural programming: the solution is structured into independent units (subroutines, macros, procedures, functions). The execution is set by the control transfers (calls in case of procedures and functions) between these units.
- Object-oriented program: parts of the data needed for the solution and the related activities (called methods) are structured into independent units (called objects). The execution is set by the control transfers (calls or messages) between the methods of the objects.

Task

In a series there are natural numbers between 0 and m. Which element is the most frequent?

- There are several ways to solve it, e.g.
  - Procedural solution: *counting* embedded in *maximum search*,
    - Let's find the maximum occurrence of the numbers between 0 and *m*,
    - Let's find the maximum occurrence of the elements of the series.
  - Object-oriented solution: we place the elements into a container (collection) where insertion and maximum searching are quick.
    - Bag containing numbers between 0 and *m* (*set* with multiplicity),
    - Bag containing any natural number

Analysis and planning



variables of the task and of the program at the same time

A : m:$\mathbb{N}$, x:$\mathbb{N}^*$, elem:$\mathbb{N}$

m's initial value: m_0
x's initial value: x_0

x is not empty,
x's elements are between 0 and m

Pre : $m = m_0 \wedge x = x_0 \wedge |x| \geq 1 \wedge \forall i \in [1 .. |x|]: x[i] \in [0 .. m]$

initial value of the variables

$Post$: $Pre \wedge$ b:Bag $\wedge$ $b = \bigcup_{i=1}^{|x|} \{x[i]\} \wedge elem = frequent(b)$

final value of the variables

executive specification

Pre means that the inputs keep their initial values

Summation:		~	bagging
indexes:	i∈[m .. n]	~	i∈[1 .. |x|]
values:	f(i)	~	{x[i]}
result:	s	~	b
operator:	H, +, 0	~	Bag, ∪, ∅

b := ∅

i = 1 .. |x|

b := b∪{x[i]}

elem := frequent(b)

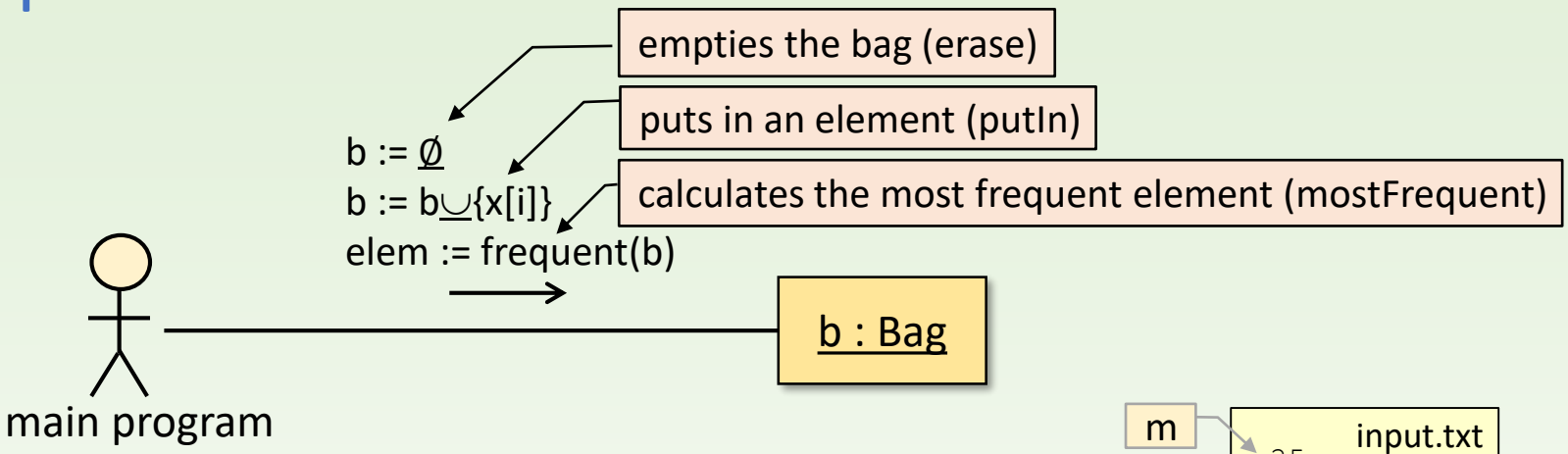
Testing strategies

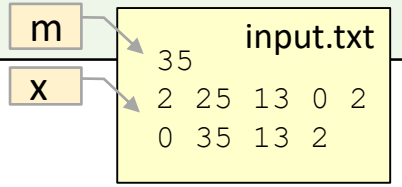
- Black box: test cases based on the specification.
  - test cases that violate the precondition
  - test cases for the postcondition
  - …
- White box: test cases based on the code.
  - cover every command
  - cover every conditional
  - …
- Grey box: test cases based on a predicted algorithm from the executive specification
  - If the specification refers to a famous algorithm then it is worthy to check the algorithm's typical test cases.

Test

Test cases of summation		**test data (after bagging series *x*, *we are looking for the most frequent element*)**		
interval [1 .. n]	length: 0	|x| = 0	x = < >	→ invalid
	length: 1	|x| = 1	x = < 2 >	→ e=2
	length: 4	|x| = 4	x = < 3, 5, 5, 1 >	→ e=5
	beginning	|x| = 2	x = < 1, 2 >	→ e=1
		|x| = 3	x = <1, 1, 2>	→ e=1
	end	|x| = 3	x = <1, 2, 2>	→ e=2
	middle	|x| = 4	x = <1, 2, 2, 3>	→ e=2
algorithm	load	|x| = 10000	x = < 2, 2, … , 2 >	→ e=2

Implementation



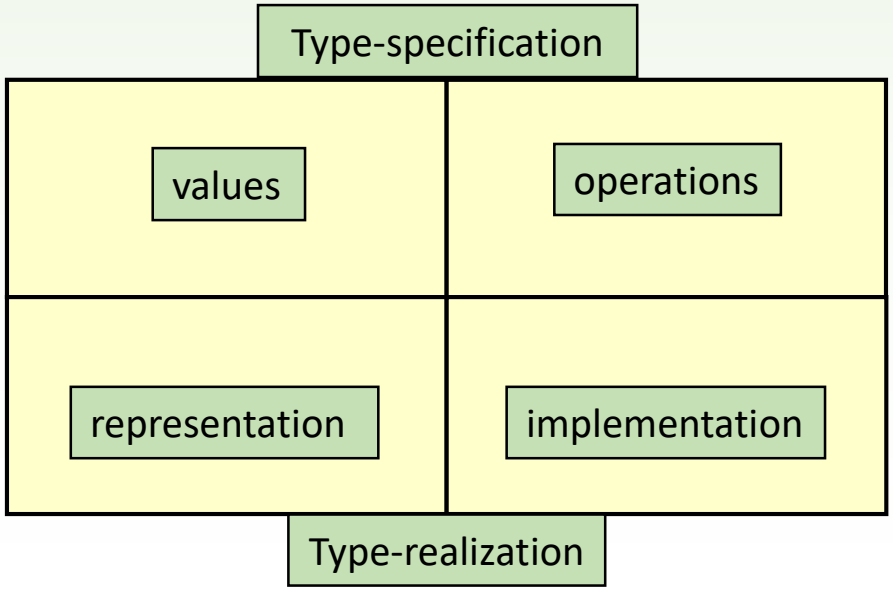



```
int main()
{
     ifstream f( "input.txt" );
     if(f.fail()){ cout << "Wrong file name!\n"; return 1;}

     int m; f >> m;

     Bag b(m);
     int e;
     b.erase();
     while(f >> e){ b.putIn(e); }

     cout << "The most frequent number: " << b.mostFrequent() << endl;
     return 0;
}
```

series *x* is in a textfile

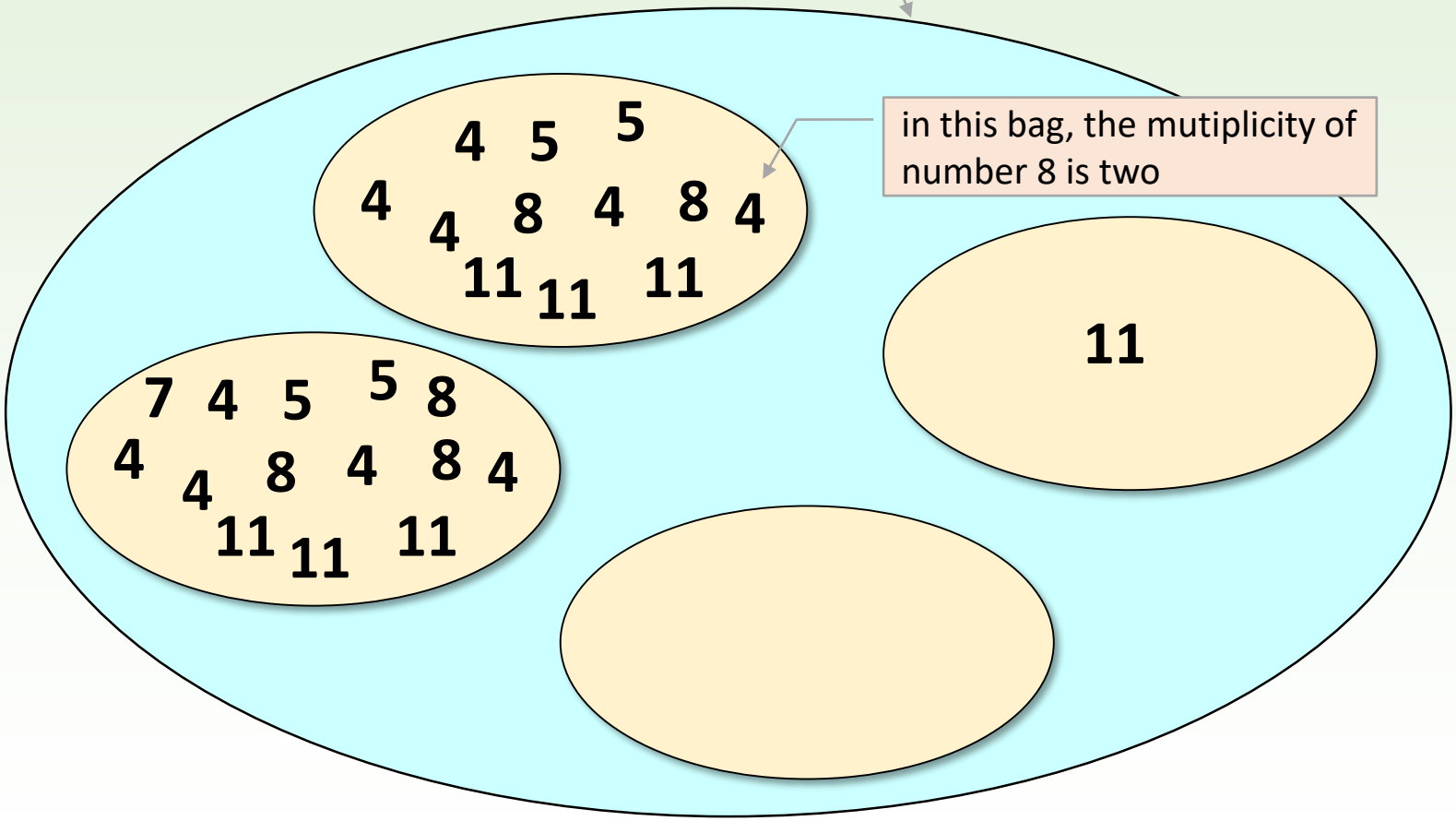
Datatype

- A datatype is given (specified) by its values and its operations.
- In certain cases we have to describe how we want to represent the values on the computer and how we implement the operations. These two are called type-realization.



Bag, set of values

The values that a Bag-type variable can have, together form the set of values of a Bag.



in this bag, the mutiplicity of number 8 is two

Bag, operations

Empty the bag (erase):

$b := \underline{\emptyset}$ $\quad b:Bag$

notation for empty bag

Insert an element into a bag (putIn):

error if $e \notin [0 .. m]$

$b := b \underline{\cup} \{e\}$ $\quad b:Bag, e:\mathbb{N}$

notation for„insert”

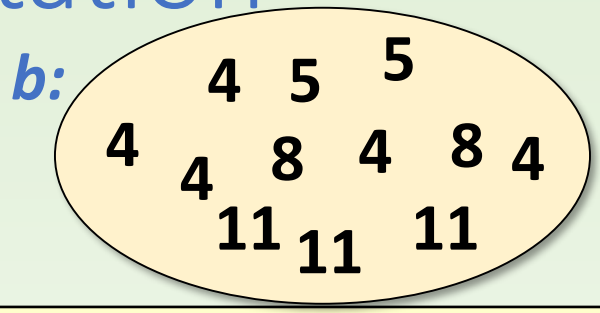
Most frequent element of the bag (mostFrequent) :

error if the bag is empty

$e := frequent(b)$ $\quad b:Bag, e:\mathbb{N}$

operation for giving the most frequent element

Representation



Here we take advantage of the fact that the elements of the bag are between 0 and m.

	0	1	2	3	4	5	6	7	8	9	10	11	…	m	
vec:	0	0	0	0	5	2	0	0	2	0	0	3	…		$: \mathbb{N}^{0..m}$

max: 4 $: \mathbb{N}$

we store separately the most frequent element

type invariant:

$max \in [0..m] \wedge$

properties of the representative attributes and their relation

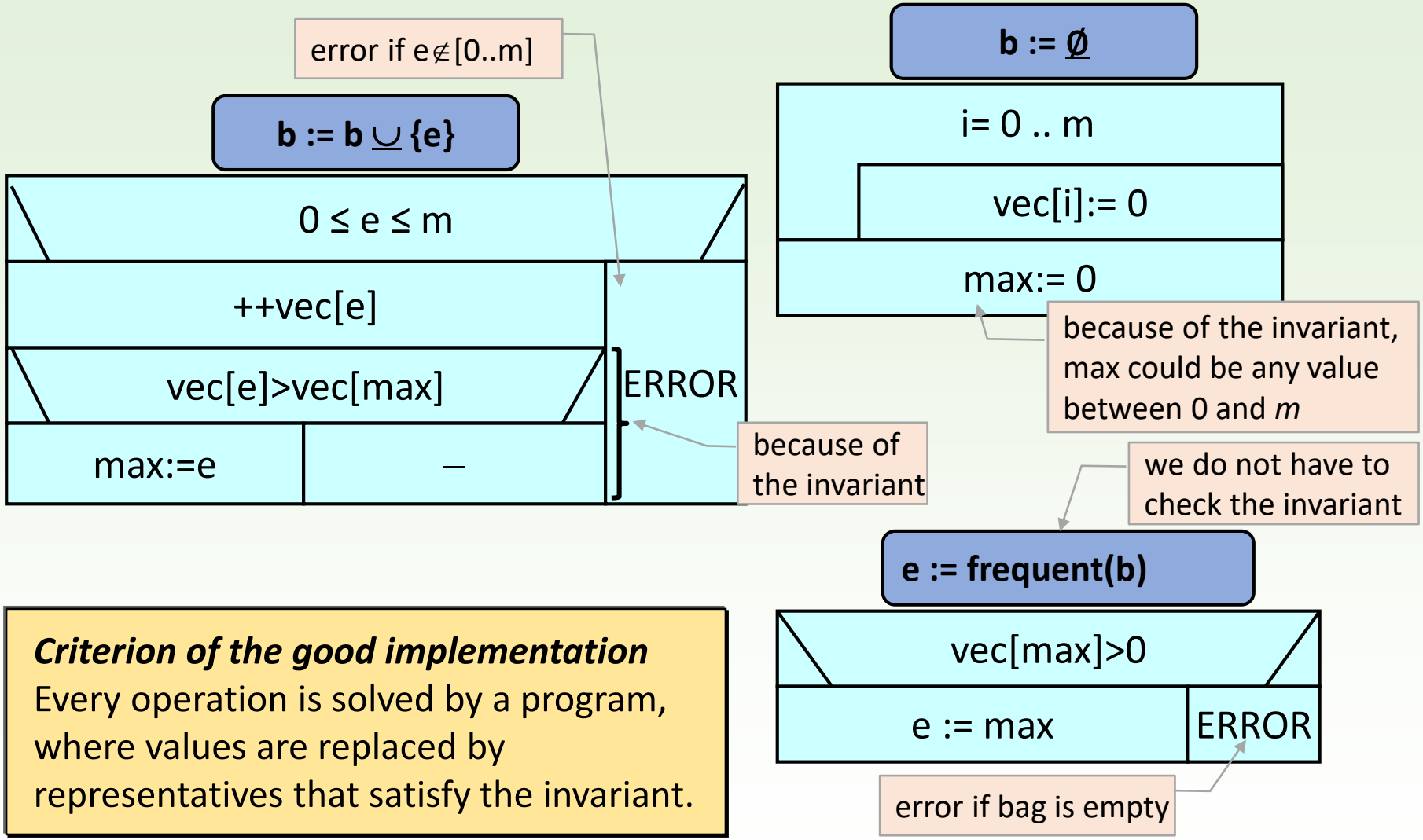
$vec[max] = \max_{i=0}^{m} vec[i]$

vec[max]=0 means the empty bag

Criterion of the good representation

Every value (bag) has a representative (here a pair of an array and a maximum value) that satisfies the invariant, and every representative corresponds to a value.

Implementation



Criterion of the good implementation
Every operation is solved by a program, where values are replaced by representatives that satisfy the invariant.

Type and class

- In case of object-oriented planning, type is realized by a class.

Type Bag

Type-specification

values

bags

operations

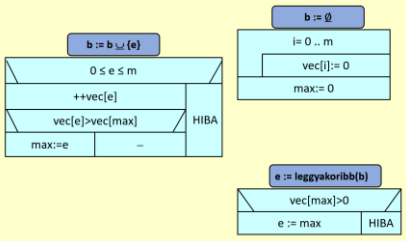
b:=∅
b:=b∪{e}
e:=frequent(b)
b:Bag, e:ℕ

representation

vec : $\mathbb{N}^{0..m}$
max : ℕ

{ max∈[0..m] ∧
vec[max] =
max$_{i=0..m}$ vec[i] }

implementation



Type-realization

Class Bag

{ max∈[0..m] ∧
vec[max] = **max**$_{i=0..m}$ vec[i] }

Bag

vec : int[0..m]
max : int

erase() : void
putIn(e:int) : void
mostFrequent() : int

```
for i=0..m loop vec[i] := 0 endloop
max := 0
```

```
if vec[max]>0 then
    e := max
endif
```

```
if 0≤e≤m then
    ++vec[i]
    if vec[e]>vec[max] then
        max := e
    endif
endif
```

UML diagram of a class

- Description of a class needs
  - its name, attributes, methods and their visibility, which can be
    - seen from outside: *public* (+)
    - hidden from outside: *private* (-) or *protected* (#)

<class name>
<+|-|#> <attribute name> : <type> …
<+|-|#> <method name>(<parameters>) : <type> …

Bag
- vec : int[0..m] - max : int
+ erase() : void + putIn(e:int) : void + mostFrequent() : int

Testing of type Bag

Test of Bag

erase():

- empty bag is created: b.mostFrequent()=0

putIn():

- put an element into an empty bag
- put an existing element into a non-empty bag
- put a non-existing element into a non-empty bag
- put 0 and m into the bag
- put an illegal element into the bag

Invariant has to be satisfied every time we put in an element.

mostFrequent(): (like a maximum search)

- in case of empty bag
- in case of bag with one element
- in case of bag with more elements and the most frequent one is obvious
- in case of bag with more elements, the most frequent one is not obvious
- the most frequent element is 0 or *m*

Integration testing (modules a tested together as a group)

- after b.erase(), b:=b.putIn(e) is run many times
- after b.erase(), e:=b.mostFrequent() is run immediately

Test cases for Bag

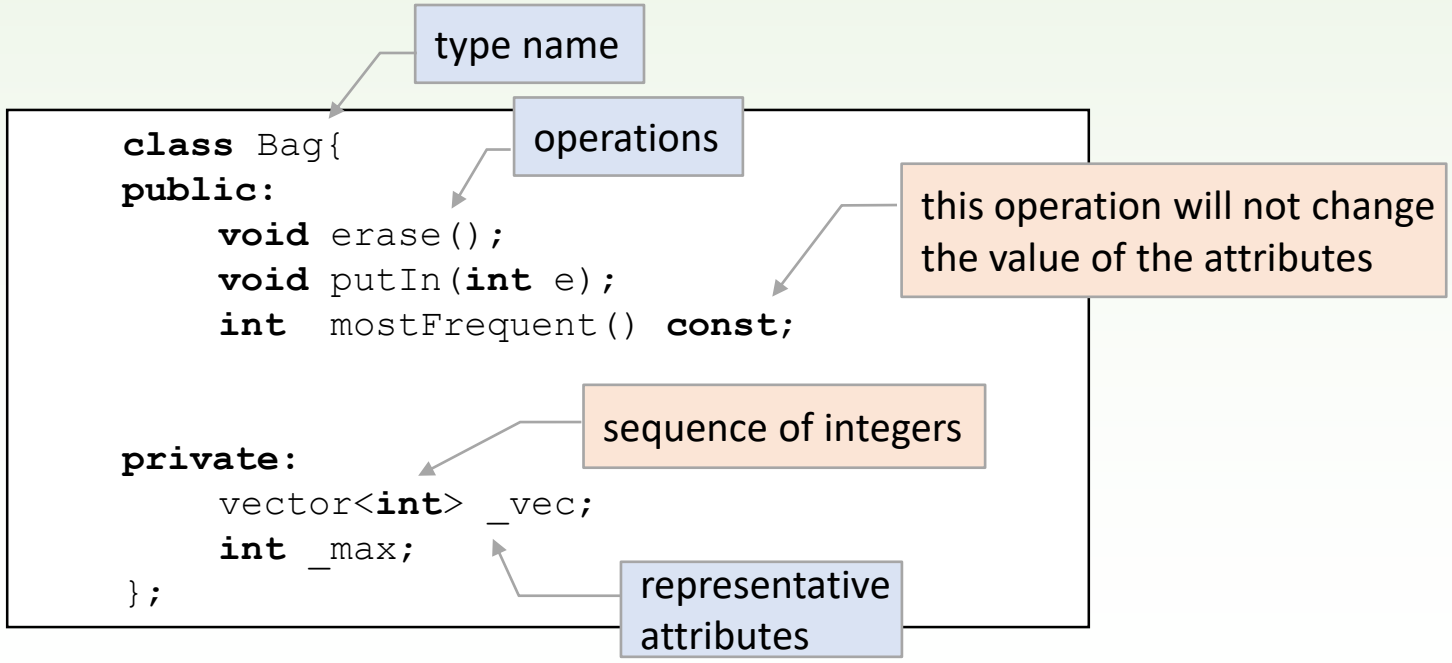
mostFrequent()	Test data		
empty bag	m = 1	vec = < 0, 0 >	→ error
one element	m = 1	vec = < 1, 0 >	→ 0
obvious maximum	m = 1	vec = < 2, 1 >	→ 0
more maxima	m = 1	vec = < 2, 2 >	→ 0 ∨ 1
max is 0	m = 1	vec = < 1, 0 >	→ 0
max is *m*	m = 1	vec = < 0, 1 >	→ 1

erase()	Test data		
create empty bag	m = 0	→ vec = < 0 >	max = 0
	m = 1	→ vec = < 0, 0 >	max = 0 ∨ 1
	m = 4	→ vec = < 0, 0, 0, 0, 0 >	max = 0

putIn()	Test data				
first	m = 1	vec = < 0, 0 >	putIn(0)	→ vec = < 1, 0 >	max = 0
new	m = 1	vec = < 1, 0 >	putIn(1)	→ vec = < 1, 1 >	max = 0
existing	m = 1	vec = < 1, 1 >	putIn(1)	→ vec = < 1, 2 >	max = 1
0	m = 1	vec = < 1, 1 >	putIn(0)	→ vec = < 2, 1 >	max = 0
m	m = 1	vec = < 1, 1 >	putIn(1)	→ vec = < 1, 2 >	max= 1
illegal	m = 1	vec = < 1, 2 >	putIn(2)	→ error	

Class from type

- In object-oriented languages, custom types are realized by classes.
  - the name of the class has to match the name of the type
  - the attributes of the representation are the attributes of the class (with name and type)
  - the operations are the methods of the class



Basic terms

e.g. the bag

- Object is responsible for a part of the task, that contains the corresponding data and operations.

vec and *max*

erase, putIn, mostFrequent

- Class gives the sample structure and behaviour of an object
  - enumerates the object's attributes (name and type)
  - gives the methods that can be called on the object
- Class is like a data type of the object: object is created based on its class, it is instantiated.
- Based on a class, more objects can be instantiated.

vec is an array, *max* is an integer

erase() : void,
putIn(int) : void,
mostFrequent() : int

1

Important criterion of object-orientation is encapsulation: the data and its manipulation needed for a domain are given as a unit, separately from the other parts of the program.

Visibility

```
class Bag{
public:
     void erase();
     void putIn(int e);
     int  mostFrequent() const;
private:
     vector<int> _vec;
     int _max;
};

int main()
{
     Bag b;
     b.erase();
     b.putIn(5);
     int a = b.mostFrequent();

     b._vec[5]++;
}
```

Methods of an object are always called on the object itself (which is a special parameter of the method). In the body of the method we can access the attributes and we can call other methods of the object.

Outside the class the private attributes are unavailable.

2 Important criterion of object-orientation is data hiding: the visibility of the encapsulated elements are restricted. (Usually, attributes are hidden, they are only accessible through public methods.)

Constructor

When an object is created (instantiated), a special method called constructor is called, that runs the constructor of the attributes.

Every object, as long as other constructor is not set, has an empty constructor, that does not need any parameters. It just calls the empty constructor of the attributes. In our case, declaration *Bag b* is meaningful, it creates a vector of length 0 and an integer. Unfortunately, it is not enough for us.

```
class Bag{
public:
    Bag(int m);
    …
private:
    vector<int> _vec;
    int _max;
};
```

A constructor is needed where the length of the array can be set. Command *Bag b(21)* would instantiate the bag with an array of length 22.

it belongs to class Bag

the constructor resizes the vector of length 0 to length of *m*+1 and runs method *erase()*.

```
Bag::Bag(int m) { _vec.resize(m+1); erase(); }
```

it does the job of erase(), too

```
Bag::Bag(int m) { _vec.resize(m+1, 0); _max = 0; }
```

not empty constructor of the vector and the int

```
Bag::Bag(int m) : _vec(m+1,0), _max(0) { }
```

Class of Bag in details

header files and guards

```
#pragma once
#include <vector>

class Bag{
public:
     enum Errors{EmptyBag, WrongInput};

     Bag(int m);
     void erase();
     void putIn(int e);
     int  mostFrequent() const;
private:
     std::vector<int> _vec;
     int _max;
};
```

error cases:
enum is a new type that has only values

bag.h

In the header files *using namespace std* is not common.
We have to indicate that the definition of vector is in namespace std

Methods of class Bag

Class definition of *Bag* is needed

```
#include "bag.h"

Bag::Bag(int m) : _vec(m+1,0), _max(0) { }

void Bag::erase() {
     for(unsigned int i=0; i<_vec.size(); ++i) _vec[i] = 0;
     _max = 0;
}

void Bag::putIn(int e) {
     if( e<0 || e>=int(_vec.size()) ) throw WrongInput;
     if( ++_vec[e] > _vec[_max] ) _max = e;
}

int Bag::mostFrequent() const {
     if( 0 ==_vec[_max] ) throw EmptyBag;
     return _max;
}
```

length of _vec

cast to int

throws an exception: indicates the error but does not handle it

throws an exception

belongs to class Bag

bag.cpp

Main program

```
#include <iostream>
#include <fstream>
#include "bag.h"
using namespace std;

int main()
{
     ifstream f( "input.txt" );
     if(f.fail()){ cout << "Wrong file name!\n"; return 1;}
     int m; f >> m;
     if(m<0){
          cout << "Upper limit of natural numbers cannot be negative!\n";
          return 1;
     }
     try{
        Bag b(m);
        int e;
        while(f >> e) { b.putIn(e); }
        cout << "The most frequented element: "<< b.mostFrequent() << endl;
     }catch(Bag::Errors ex){
          if      (ex==Bag::WrongInput){ cout << "Illegal integer!\n"; }
          else if(ex==Bag::EmptyBag)   { cout << "No input no maximum!\n"; }
     }
     return 0;
}
```

Class definition of *Bag* is needed

exception filter

catches an exception and handles it

main.cpp

https://github.com/philsquared/Catch/blob/master/docs/tutorial.md

Automatic test

```
#include <iostream>
#include <fstream>
#include "bag.h"
using namespace std;

#define CATCH_CONFIG_MAIN
#include "catch.hpp"

 TEST_CASE("empty file", "[sum]"){
      ifstream f( "input1.txt" );
      int m; f >> m;
      Bag b(m); b.erase();
      int e;
      while(f >> e){ b.putIn(e); }
      CHECK_THROWS(b.mostFrequent());
 }
 TEST_CASE("one element in the file", "[sum]"){
      ifstream f( "input2.txt" );
      int m; f >> m;
      Bag b(m); b.erase();
      int e;
      while(f >> e){ b.putIn(e); }

      CHECK(b.mostFrequent()==2);
 }
 …
```

summation
length if interval: 0
f = < 15 > (m = 15)
b = { } → no frequent element
throw Bag::EmptyBag

summation
length if interval: 1
f = < 15, 2 > (m = 15)
b = { 2(1) } → mostFrequent = 2

https://github.com/philsquared/Catch/blob/master/docs/tutorial.md

Automatic test

```
TEST_CASE("create an empty bag", "[bag]")
{
     int m = 0;
     Bag b(m);
     vector<int> v = { 0 };
     CHECK(v == b.getArray());
}
TEST_CASE("new element into empty bag", "[putIn]")
{
     Bag b(1);
     b.putIn(0);
     vector<int> v = { 1, 0 };
     CHECK(v == b.getArray());
}
…
```

bag()
create an empty bag:
m = 0 → _vec = < 0 >

getArray() may come in handy for the testing

putIn()
new element into empty bag:
m = 1, e = 0 → _vec = < 1, 0 >

```
class Bag {
private:
     std::vector<int> _vec;
     int _max;
public:
     …
     const std::vector<int>& getArray() const {return _vec;}
};
```

Not a nice solution, it is like "littering" in the code. It would be more elegant to create a class Bag_Test that inherits all the properties of class Bag. Then, it could be extended with method getArray(). The test cases could use class Bag_Test instead of Bag.

"inline" definition