Abstract Data Types (II)
(and Object-Oriented Programming)

Jordi Cortadella and Jordi Petit
Department of Computer Science

Public or private?

- What should be public?
  - Only the methods that need to interact with the external world. Hide as much as possible. Make a method public only if necessary.
- What should be private?
  - All the attributes.
  - The internal methods of the class.
- Can we have public attributes?
  - Theoretically yes (C++ and python allow it).
  - Recommendation: never define a public attribute. Why? See the next slides.

Class Point: a new implementation

- Let us assume that we need to represent the point with polar coordinates for efficiency reasons (e.g., we need to use them very often).

- We can modify the private section of the class without modifying the specification of the public methods.

- The private and public methods may need to be rewritten, but not the programs using the public interface.

Let us design the new type for Point

```
// The declaration of the class Point
class Point {

public:
  // Constructor
  Point(double x_coord, double y_coord);

  // Constructor for (0,0)
  Point();

  // Gets the x coordinate
  double getX() const;

  // Gets the y coordinate
  double getY() const;

  // Returns the distance to point p
  double distance(const Point& p) const;

  // Returns the distance to the origin
  double distance() const;

  // Returns the angle of the polar coordinate
  double angle() const;

  // Creates a new point by adding the coordinates of two points
  Point operator + (const Point& p) const;

private:
  double _radius, _angle;  // Polar coordinates

};
```

Class Point: a new implementation

```
Point::Point(double x, double y) :
  _radius(sqrt(x*x + y*y)),
  _angle(x == 0 and y == 0 ? 0 : atan(y/x))
{}

double Point::getX() const {
  return _radius*cos(_angle);
}

double Point::getY() const {
  return _radius*sin(_angle);
}

double Point::distance(const Point& p) const {
  double dx = getX() - p.getX();
  double dy = getY() - p.getY();
  return sqrt(dx*dx + dy*dy);
}

double Point::distance() const {
  return _radius;
}
```

Class Point: a new implementation

```
double Point::angle() const {
  return _angle;
}

// Notice that no changes are required for the + operator
// with regard to the initial implementation of the class
Point Point::operator + (const Point& p) const {
  return Point(getX() + p.getX(), getY() + p.getY());
}
```

Discussion:

- How about having `x` and `y` (or `_radius` and `_angle`) as public attributes?
- Programs using `p.x` and `p.y` would not be valid for the new implementation.
- Programs using `p.getX()` and `p.getY()` would still be valid.

Recommendation (reminder):

- All attributes should be ***private***.

Public/private: let's summarize

```
class Z {
public:
  …
  void f(const Z& x) {
    … a … // "this" attribute
    … x.a … // x's attribute

    g(); // Ok
    x.g(); // Ok
  }
  …
private:
  T a; // Attribute

  …
  void g(…) {…}
  …
};
```

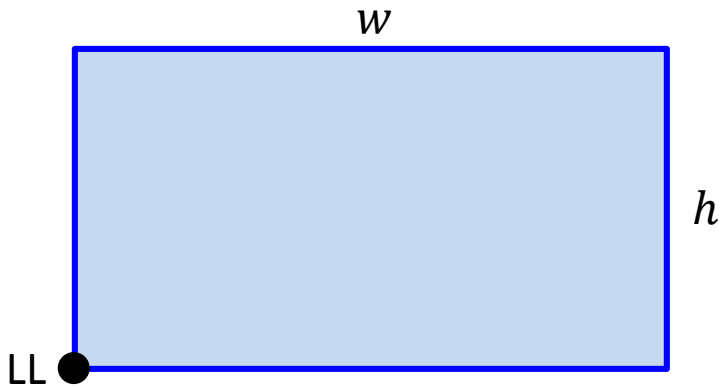
```
int main () {

  Z v1, v2;
  …
  v1.f(v2); // Ok
  …

  v1.g(…); // Wrong! (private)

  … v1.a … // Wrong! (private)

  v1 = v2; // Ok (copy)

}
```

A new class: Rectangle

- We will only consider orthogonal rectangles (axis-aligned).

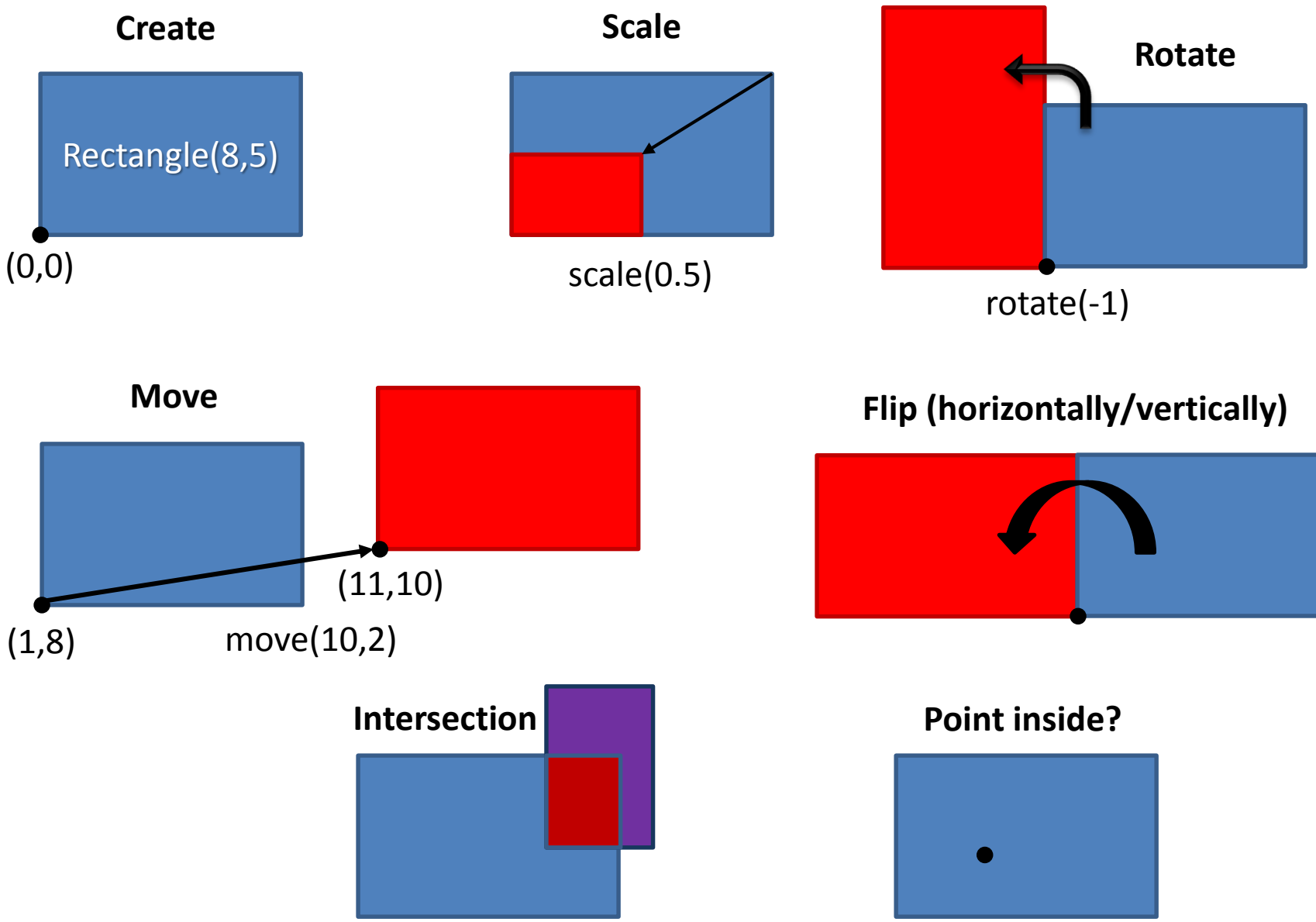
- An orthogonal rectangle can be represented in different ways:

Two points (extremes of diagonal)



One point, width and height

Rectangle: abstract view



Rectangle: ADT (incomplete)

```
class Rectangle {
public:
  // Constructor (LL at the origin)
  Rectangle(double width, double height);

  // Returns the area of the rectangle
  double area() const;

  // Scales the rectangle with a factor s > 0
  void scale(double s);

  // Returns the intersection with another rectangle
  Rectangle operator * (const Rectangle& R) const;

  ...

};
```

Rectangle: using the ADT

```
Rectangle R1(4,5);  // Creates a rectangle 4x5
Rectangle R2(8,4);  // Creates a rectangle 8x4

R1.move(2,3);               // Moves the rectangle
R1.scale(1.2);              // Scales the rectangle
double Area1 = R1.Area(); // Calculates the area

Rectangle R3 = R1 * R2;

if (R3.empty()) …
```

Rectangle: ADT

```
class Rectangle {
public:
```

```
private:
  Point  ll;          // Lower-left corner of the rectangle
  double w, h;        // width/height of the rect.
```

Other private data and methods (if necessary)

```
};
```

Rectangle: a rich set of constructors

```
// LL at the origin
Rectangle::Rectangle(double w, double h) :
  ll(Point(0,0)), w(w), h(h) {}

// LL specified at the constructor
Rectangle::Rectangle(const Point& p, double w, double h) :
  ll(p), w(w), h(h) {}

// LL and UR specified at the constructor
Rectangle::Rectangle(const Point& ll, const Point& ur) :
  ll(ll), w(ur.getX() - ll.getX()), h(ur.getY() - ll.getY())
{}

// Empty rectangle (using another constructor)
Rectangle::Rectangle() : Rectangle(0, 0) {}
```

Rectangle: overloaded operators

```
Rectangle& Rectangle::operator *= (const Rectangle& R) {
  // Calculate the ll and ur coordinates
  Point Rll = R.getLL();
  ll = Point(max(ll.getX(), Rll.getX()),
             max(ll.getY(), Rll.getY()));

  Point ur = getUR();
  Point Rur = R.getUR();
  double urx = min(ur.getX(), Rur.getX());
  double ury = min(ur.getY(), Rur.getY());

  // Calculate width and height (might be negative → empty)
  w = urx - ll.getX();
  h = ury - ll.getY();

  return *this;
}

// Use *= to implement *
Rectangle Rectangle::operator * (const Rectangle& R) const {
  Rectangle result = *this;  // Make a copy of itself
  result *= R;
  return result;
}
```

Rectangle: other public methods

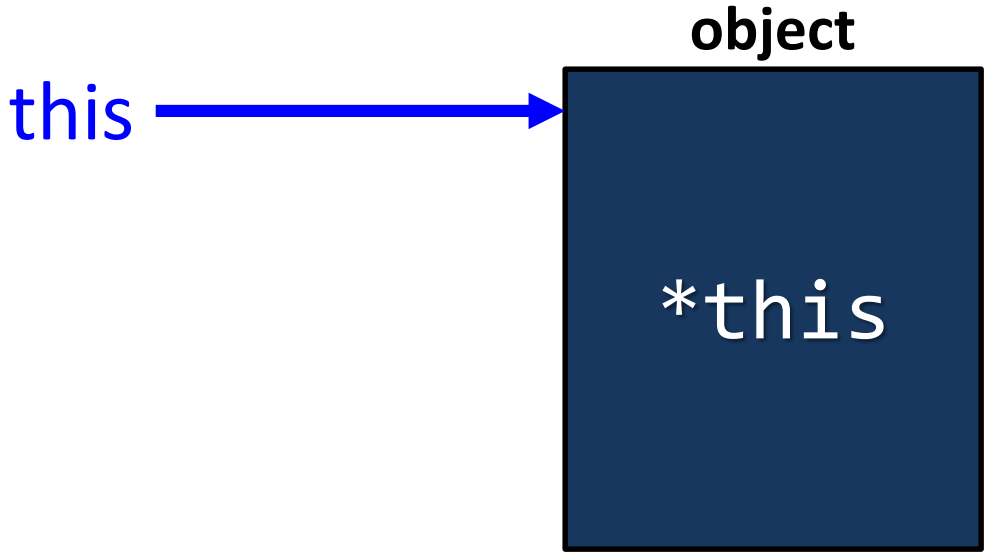
```
Point Rectangle::getLL() const {
  return ll;
}

Point Rectangle::getUR() const {
  return ll + Point(w, h);
}

double Rectangle::getWidth() const {
  return w;
}

double Rectangle::getHeight() const {
  return h;
}
```

```
double Rectangle::area() const {
  return w*h;
}

// Notice: not a const method
void Rectangle::scale(double s) {
  w *= s;
  h *= s;
}

bool Rectangle::empty() const {
  return w <= 0 or h <= 0;
}
```

What is *this?

- `this` is a pointer (memory reference) to the object (pointers will be explained later)

- `*this` is the object itself



Let us work with rectangles

```
Rectangle R1(Point(2,3), Point(6,8));
double areaR1 = R1.area();  // areaR1 = 20

Rectangle R2(Point(3,5), 2, 4); // LL=(3,5) UR=(5,9)

// Check whether the point (4,7) is inside the
// intersection of R1 and R2.
bool in = (R1*R2).isPointInside(Point(4,7));
// The object R1*R2 is "destroyed" after the assignment.

R2.rotate(false);  // R2 is rotated counterclockwise
R2 *= R1;          // Intersection with R1
```

Exercise: draw a picture of R1 and R2 after the execution of the previous code.

Yet another class: Rational

What we would like to do:

```
Rational R1(3);         // R1 = 3
Rational R2(5, 4);      // R2 = 5/4
Rational R3(8, -10);    // R3 = -4/5

R3 += R1 + Rational(-1, 5);   // R3 = 2

if (R3 >= Rational(2)) {
  R3 = -R1*R2;          // R3 = -15/4
}

cout << R3.to_str() << endl;
```

The class Rational

```
class Rational {
private:
  int num, den; // Invariant: den > 0 and gcd(num,den)=1

  // Simplifies the fraction (used after each operation)
  void simplify();

public:
  // Constructor (if some parameter is missing, the default value is taken)
  Rational(int num = 0, int den = 1);

  // Returns the numerator of the fraction
  int getNum() const {
    return num;
  }

  // Returns the denominator of the fraction
  int getDen() const {
    return den;
  }

  // Returns true if the number is integer and false otherwise.
  bool isInteger() const {
    return den == 1;
  }
  ...

};
```

The class Rational

```
class Rational {

public:

  ...
  // Arithmetic operators
  Rational& operator += (const Rational& r);
  Rational operator + (const Rational& r) const;
  // Similarly for -, * and /

  // Unary operator
  Rational operator – () const {
    return Rational(-getNum(), getDen());
  }

  // Equality comparison
  bool operator == (const Rational& r);

  // Returns a string representing the rational
  string to_str() const;
};
```

Rational: constructor and simplify

```
Rational::Rational(int num, int den) : num(num), den(den) {
  simplify();
}

void Rational::simplify() {
  assert(den != 0);  // We will discuss assertions later
  if (den < 0) {
    num = -num;
    den = -den;
  }

  // Divide by the gcd of num and den
  int d = gcd(abs(num), den);
  num /= d;
  den /= d;
}
```

Rational: arithmetic and relational operators

```
Rational& Rational::operator += (const Rational& r) {
  num = getNum()*r.getDen() + getDen()*r.getNum();
  den = getDen()*r.getDen();
  simplify();
  return *this;
}

Rational Rational::operator + (const Rational& r) {
  Rational result = *this; // A copy of this object
  result += r;
  return result;
}

bool Rational::operator == (const Rational& r) {
  return getNum() == r.getNum() and getDen() == r.getDen();
}

bool Rational::operator != (const Rational& r) {
  return not operator ==(r);
}

string Rational::to_str() const {
  string s(to_string(getNum()));
  if (not isInteger()) s += "/" + to_string(getDen());
  return s;
}
```

Classes and Objects in Python

A Python session with rational numbers

```
>>> from rational import Rational # from file rational.py
>>> a = Rational(4, -6)      # construct with num and den
>>> print(a)
-2/3
>>> b = Rational(4)                    # integer value
>>> print(b)
4
>>> print((a+b).num(), (a+b).den())
10 3
>>> c = Rational()                          # c = 0
>>> if a < c:
...     print(a, "is negative")
...
-2/3 is negative
>>> print(a*b)      # uses the __str__ method (see later)
-8/3
>>> a/b            # uses the __repr__ method (see later)
Rational(-1/6)
>>>
```

The Rational class in Python

```
class Rational:

  def __init__(self, num=0, den=1):
    if not isinstance(num, int):
      raise TypeError("The numerator is not an integer")
    if not isinstance(den, int):
      raise TypeError("The denominator is not an integer")
    if den == 0:
      raise ZeroDivisionError("The denominator is zero")
    self._num = num
    self._den = den
    self._simplify()
```

Classes only have one constructor (**__init__**). Multiple constructors can be "*simulated*" by checking the number and type of arguments.

All class methods and attributes are public.
Naming convention: use underscore prefix for *"private"* attributes and methods (e.g., `_num`, `_den`, `_simplify`)

self in Python is similar to **this** in C++.
All methods must be declared with **self** as first argument.
Exception: static methods (not discussed here).

The Rational class in Python

Disclosure: recommended indentation is 4 spaces (here we use only 2 spaces for convenience). Comments are not included, but they should be there !

```
class Rational:

        ⋮

  def num(self):          # Public method
    return self._num

  def den(self):          # Public method
    return self._den

  def _simplify(self):    # Private method
    if self._den < 0:
      self._num *= -1
      self._den *= -1
    d = math.gcd(abs(self._num), self._den)
    self._num //= d
    self._den //= d
```

Python __underscore__ methods

Methods with double leading and trailing underscore have special meanings for the Python language. Recommendation: avoid this naming scheme for your methods and attributes.

```
class Rational:

        ⋮

  def __str__(self):
    """Returns a user-friendly string with information
    about the value of the object. It is invoked by
    str(x) or print(x)."""
    if self._den == 1:
      return str(self._num)
    return str(self._num) + "/" + str(self._den)

  def __repr__(self):
    """Returns a string with information about the
    representation of the class. It is invoked by repr(x)
    or simply 'x'."""
    return "Rational(" + str(self) + ")"
```

Arithmetic operators

```
class Rational:

        ⋮

  def __neg__(self):
    """Returns -self."""
    return Rational(-self._num, self._den)


  def __add__(self, rhs):
    """Returns self + rhs."""
    num = self._num*rhs._den + self._den*rhs._num
    den = self._den*rhs._den
    return Rational(num, den)

  # Similarly for __sub__, __mul__, __truediv__
```

Relational operators

```
class Rational:

        ⋮

  def __eq__(self, rhs):
    """Checks whether self == rhs."""
    return self._num == rhs._num and self._den == rhs._den

  def __ne__(self, rhs):
    """Checks whether self != rhs."""
    return not self == rhs

  def __lt__(self, rhs):
    """Checks whether self < rhs."""
    return self._num*rhs._den < self._den*rhs._num

  def __le__(self, rhs):
    """Checks whether self <= rhs."""
    return not rhs < self

  # Similarly for __gt__ and __ge__
```

Python documentation: *docstrings*

```
>>> from rational import Rational
>>> help(Rational.__add__)
Help on function __add__ in module rational:

__add__(self, rhs)
    Returns self + rhs.
>>> help(Rational)
class Rational(builtins.object)
 |  Rational(num=0, den=1)
 |
 |  Class to manipulate rational numbers.
 |
 |  The class includes the basic arithmetic and relational
 |  operators.
 |
 |  Methods defined here:
 |
 |  __add__(self, rhs)
 |      Returns self + rhs.
 |
 |  __eq__(self, rhs)
 |      Checks whether self == rhs.
```

Python documentation: *docstrings*

- The first line after a module, class or function can be used to insert a string that documents the component.
- Triple quotes (""") are very convenient to insert multi-line strings.
- The ***docstrings*** are stored in a special attribute of the component named __doc__.
- Different ways of print the ***docstrings*** associated to a component:
  - `print(Rational.num.__doc__)`
  - `help(Rational.num)`

Designing a module: example

```
# geometry.py

"""geometry.py
Provides two classes for representing Polygons and Circles."""

# author: Euclid of Alexandria

from math import pi, sin, cos


class Polygon:

    """Represents polygons and provides methods to
    calculate area, intersection, convex hull, etc."""

    def __init__(self, list_vertices):
        """Creates a polygon from a list of vertices."""
    ...

class Circle:
    ...
```

Documentation of the module

Documentation of the class

Documentation of the method

Using a module: example

```
import geometry
p = geometry.Poligon(…)
c = geometry.Circle(…)
```

Imports the module. Now all classes can be used with the prefix of the module.

```
import geometry as geo
p = geo.Poligon(…)
c = geo.Circle(…)
```

Imports and renames the module.

```
from geometry import *
p = Poligon(…)
c = Circle(…)
```

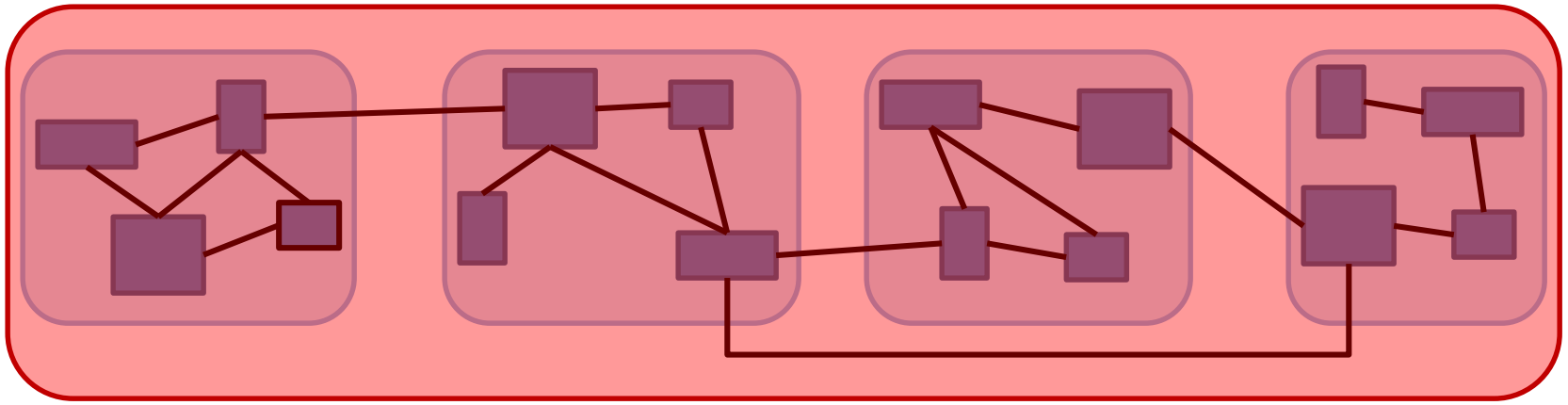
Imports all classes in the module. No need to add the prefix of the module.

```
from geometry import Poligon as plg, Circle as cir
p = plg(…)
c = cir(…)
```

Imports and renames the classes in the module.

Conclusions

- Finding the appropriate hierarchy is a fundamental step towards the design of a complex system.
- User-friendly documentation is indispensable.



EXERCISES

Implement methods

Implement the following methods for the class Rectangle:

```
// Rotate the rectangle 90 degrees clockwise or
// counterclockwise, depending on the value of the parameter.
// The rotation should be done around the lower-left corner
// of the rectangle.
void rotate(bool clockwise);

// Flip horizontally (around the left edge) or vertically (around
// the bottom edge), depending on the value of the parameter.
void flip(bool horizontally);

// Check whether point p is inside the rectangle
bool isPointInside(const Point& p) const;
```

Re-implement a class

Re-implement the class Rectangle using an internal representation with two Points:

- Lower-Left (LL)
- Upper-Right(UR)