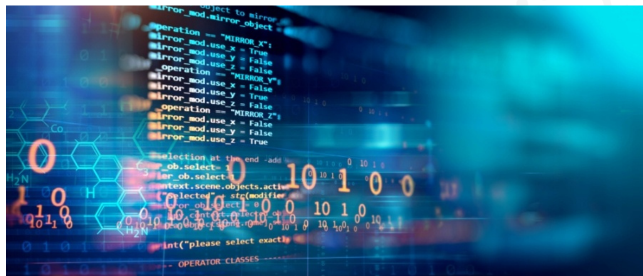
Clases

Tecnología de Programación



Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-01-28

TADs vs Clases

Recordando los **registros**...

Recordando los **registros**...

Y la implementación de TADs mediante **registros**...

Recordando los **registros**...

Y la implementación de TADs mediante **registros**...

Define un tipo abstracto de datos `Rational` que represente un número racional.

```cpp
struct Rational
{
  int num;
  int den;
};
```

```cpp
#include <iostream>
#include "rational.h"

int main(int argc, char** argv)
{
  Rational r1, r2;
  r1.num = 1; r1.den = 5;
  f2.num = 2; r2.den = 3;

  cout << r1.num << "/" << r1.den << ", ";
  cout << r2.num << "/" << r2.den << endl;

  Rational rs[10];
  ...
}
```

Define un tipo de abstracto de datos **`Rational`** que represente un número racional, que oculte su implementación al usuario, y que por tanto incluya operaciones para:

- Construir racionales, a partir de numerador y denominador.
- Sumar, restar, multiplicar y dividir racionales, bien con otros racionales o bien con números enteros.
- Leer y escribir racionales.

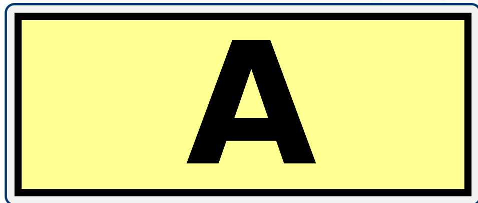
En un proyecto grande en el que tuvieras que utilizar números racionales seguramente tendrías:

- El `struct` que define el tipo racional.
- Una serie de funciones para operar con racionales: sumar, restar, multiplicar, dividir ...

De ese modo conseguirías manejar un **tipo abstracto de datos**.
A partir de ese momento tu programa principal no necesitaría acceder directamente al registro sino a las correspondientes funciones.

Como la abstracción de datos es útil (muy útil) (extremadamente útil) se inventó el concepto de **clase**:

Como la abstracción de datos es útil (muy útil) (extremadamente útil) se inventó el concepto de **clase**:



Similar a un tipo de datos.
A la variable que se crea como una *instancia* de esa clase se le denomina un **objeto**.

Objetivos de la Programación Orientada a Objetos:

- Ocultar completamente la implementación de las estructuras de datos.
- Centrar el diseño del software en las estructuras de datos, y en sus relaciones, no en los algoritmos.
- Hacer que el comportamiento de las estructuras de datos pudiera cambiar durante la ejecución del programa.

En Smalltalk, hasta las estructuras de control son objetos...

```
(a > 0) ifTrue:  [ 'positive' printNL ]
        ifFalse: [ 'negative' printNL ]
```

≈ (traducción muy libre)

```
(a > 0).ifTrueifFalse( {'positive'.printNL()} ,
                       {'negative'.printNL()}    );
```

Filosofía de trabajo con **TAD**s:

- El TAD es un **tipo** de datos del lenguaje, las variables se declaran:

```
struct String { ... }
...
String str;
```

- Tenemos **funciones** que trabajan con la variable:

```
l = length(str)
clear(str)
```

- Que pueden necesitar **parámetros** adicionales:

```
fill(str,16,'x')
```

Filosofía de trabajo con **clases** (original, estilo Smalltalk):

- La clase es un **patrón**, los objetos se construyen como clones de la clase.

```
class String { ... }
...
str = String <- "clone"
```

- Para trabajar con los objetos les enviamos **mensajes**:

```
l = str <- "length"
str <- "clear"
```

- Que pueden necesitar **parámetros** adicionales:

```
str <- "fill" 16 "with" 'x'
```

Filosofía de trabajo con **clases** (original, estilo Smalltalk):

- Los mensajes posibles son propios de cada clase:

```
class String:
    int  sz
    char text[...]

    (int)  length : [ ... ]
    (void) clear  : [ ... ]
    (void) fill   : (int)n  with: (char)c | [ ... ]
```

Filosofía de trabajo con **clases** (sintaxis adaptada):

- La clase es un **tipo** de datos, los objetos (variables) se declaran:

```
class String { ... }
...
String str;
```

- Para trabajar con los objetos les enviamos **mensajes**:

```
l = str.length()
str.clear()
```

- Que pueden necesitar **parámetros** adicionales, como una función:

```
str.fill(16,'x')
```

Filosofía de trabajo con **clases** (sintaxis adaptada):

- Los mensajes (funciones) posibles son propios de cada clase, se definen de forma similar a las componentes de datos
  (por consistencia con el operador '.' para los datos):

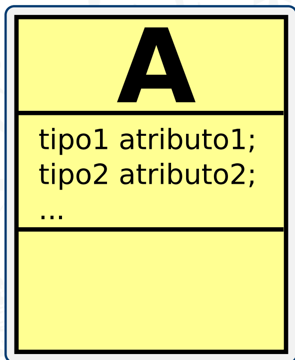
```
class String {
    int  sz;
    char text[...];

    int  length();
    void clear();
    void fill(int sz,char c);
}
```

Atributos y métodos

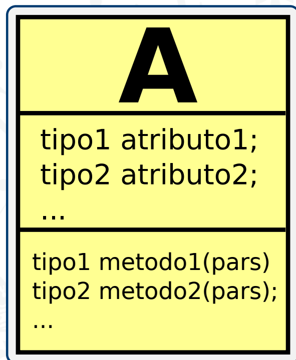
Como la abstracción de datos es útil se inventó el concepto de **clase**.

- Sus componentes se llaman **atributos**.
- Una clase que sólo tiene atributos es exactamente idéntica a un registro (`struct`).



Como la abstracción de datos es útil se inventó el concepto de **clase**.

- Sus componentes se llaman **atributos**.
- Sus funciones se llaman **métodos**.



Como la abstracción de datos es útil se inventó el concepto de **clase**.

- Sus componentes se llaman **atributos**.
- Sus funciones se llaman **métodos**.

```
class A {
    tipo1 atributo1;
    tipo2 atributo2;
    ...
    tipo metodo(parametros)
    {
        implementacion
    }
    ...
};
```

Implementación de operaciones con **TAD**s:

- El dato al que se le aplica la operación se pasa como argumento a una función:

```
fill(str,16,' ');
```

Implementación de operaciones con **TAD**s:

- Las funciones se aplican a un dato especificado como parámetro:

```
void fill(String& str,int count,char c)
{
    str.sz = count;
    for (int i=0; i<str.sz; i++)
      str.text[i] = c;
}
```

Implementación de operaciones con **clases**:

- El dato al que se le aplica la operación se especifica como receptor del mensaje:

```
str.fill(16,' ');
```

- El concepto de envío de un mensaje se sustituye por el acceso ('.') a un *método* de los disponibles en la clase:

```
str.fill(16,' ');
```

- Ese acceso provoca la llamada al *método*, de forma similar a una función:

```
str.fill(16,' ');
```

Implementación de operaciones con **clases**:

- Si la operación necesita argumentos (además del propio objeto), se envían como parámetros:

```
str.fill(16,' ');
```

- Los métodos referencian el objeto mediante una variable implícita (oculta, no se declara) llamada `this`, que está definida como un puntero al objeto:

```
void fill(int count,char c)
{
    this->sz = count;
    for (int i=0; i<this->sz; i++)
      this->text[i] = c;
}
```

Implementación de operaciones con **clases**:

- Si la operación necesita argumentos (además del propio objeto), se envían como parámetros:

```
str.fill(16,' ');
```

- Mediante las reglas de **resolución de ámbito** se simplifica el uso de `this`:

```
void fill(int count,char c)
{
    sz = count;
    for (int i=0; i<sz; i++)
      text[i] = c;
}
```

Implementación de operaciones con **clases**:

- Mediante las reglas de **resolución de ámbito** se simplifica el uso de `this`. Aparece el **ámbito de clase**:

```
class String {
    int  sz;
    char text[...];

    void fill(int count,char c)
    {
        sz = count;
        for (int i=0; i<sz; i++)
          text[i] = c;
    }
}
```

La resolución de nombres puede hacerse de dos formas:

explícita, con this

```
void fill(int count,char c)
{
    this->sz = count;
    for (int i=0; i<this->sz; i++)
      this->text[i] = c;
}

String one,two;

one.fill(8,'1');
    // this ≡ &one

two.fill(8,'2');
    // this ≡ &two
```

implícita, por el *ámbito*

```
void fill(int count,char c)
{
    sz = count;
    for (int i=0; i<sz; i++)
      text[i] = c;
}

String one,two;

one.fill(8,'1');
  // sz   ≡ one.sz
  // text ≡ one.text
two.fill(8,'2');
  // sz   ≡ two.sz
  // text ≡ two.text
```

El uso de `this` sólo es obligatorio para resolver ambigüedades...

```
class String {
    int  sz;
    char text[...];

    void fill(int sz,char c)
    {
        sz = sz;
        for (int i=0; i<sz; i++)
          text[i] = c;
    }
}
```

El uso de `this` sólo es obligatorio para resolver ambigüedades...

```
class String {
    int  sz;
    char text[...];

    void fill(int sz,char c)
    {
        this->sz = sz;
        for (int i=0; i<sz; i++)
          this->text[i] = c;
    }
}
```

...pero mejor evitarlas.

Un programador escribe esto...

```
class String { ... }
...
String str;

str.fill(16,' ')
l = str.length()
```

Mientras tanto, a bajo nivel...

```
class String { ... }
...
String str;

String_fill_v2ic(&str,16,' ');
l = String_length_i0(&str);
```

Se conoce como **name mangling**.

Los **atributos** son equivalentes a las componentes del registro. Cada instancia de la clase contiene dichos atributos con sus tipos correspondientes.

```
class Complex
{
    float _real, _imag;
    ...
}
```

Los **atributos** tienen modo de acceso:

- **private**: No se puede acceder al atributo salvo desde la propia definición de la clase (**ámbito de clase**).
- **public**: Se puede acceder al atributo desde cualquier parte.

```
class Complex
{
private:
    float _real, _imag;
    ...
}
```

Los **métodos** son equivalentes a las funciones que trabajan sobre la propia clase.

```
float real()
{ return _real; }

float imag()
{ return _imag; }

float modulus()
{ return sqrt(_real*_real + _imag*_imag); }
```

Los métodos también tienen modo de acceso:

```
class Complex
{
private:
    float _real, _imag;

public:
    float real() { return _real; }
    float imag() { return _imag; }
    float modulus() { return sqrt(_real*_real+_imag*_imag); }
};
...
cout << c.modulus() << endl;
```

Los métodos también tienen modificadores:

```
class Complex
{
    float modulus()
    {
      return sqrt(_real*_real+_imag*_imag);
    }
};

...
void write(Complex& c)
{
    cout << c.modulus() << endl;
}
```

Pregunta:

¿ Por qué ?

What If ... ?

```
class Complex
{
    float modulus()
    {
      _real = 0; // Bug :)
      return sqrt(_real*_real+_imag*_imag);
    }
};
...
void write(Complex& c)
{
    cout << c.modulus() << endl; // Bug !!!
}
```

Los métodos también tienen modificadores:

```
class Complex
{
    float modulus()
    {
      _real = 0;
      return sqrt(_real*_real+_imag*_imag);
    }
};
...
void write(const Complex& c)
{
    cout << c.modulus() << endl; // Error en compilación !!!
}
```

Los métodos también tienen modificadores:

```
class Complex
{
    float modulus() const
    {
      _real = 0; // Error en compilación !!!
      return sqrt(_real*_real+_imag*_imag);
    }
};
...
void write(const Complex& c)
{
    cout << c.modulus() << endl; // OK
}
```

TAD Implementación en línea (*inline*).

complex.h

```
// Interfaz e implementación
struct Complex
{
  float re,im;
}

float real(const Complex& c)
{ return c.re; }

float imag(const Complex& c)
{ return c.im; }

float modulus(const Complex& c)
{ return sqrt(c.re*c.re + c.im*c.im); }
```

TAD Implementación separada o fuera de línea (*out–of–line*).

complex.h

```
// Interfaz
struct Complex
{
  float re,im;
}

float real(const Complex&);
float imag(const Complex&);
float modulus(const Complex&);
```

complex.cc

```
// Implementacion
#include "complex.h"

float real(const Complex& c)
{ return c.re; }

float imag(const Complex& c)
{ return c.im; }

float modulus(const Complex& c)
{ return sqrt(c.re*c.re + c.im*c.im); }
```

Clase Implementación en línea (*inline*).

complex.h

```
// Interfaz e implementación
class Complex
{
  float re,im;

  float real() const
  { return re; }

  float imag() const
  { return im; }

  float modulus() const
  { return sqrt(re*re + im*im); }
}
```

Clase Implementación separada o fuera de línea (*out–of–line*).

complex.h

```
// Interfaz
class Complex
{
  float re,im;

  float real() const;

  float imag() const;

  float modulus() const;
}
```

complex.cc

```
// Implementacion
#include "complex.h"

float Complex::real() const
{ return re; }

float Complex::imag() const
{ return im; }

float Complex::modulus() const
{ return sqrt(re*re + im*im); }
```

Existe un método especial: el **constructor**.
Define las operaciones que se realizan sobre un objeto cuando se construye.

```
class Complex
{
    ...
    Complex( ) {
        _real = _imag = 0.0;
    }
}

Complex c; // Llama automáticamente a Complex( )
```

Pregunta:
¿ Por qué ese nombre de función ?

Existe un método especial: el **constructor**.

El constructor puede tener parámetros.

```
class Complex
{
    ...
    Complex(float r, float i = 0.0f) {
        _real = r;
        _imag = i;
    }
}

Complex p(1,2);
Complex q(3);
```

Pueden existir múltiples constructores:

```
class Complex
{
    float _real, _imag;
    Complex( ) {
        _real = _imag = 0.0;
    }
    Complex(float r, float i = 0.0f) {
        _real = r;
        _imag = i;
    }
};
```

Invocación del constructor (implícita):

- Variables **automáticas**:

```
Complex zero;             // -> Complex()
Complex unit(1.0);        // -> Complex(1.0,0.0)
Complex iunit(0.0,1.0);   // -> Complex(0.0,1.0)
```

- Con **memoria dinámica**:

```
Complex* zero;
Complex* unit;
Complex* iunit;
zero  = new Complex;            // -> Complex()
unit  = new Complex(1.0);       // -> Complex(1.0,0.0)
iunit = new Complex(0.0,1.0);   // -> Complex(0.0,1.0)
```

¿ Qué ocurre en este caso ?

```
class Complex
{
    float _real{}, _imag{};

    Complex(float r, float i = 0.0f)
    {
        _real = r;
        _imag = i;
    }
}
```

Notación: invocación de constructores de atributos

```
class Complex
{
    float _real, _imag;

    Complex(float r, float i = 0.0f)
        : _real(r), _imag(i)
    {
    }
}
```

Pueden existir múltiples constructores:

```
class Complex
{
    float _real, _imag;
    Complex( )
        : _real{}, _imag{}
    {}
    Complex(float r, float i = 0.0f)
        : _real(r), _imag(i)
    {}
};
```

Existe un método especial: el **destructor**.
Define las operaciones que se realizan sobre un objeto cuando se elimina.

```
class Store
{
  int* data;

  Store(int sz)
  { data = new int[sz]; }
};
```

```
void f()
{
  Store s(128);
  ...
}
```

Pregunta:
¿ Problema ?

Existe un método especial: el **destructor**.
Define las operaciones que se realizan sobre un objeto cuando se elimina.

```
class Store
{
  int* data;

  Store(int sz)
  { data = new int[sz]; }

  ~Store()
  { delete[] data; }
};
```

```
void f()
{
  Store s(128);
  ...
}
```

Respuesta:
Destructor ¡¡ OK !!

Sólo puede existir un destructor, de invocación implícita:

- Variables **automáticas**: cuando termina su ámbito

```
{
  Store s(128)
  ...
} // -> ~Store()
```

- Con **memoria dinámica**: `delete`

```
Store* s = new Store(128);
...
delete s; // -> ~Store()
```

Invocación de métodos:

- Variables **automáticas**:

```
Complex iunit(0.0,1.0);
...
cout <<  iunit.modulus( ) << endl;
```

Invocación de métodos:

- Con **memoria dinámica**:

```
Complex* iunit;
...
iunit = new Complex(0.0,1.0);
...
// Válido, pero inusual
cout <<  (*iunit).modulus() << endl;
// Más correcto idiomáticamente
cout <<  iunit->modulus() << endl;
```

Los lenguajes modernos (C++, ADA) equiparan los operadores y las funciones.

Un operador binario como la suma

```
a + b // a y b de tipo Complex
```

puede interpretarse para cualquier tipo como una función

```
operator+(a,b)
```

pero con clases, también como un método

```
a.operator+(b)
```

Un operador binario como la suma

```
a + b // interpretado como operator+(a,b)
```

puede definirse como una función externa a la clase:

```
class Complex { ... };

Complex operator+(const Complex& a, const Complex& b)
{
  // a + b
}
```

Un operador binario como la suma

```
a + b // interpretado como a.operator+(b)
```

puede definirse como un método:

```
class Complex
{
    Complex operator+(const Complex& b) const
    {
      // this + b
    }
};
```

Ejemplo:

```
class Complex {
  Complex operator+(const Complex& c) const
  {
     return Complex(_real+c._real,_imag+c._imag);
  }

  Complex& operator*=(float f)
  {
    _real*=f; _imag*=f;
    return *this;
  }
}
```

Nota: `c = a + (b*=2);`

A veces sólo es posible mediante funciones externas:

```
Complex c;
// interpretado como operator<<(ostream&,Complex)
cout << c;
...
ostream& operator<<(ostream& os, const Complex& c)
{
  os << c.real() << "+" << c.imag() << "i";
  return os;
}
```

Nota: `cout << a << b << c;`

Atributos de clase

En C/C++ los TADs pueden usar variables globales:

process.h

```
1  extern int pcount;
2
3  struct Process
4  {
5    int pid;
6  };
7
8  void init(Process& p);
```

process.cc

```
1  #include "process.h"
2
3  static int next_pid = 1000;
4  int pcount = 0;
5
6  void init(Process& p) {
7    p.pid = next_pid++;
8    pcount++;
9  }
```

Atributos de clase

En C++ también existen los **atributos de clase**, compartidos por todos los objetos de una clase, y sujetos a las reglas de privacidad:

process.h

```
class Process
{
private:
  static int next_pid;
  static int pcount;
  int pid;

public:
  static int processCount();

  Process();
  void info();
};
```

process.cc

```
#include "process.h"

int Process::next_pid = 1000;
int Process::pcount = 0;

int Process::processCount()
  { return pcount; }

Process::Process() {
    pid = next_pid++;
  pcount++;
}

void Process::info()
  { cout << "pid " << pid << endl; }
```

Compartidos por todos los objetos de una clase:

process.h

```
class Process
{
private:
  static int next_pid;
  static int pcount;
  int pid;

public:
  static int processCount();

  Process();
  void info();
};
```

main.cc

```
#include "process.h"
#include <iostream>
using namespace std;
int main()
{
  cout << Process::processCount() << endl;
  Process* p[10];
  for (int i=0; i<10; i++) {
    p[i] = new Process();
    p[i]->info();
  }
  cout << Process::processCount() << endl;
  for (int i=0; i<10; i++) delete p[i];
  return 0;
}
```

```java
class Process
{
  private static int next_pid;
  private static int pcount;
  static {
    next_pid = 1000; pcount = 0;
  }
  static int processCount() {
    return pcount;
  }
  private int pid;
  Process() {
    pid = next_pid++; pcount++;
  }
  void info() {
    System.out.println("pid " + pid);
  }
}
```

```java
class Process
{
  private static int next_pid;
  private static int pcount;
  static {
    next_pid = 1000; pcount = 0;
  }
  static int processCount() {
    return pcount;
  }
  private int pid;
  Process() {
    pid = next_pid++; pcount++;
  }
  void info() {
    System.out.println("pid " + pid);
  }
}
```

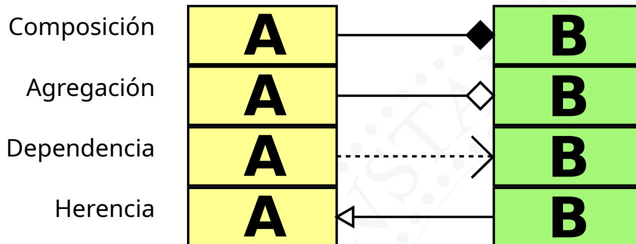
```java
class main
{
  public static void main(String[] args)
  {
    Process[] p = new Process[10];
    for (int i=0; i<10; i++) {
      p[i] = new Process();
      p[i].info();
    }
    System.out.println(Process.processCount());
  }
}
```

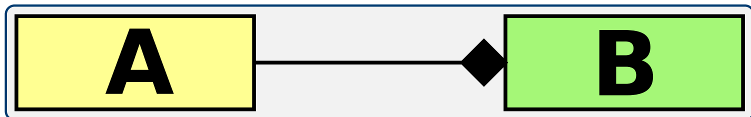
Se puede definir código común a todos los constructores:

```
1   class Process
2   {
3     private static int next_pid;
4     private static int pcount;
5     static {
6       next_pid = 1000;  pcount = 0;
7     }
8     private int pid; {   //Common constructor
9       pid = next_pid++; pcount++;
10    }
11    static int processCount() {
12      return pcount;
13    }
14    void info() {
15      System.out.println("pid " + pid);
16    }
17  }
```

Relaciones entre clases

Diferentes tipos de asociación:





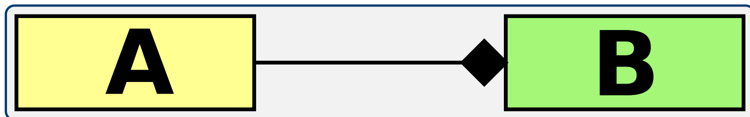
Ambos objetos comparten tiempo de vida.

```
class B { ... };
class A {
   B b;
   ...
};
```

```
class Fecha { int dia, mes, anyo; };
class Hora  { int hora, minuto, segundo; };

class MarcaTiempo {
    Fecha fecha;
    Hora  hora;

    MarcaTiempo(...) :
        fecha(...), hora(...) {}
}
```



Ambos objetos comparten tiempo de vida (con punteros).

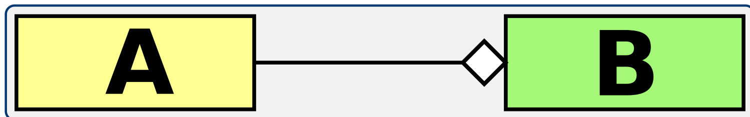
```
class B { ... };
class A {
   B* b;
   A() { b = new B(); }
   ~A() { delete b; }
   ...
};
```

```
class Fecha { int dia, mes, anyo; };
class Hora  { int hora, minuto, segundo; };

class MarcaTiempo {
    Fecha* fecha;
    Hora* hora;

    MarcaTiempo(...) : fecha(new Fecha(...)), hora(new Hora(...))
    {}

    ~MarcaTiempo()
    { delete fecha; delete hora; }
}
```



Objetos referenciados con tiempos de vida independientes.

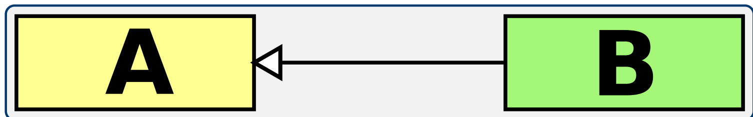
```
class B { ... };
class A {
   B* b;
   ...
};
```

Pregunta:
Esto es **peligroso**. ¿ Por qué ?

```
class Jugador;
class Equipo {
    string nombre, cif;
    Jugador* jugadores[11];
};
class Jugador {
    string nombre;
    int dorsal;
    Equipo* equipo;
};
```

Un objeto utiliza otro (sin otro tipo de relación)

```
class B { ... };
class A {
   void metodo(const B& b) { ... }
   ...
};
```

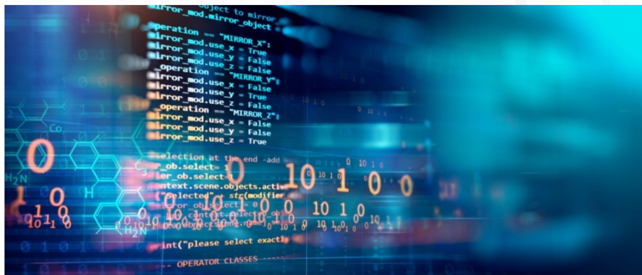


Una es subclase de otra (próximo tema)

```
class B { ... };
class A : public B {
    ...
};
```

Clases

Tecnología de Programación



Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-01-28