Paradigmas de Programación

3.1. Orientación a Objetos

Departamento de Informática
Universidad de Valladolid

Curso 2020-21

Grado en Ingeniería Informática
Grado en Ingeniería Informática de Sistemas

CONCEPTOS FUNDAMENTALES

Prog. Orientada a Objetos

Es un paradigma de programación que afecta a distintos niveles del desarrollo del software:

- El análisis y diseño del software.
- El modo de organización de los programas.
- El sistema de tipado.
- La filosofía de la programación.

Un lenguaje de programación es orientado a objetos si incorpora elementos sintácticos que permitan y faciliten el uso del paradigma.

Conceptos Clave

- Clase: Un tipo abstracto de datos junto con una implementación (posiblemente parcial).
  - Pertenece a la parte estática (definiciones y declaraciones), definida antes de la compilación. Una clase puede imaginarse como el tipo de un objeto.
  - Facilitan la estructuración y jerarquización.
- Objeto: Una instancia de una clase. Encapsula estado (atributos) y comportamiento (métodos)
  - Pertenece a la parte dinámica: Se pueden crear y destruir en tiempo de ejecución. Los objetos de la misma clase pueden tener estados distintos, pero comparten la misma definición de operaciones.
  - Interaccionan con otros objetos mediante paso de mensajes (invocación a métodos)

Diseño Orientado a Objetos

- El paradigma modular hace énfasis en la descomposición funcional: Dividir la aplicación en un grupo de tareas relacionadas.
  - La ejecución de un programa consiste en llamadas a subrutinas que se pasan datos unas a otras.
  - Las modificaciones o ampliaciones de tipos de datos afectan a muchas partes de la aplicación.
- La orientación a objeto promueve una descomposición basada en objetos: Dividir la aplicación en los elementos del dominio del problema:
  - La ejecución consiste en una colección de objetos, de distintos tipos (clases), que interactuan entre sí mediante paso de mensajes.
  - Se encapsulan datos con las operaciones que les afectan.

Objetivos de la P.O.O.

- **Reutilización**: Mediante el mecanismo de herencia.
- **Modularidad**: Las clases establecen un nuevo nivel de estructuración de programas.
- **Abstracción**: Representación de tipos abstractos de datos mediante clases.
- **Extendibilidad**: Definición de nuevas clases, separación interfaz-implementación
- **Genericidad**: Jerarquía de clases, polimorfismo, ligadura dinámica.

Técnicas/requisitos de la P.O.O.

- **Clases y Objetos:** Asociación de datos junto con los métodos que actuan sobre ellos en una única entidad.
- **Encapsulación:** La capacidad de establecer la visibilidad de componentes de los objetos para entidades externas.
- **Herencia:** La capacidad de definir clases que amplien o modifiquen la funcionalidad de otras clases.
- **Interfaz:** La posibilidad de definir métodos sin proporcionar una implementación para ellos.
- **Polimorfismo:** La capacidad de representar un objeto como perteneciente a una clase más básica que la suya.
- **Ligadura dinámica**: La garantía de ejecución del método asociado a un objeto independientemente de la definición de clase de la variable que lo representa.

Otras técnicas O.O.P.

- **Autoreferencia:** La posibilidad de acceder a una referencia al propio objeto en la definición de sus métodos. Tipicamente se implementa mediante un atributo predefinido (*this*, *self*). Existen lenguajes (ej. Python) donde es el único medio para acceder a los atributos (ámbito local) del objeto.
- **Métodos estáticos:** La posibilidad de definir métodos asociados a la clase en lugar de al objeto. Estos métodos pueden invocarse sin necesidad de instanciar un objeto, y no pueden acceder a atributos (no estáticos). En algunos lenguajes (Delphi, Python, C++) el constructor de objetos es una función estática.
- **Metaclases:** La capacidad de trabajar con referencias a clases (no a objetos). Uso: Genericidad.
- **Reflexion:** La posibilidad de obtener y usar información sobre la clase a la que pertenece un objeto.

VISIÓN GENERAL

Estructura estática: Clases

- Una clase consiste en la definición de un tipo abstracto de datos mas la (posible) implementación de las operaciones.
  - Si una clase no implementa todas las operaciones se dice que es una clase abstracta (o diferida).
- Los objetos son instancias (materializaciones) de clases.
- Las clases definen:
  - Atributos: Datos que definen el estado del objeto
  - Métodos: Código encargado de modelar el comportamiento de los objetos: Acceso y modificación de los atributos, respuesta a mensajes de otros objetos.
- Permiten la creación de nuevos TADs con un mayor nivel de abstracción que en el paradigma modular.

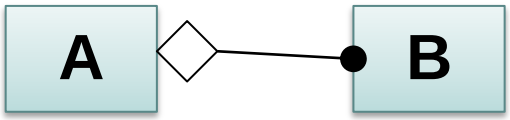
Clases: Encapsulamiento

- Las clases permiten definir niveles de acceso y/o visibilidad a sus atributos y métodos:
  - private: Sólo accesible por la propia clase
  - protected: Sólo accesible por la propia clase y las clases que deriven de ella.
  - public: Accesible a todas las clases.
  - Existen lenguajes (Eiffel) donde es posible indicar las clases concretas que tienen acceso.
- La filosofía O.O. no permite el acceso/modificación directo de los atributos, debe realizarse usando métodos (código) que actuen de intermediarios
  - El objetivo es garantizar la corrección del estado de un objeto y aislar su interfaz de la implementación (poder modificar sus atributos sin afectar a los clientes de la clase)

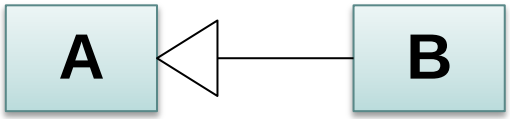
Clases: Herencia

Relaciones entre clases:

- Agregación: Los objetos de clase A contienen objetos de la clase B.



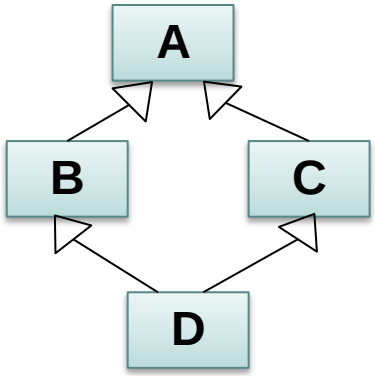
- **Herencia**: La clase B hereda o se deriva de la clase A



  - La clase B es una especialización de la clase A
  - Un objeto de clase B es un objeto de tipo A, posiblemente con añadidos y modificaciones.

Herencia Múltiple

- **Herencia múltiple**: Una clase puede derivarse (heredar) de varias clases base. Jerarquía de tipo grafo.
- Problemas: (C++, Eiffel)
  - Es posible que las clases base tengan métodos con el mismo nombre. Se necesita un mecanismo de redefinición y selección de métodos.
  - El problema del diamante:

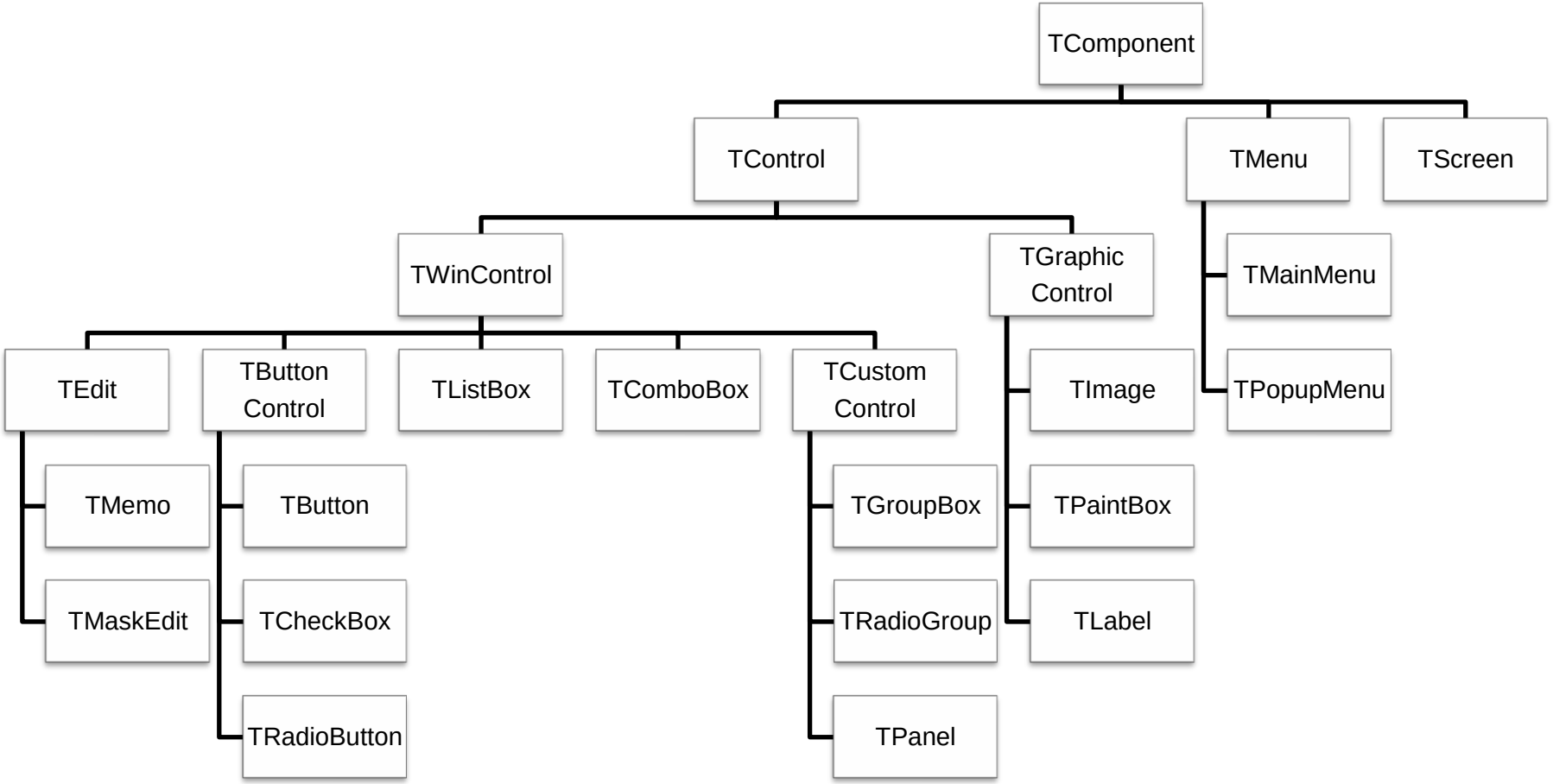


- Algunos lenguajes (Java, C#) permiten herencia simple entre clases y múltiple mediante interfaces.

Herencia – Sistema de tipado

- La relación de herencia permite establecer una jerarquía de subtipos.
- **Polimorfismo**: Toda clase derivada se considera un subtipo de su clase base: Sus valores (objetos) pueden usarse (conversiones implícitas) como valores de la clase base
- Clase Universal: Aquella de la que heredan el resto de clases (TObject, Object, ANY, ...)
- Interface: Clase totalmente abstracta, sólo métodos abstractos, sin atributos. (Java, C#)
- **Herencia simple**: Toda clase hereda de una única clase. Relación tipo arbol. (Delphi)

Ejemplo de Jerarquía (GUI-Delphi)



Estructura dinámica: Objetos

- Un objeto es una instancia (materialización) de una clase. Los objetos se crean en tiempo de ejecución.
- Mecanismos de creación: Funciones constructoras (Delphi, Python) y/o operadores (new en C++, Java, C#)
- Las variables suelen almacenar referencias a objetos.
- Polimorfismo: Las variables de tipo clase A pueden almacenar referencias a objetos de cualquier tipo derivado de A.
  - No existe conversiones (implícitas o explícitas) de objetos de clase A a objetos de clase B o viceversa.
  - Lo que se convierte es una referencia a objeto clase B a una referencia a objeto clase A (el objeto no cambia, sólo la información que tiene de él el sistema de tipos).

Objetos – Invocación a métodos

- La operación básica entre objetos consiste en el envío de mensajes a otros objetos para pedir la invocación de un método.
- Algunos lenguajes tienen una sintaxis especial para reflejar este hecho (Smalltalk, Objective-C):

```
Objeto mensaje: parámetros
```

- Las gran mayoría, sin embargo, utiliza la sintaxis de llamada a subrutinas junto con el operador de acceso a campos de registros:

```
Objeto.metodo(parámetros)
```

VISIÓN DETALLADA

Definición de métodos - ámbito

- En el código de un método, su **ambito local** (elementos – datos y código – que son accesibles) está formado por sus parámetros, las variables locales y:
  - Una variable/parámetro predefinido (this en C++/Java/C#, self en Delphi/Python) que representa al objeto sobre el que se ha invocado al método.
  - Los atributos y métodos de la clase (todos) y las clases derivadas (salvo los privados). El acceso es directo o via la variable que representa el objeto (Python, JavaScript).
  - Suele existir un mecanismo para acceder a la versión de la clase base de aquellos métodos que han sido redefinidos en la clase (objeto super en Java, marca inherited en Delphi, función Super en Python, modificadores de ámbito en C++)

Tipos de métodos

- **Finales**: Métodos que no se pueden redefinir en clases derivadas (marca final en Java, por defecto C++/Delphi)
- **Virtuales:** Métodos que se pueden redefinir en clases derivadas (marca virtual en C++/Delphi, por defecto en Java)
- **Abstractos**: Métodos para los que la clase no proporciona implementación (la idea es que lo implementen las subclases). Marca abstract en Delphi/Java, asignación a 0 en C++.
- **Estáticos:** Métodos asociados a la clase, no a los objetos (marca static en Java/C++, class en Delphi)
- **Constructores / Destructores:** Encargados de inicializar y liberar recursos, respectivamente, del objeto.

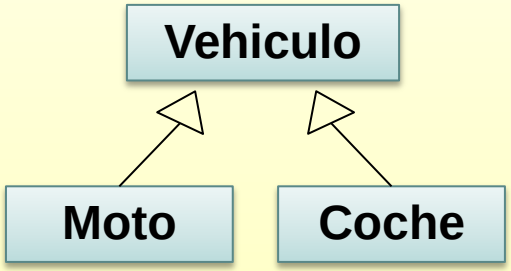
Métodos Estáticos

- No pueden acceder a atributos ni invocar métodos no estáticos (no existe ningún objeto definido sobre el que actuar).
- La invocación usa la sintaxis: **clase.metodo(params)**
- Algunos lenguajes (Java, C++) permiten también atributos estáticos (si pueden accederse por métodos estáticos).
- **Utilidad:**
  - Punto de inicio de programas
  - Enlace con librerías no orientadas a objeto
  - Métodos generales, no asociados a clases en particular
  - Patrón factoria de objetos

Herencia, Polimorfismo, Ligadura Dinámica

- Supongamos que la clase B deriva (hereda) de A.
- Sea **x** una variable de clase A e ***y*** una variable de clase B
- **x** puede referenciar a objetos de clase A y *de todas las clases que deriven de A* (objetos de clase B entre ellos)
- **y** no puede referenciar a objetos de tipo clase A.
- Si hacemos **x = objeto de clase B** no se produce ningún tipo de conversión. Lo único que cambia es la información de que dispone el compilador sobre lo que almacena **x** (una referencia a un objeto de clase A o derivada)
- Cuando se invoca a un método de un objeto, se usa la definición de la clase del propio objeto, no la de la clase a la que pertenece la variable.

Polimorfismo




```
Vehiculo v = new Vehiculo();
Moto     m = new Moto();
Coche    c = new Coche();

Vehiculo vm = m; // Ok. Una moto es un vehículo
Vehiculo vc = c; // Ok. Un coche es un vehículo
Moto     mc = c; // Error compilación. Un coche no es una moto
Coche    cv = v; // Error compilación. Un vehículo puede no ser coche
cv = vc; // Error compilación. El compilador no sabe que vc almacena
         // un coche, sólo sabe que es un vehículo.
cv = (Coche) vc; // Ok. El programador pide al compilador que asuma
                 // que vc contiene un coche, y es cierto.
cv = (Coche) vm; // Compila, pero da error en ejecución porque vm
                 // contiene una moto, no un coche.
```

Ligadura dinámica

```
public class A {
  public A() { }
  public String test() { return “Soy de clase A”; }
}
public class B extends A {
  public B() { }
  // redefine test respecto a A
  public String test() { return “Soy de clase B”; }
}
...
A x,y; // Las variables x e y son de clase A
B z;   // La variable z es de clase B
x = new A(); // La variable x referencia a objeto clase A
z = new B(); // La variable z referencia a objeto clase B
y = z;       // La variable y referencia a objeto clase B
System.out.println(x.test());       // Escribe “Soy de clase A”
System.out.println(y.test());       // Escribe “Soy de clase B”
System.out.println(z.test());       // Escribe “Soy de clase B”
System.out.println(((A) z).test()); // Escribe “Soy de clase B”
```

Gestión de Memoria

- **Gestión Manual**: El programador debe encargarse de destruir los objetos al finalizar su uso. Delphi, C++.
  - Problema de “agujeros de memoria” al no destruirlos.
  - Problema de acceso a objeto ya destruido (cuando la referencia a un objeto se comparte en varios puntos)
  - Problema general: Delimitar quién y cuando se encarga de la destrucción de objetos compartidos
- **Gestión Automática (Garbaje collection)**: El entorno tiene un proceso encargado de detectar cuando un objeto no es accesible y destruirle. Java, C#, Python.
  - Técnica iniciada por Lisp, común en lenguajes funcionales.
  - No trivial: Problema de las referencias circulares.
  - Variantes: Reference counting, mark-sweep, etc.

Grados de pureza en O.O.

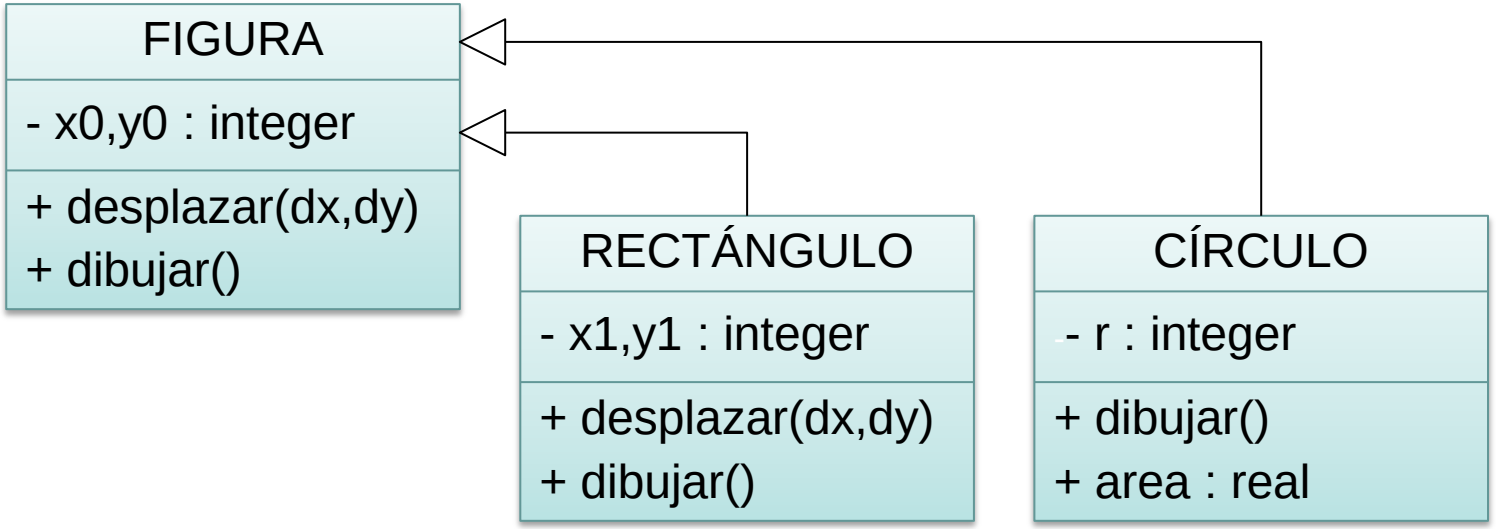
- Existencia de mecanismo sintáctico de paso de mensajes
- Todo tipo es una clase.
- No existe ámbito global.
- Toda subrutina es un método.
- Todo operador es un método.
- Todo módulo es una clase.
- Todo objeto puede sustituirse por uno equivalente de una clase derivada (*Liskov substitution principle*)
- Herencia, Encapsulamiento, Ligadura dinámica.

Pureza en O.O.

	Delphi	C++	Java	C#	Python	Smalltalk	Obj.-C
Invocación a Métodos	Llamada	Llamada	Llamada	Llamada	Llamada	Paso de mensajes	Paso de mensajes
Tipo → Clase	No	No	Parcial autoboxing	Parcial autoboxing	Todo valor es objeto	Si	No
Ámbito Global	Si	Si	No	No	Si	No	Si
Subrutina → Método	No	No	Si	Si	No	Si	No
Operador → Método	No	No	No	No	No	Si	No
Módulo → Clase	No	No	Si	Si	No	Si	Si
Herencia	Simple	Múltiple	Simple C Múltiple I	Simple C Múltiple I	Múltiple	Simple	Simple

Un problema, 3 soluciones

- **Problema**: Un programa de dibujo vectorial, donde queremos representar varias primitivas gráficas (lineas, rectángulos, círculos, etc.), cada una de ellas con distinta información y varias operaciones comunes (desplazar, dibujar, etc.) y algunas específicas. Se desea una representación lo más uniforme posible.



Delphi (I)

```
unit uFiguras;
interface
type
  TFigura = class
  protected
    x0,y0 : integer;
  public
    constructor Create(x0,y0: integer);
    procedure Desplazar(dx,dy: integer); virtual;
    procedure Dibujar; virtual; abstract;
  end;
  TRectangulo = class(TFigura)
  protected
    x1,y1 : integer;
  public
    constructor Create(x0,y0,x1,y1: integer); overload;
    procedure Desplazar(dx,dy: integer); override;
    procedure Dibujar; override;
  end;
```

Visibilidad

Atributos

Métodos

Herencia

Polimorfismo,
Ligadura dinámica

Delphi (II)

```
  TCirculo = class(TFigura)
  protected
    r : integer;
  public
    constructor Create(x0,y0,r: integer); overload;
    procedure Dibujar; override;
    function Area : double;
  end;
implementation
  { -- TFigura -- }
  constructor TFigura.Create(x0,y0: integer);
  begin { La llamda a un constructor devuelve un objeto }
    self.x0 := x0; { Se usa self para evitar el enmascaramiento }
    self.y0 := y0;
  end;
  procedure TFigura.Desplazar(dx,dy: integer);
  begin
    x0 := x0 + dx;
    y0 := y0 + dy;
  end;
```

Delphi (III)

```
{ -- TRectangulo -- }
constructor TRectangulo.Create(x0, y0, x1, y1: integer);
begin
  inherited Create(x0,y0); { Invoca al constructor de la clase base }
  self.x1 := x1;
  self.y1 := y1;
end;

procedure TRectangulo.Desplazar(dx, dy: integer);
{ Desplazar un rectángulo es desplazar su figura mas actualizar las
  coordenadas extra x1,y1 }
begin
  inherited; { Invoca a Desplazar tal como se define en TFigura }
  x1 := x1 + dx;
  y1 := y1 + dy;
end;

procedure TRectangulo.Dibujar; { No se define en TFigura }
begin
  writeln('Rectangulo en (',x0,',',y0,') - (',x1,',',y1,')');
end;
```

Delphi (IV)


```
  { -- TCirculo -- }
  constructor TCirculo.Create(x0, y0, r: integer);
  begin
    inherited Create(x0,y0);
    self.r := r;
  end;

  procedure TCirculo.Dibujar; { No se define en TFigura }
  begin
    writeln('Circulo en (',x0,',',y0,') con radio ',r);
  end;

  function TCirculo.Area: double; { Sólo existe en TCirculo }
  begin
    Area := Pi*r*r;
  end;

end. { módulo uFiguras }
```

Delphi (V)

```
Rectangulo en (1,1) - (100,100)
Rectangulo en (11,11) - (110,110)
Circulo en (50,50) con radio 10
Circulo en (60,60) con radio 10
Area del circulo: 314.16
```

```
program PruebaOO;
uses uFiguras;
var
  I : integer;
  Vec : array[1..2] of TFigura;
begin
  Vec[1] := TRectangulo.Create(1,1,100,100);
  Vec[2] := TCirculo.Create(50,50,10);
  for I := 1 to 2 do
  begin
    Vec[I].Dibujar;
    Vec[I].Desplazar(10,10);
    Vec[I].Dibujar;
  end;
  writeln('Area del circulo: ',(Vec[2] as TCirculo).Area:2:2);
  Vec[1].free;
  Vec[2].free;
end.
```

Variables de tipo TFigura pueden contener objetos de tipo TRectangulo y TCirculo (Polimorfismo)

Creación de objetos

Ligadura dinámica

Conversión explícita

Destrucción de objetos por el programador

Java (I)


```
interface Figura {
  public void desplazar(int dx, int dy);
  public void dibujar();
}
```

← "Clase" abstracta (TAD)

```
class Rectangulo extends Figura {
  private int x0,y0,x1,y1;
  public Rectangulo(int x0, int y0, int x1, int y1) {
    this.x0 = x0; this.y0 = y0;
    this.x1 = x1; this.y1 = y1;
  }
  public void desplazar(int dx, int dy) {
    x0 += dx; y0 += dy; x1 += dx; y1 += dy;
  }
  public void dibujar() {
    System.out.println("Rectangulo en ("+x0+","+y0+")-("+x1+","+y1+")");
  }
}
```

Java (II)

```
class Circulo extends Figura {
  private int x0,y0,r;
  public Circulo(int x0, int y0, int r) {
    this.x0 = x0; this.y0 = y0; this.r = r;
  }
  public void desplazar(int dx, int dy) {
    x0 += dx; y0 += dy;
  }
  public void dibujar() {
    System.out.println("Circulo en ("+x0+","+y0+") con radio"+r);
  }
 public double area() {
    return(Math.PI*r*r);
  }
}
```

Atributo estático de la clase Math

Java (III)


```
class Programa {
  public static void main(String[] args)
  {
    Figura[] vec = new Figura[2];
    vec[0] = new Rectangulo(1,1,100,100);
    vec[1] = new Circulo(50,50,10);
    for(int i = 0; i < 2; i++) {
      vec[i].dibujar();
      vec[i].desplazar(10,10);
      vec[i].dibujar();
    }
    System.out.println("Area del circulo: "+((Circulo) vec[1]).area());
  }
}
```

Creación de objetos
(los arrays son objetos)

Conversión explícita

Python (I)

```
class Figura(object):
  def __init__(self,x0,y0):
    self.x0 = x0;
    self.y0 = y0;
  def desplazar(self,dx,dy):
    self.x0 += dx;
    self.y0 += dy;
  def dibujar(self):
    raise NotImplementedError("dibujar() no definido!");

class Rectangulo(Figura):
  def __init__(self,x0,y0,x1,y1):
    super(Rectangulo,self).__init__(x0,y0);
    self.x1 = x1;
    self.y1 = y1;
  def dibujar(self):
    print 'Rectangulo en (%d,%d)-(%d,%d)' % \
          (self.x0,self.y0,self.x1,self.y1);
```

Python (II)


```
 def desplazar(self,dx,dy):
    super(Rectangulo,self).desplazar(dx,dy);
    self.x0 += dx;
    self.y0 += dy;


class Circulo(Figura):
  def __init__(self,x0,y0,r):
    super(Circulo,self).__init__(x0,y0);
    self.r = r;
  def dibujar(self):
    print 'Circulo en (%d,%d) con radio %d' % \
          (self.x0,self.y0,self.r);
  def area(self):
    return math.pi*self.r*self.r;
```

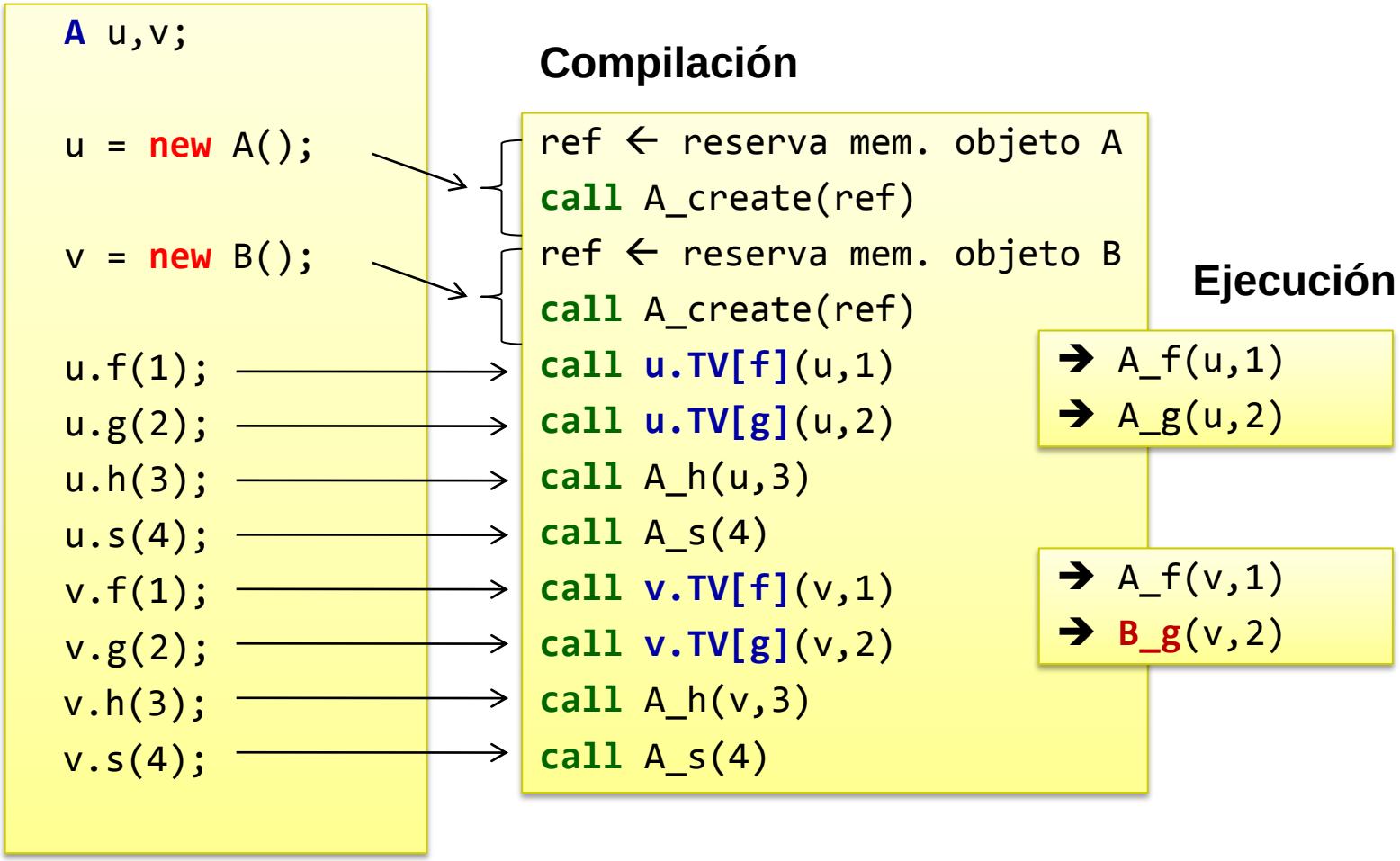
Python (III)


```
# Programa principal
import math;

vec = [];
vec.append(Rectangulo(1,1,100,100));
vec.append(Circulo(50,50,10));
for fig in vec:
  fig.dibujar();
  fig.desplazar(10,10);
  fig.dibujar();

# Tipado “del pato”
print ‘Area del circulo: %f’ % (vec[1].area());
```

Como funciona.. (1)


```
class A {
    int x;
    A() {...} // Constructor
    void f(int n) {...} // Virtual
    void g(int n) {...} // Virtual
    final void h(int n) {...} // Final
    static void s(int n) {...} // Estático
}
class B extends A {
    int y;
    @override void g(int n) {...} // Redefinido
}
```

Como funciona.. (2)



Como funciona.. (3)