UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y CC. Computación
E.T.S.I. Telecomunicación

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

Tema 4

Programación II

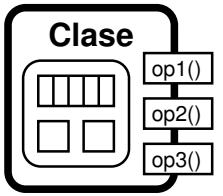
T4-Anexo: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

INTRODUCCIÓN

- La Programación Orientada a Objetos es un paradigma de programación que nos permite diseñar programas definiendo **abstracciones** que modelan los **datos** que representan el problema que queremos resolver.

- La Programación Orientada a Objetos extiende los conceptos fundamentales de **abstracción** y **encapsulación** de los Tipos Abstractos de Datos (véase Tema 2), añadiendo los siguientes conceptos:

  - Herencia, Polimorfismo y Vinculación Dinámica



- Una Clase define una abstracción, y los métodos definen su comportamiento.

- Objetos: entidades activas que encapsulan datos y algoritmos.

  - Atributos (datos): contienen el estado y la representación interna del objeto, cuyo acceso está restringido.
  - Métodos (algoritmos): permiten la manipulación e interacción entre objetos. Definen el comportamiento del objeto.

- Proporciona un mecanismo adecuado para el diseño y desarrollo de software complejo, modular, reusable, adaptable y extensible.

- Estudiaremos la **POO** con un enfoque reducido para disminuir su complejidad.

INTRODUCCIÓN: Clase y Objeto

- Una **Clase** representa una abstracción de datos, especifica las características de unos **objetos**: su estado y su comportamiento.
- Un **objeto** es una *instancia* de una determinada *Clase*.
- Las características del *objeto* (estado y comportamiento) están determinadas por la *Clase* a la que pertenece.
- Puede haber muchos objetos **distintos** que sean de la misma *Clase* (y tambien de distintas Clases).
- Cada *objeto* almacena y contiene su propio estado interno (*atributos*), de forma *independiente* de los otros *objetos*.
- El objeto podrá ser manipulado e interactuar con otros objetos a través de los *métodos* definidos por la *Clase* a la que pertenece.

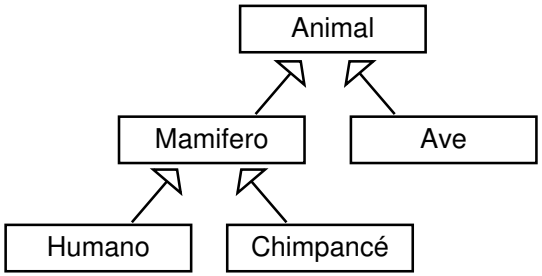
INTRODUCCIÓN: Métodos y Atributos

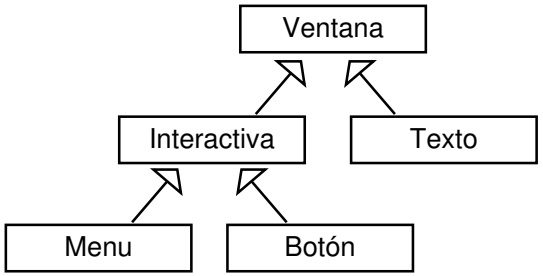
- La clase representa una abstracción de datos, y los métodos definen su comportamiento.
- Los **métodos** son algoritmos *especiales* definidos por la *Clase*, y se aplican sobre los objetos.
- Los métodos manipulan el estado interno del objeto sobre el que se aplican.
- Los métodos se invocan aplicándolos sobre un objeto.
- Los objetos responden a las invocaciones de los métodos dependiendo de su estado interno.
- La invocación a métodos puede llevar parámetros asociados, así como producir un resultado, además de manipular el estado interno del objeto sobre el que se aplica.
- Para invocar a un determinado método sobre un objeto, ese método debe estar definido por la clase a la que el objeto pertenece.
- Los **atributos** almacenan los valores del estado interno del objeto.
- Cada objeto tiene un estado interno asociado, independiente de los otros objetos.
- Los atributos estan protegidos. Sólo se permite su acceso y manipulación a través de los métodos.

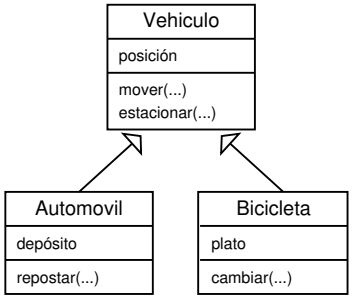
T4-Anexo: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

HERENCIA

- La **herencia** permite modelar relaciones **ES UN** entre clases, definiendo **jerarquías** de clases.
- Permite definir una nueva **clase derivada** (o sub-clase) como una especialización o extensión de una **clase base** (o super-clase) más general.
- La clase derivada **hereda** tanto los atributos como los métodos definidos por la clase base (reusabilidad del código).
- La clase derivada puede *definir nuevos* atributos y nuevos métodos (extensibilidad), así como *redefinir* métodos de la clase base.

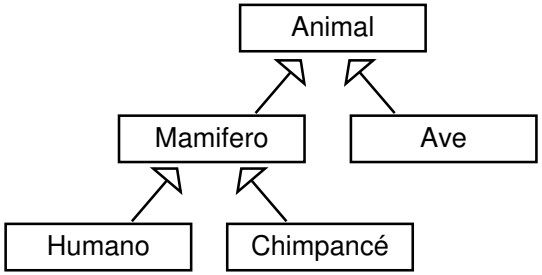


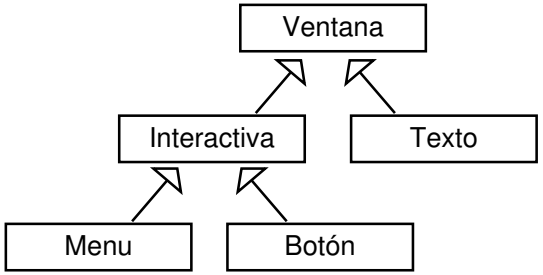


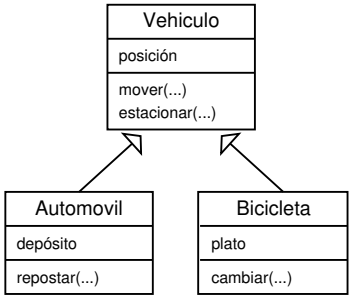


POLIMORFISMO Y VINCULACIÓN DINÁMICA

- El **polimorfismo** permite que un objeto de una **clase derivada** pueda ser considerado como un objeto de la **clase base**. *Principio de sustitución*.
  - La dirección de correspondencia opuesta **no** se mantiene, ya que **no** todos los objetos de la clase base son también objetos de la clase derivada.
- La **vinculación dinámica** permite que las clases derivadas puedan **redefinir** el comportamiento de los métodos definidos en la clase base.
  - En *contextos polimórficos*, los métodos invocados se seleccionan adecuadamente, en tiempo de ejecución, dependiendo del *tipo real* del objeto, y no del *tipo aparente*.







T4-Anexo: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

DEFINICIÓN E IMPLEMENTACIÓN DE CLASES POLIMÓRFICAS (I)

- C++ proporciona soporte adecuado tanto al paradigma de *Tipos Abstractos de Datos* (mediante **clases normales**), como al paradigma de *Programación Orientada a Objetos* (mediante **clases polimórficas**).

- Un nuevo ámbito de acceso restringido `protected` permite el acceso únicamente desde las clases derivadas.

- La clase derivada debe especificar, en su definición, que hereda de forma pública de la clase base.

- Los constructores de la clase derivada deben invocar al constructor de la clase base al principio de la lista de inicialización.

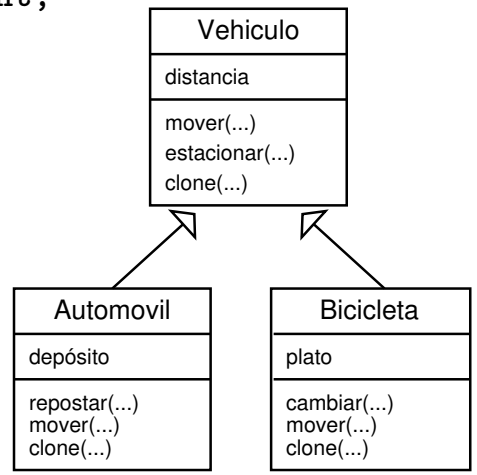
- Tanto el **destructor** como todos los **métodos públicos** deben ser especificados como **virtual** en la definición de la clase (**hpp**), pero no en su implementación (**cpp**).

- El operador de asignación no funciona adecuadamente en estos *contextos polimórficos*. Se debe desactivar y sustituir por un método `clone()`.

    - La implementación del método `clone()` debe devolver un puntero a un nuevo objeto de la clase creado como **copia** del objeto actual.

DEFINICIÓN E IMPLEMENTACIÓN DE CLASES POLIMÓRFICAS (II)

```
#ifndef vehiculo_hpp_
#define vehiculo_hpp_
#include <string>
namespace umalcc {
    class Vehiculo {
    public:
        Vehiculo(const std::string& i) ;
        virtual ~Vehiculo() ;
        virtual std::string id() const ;
        virtual void estacionar() ;
        virtual void mover() ;
        virtual Vehiculo* clone() const ;
    protected:
        Vehiculo(const Vehiculo& o) ;
        virtual void mover(int x) ;
    private:
        Vehiculo& operator=(const Vehiculo&) ;
        std::string ident;
        int distancia ;
    };
}
#endif
```

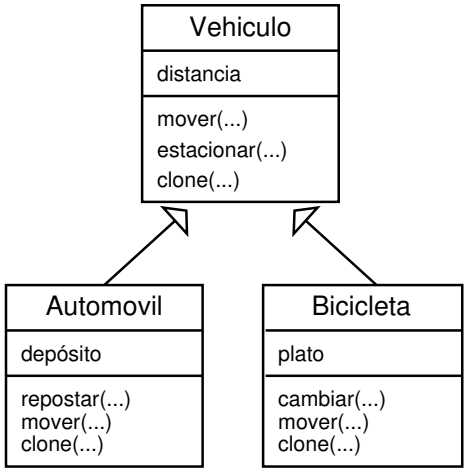



```
#include "vehiculo.hpp"
namespace umalcc {
    Vehiculo::~Vehiculo() {}
    Vehiculo::Vehiculo(const std::string& i)
        : ident(i), distancia(0) {}
    Vehiculo::Vehiculo(const Vehiculo& o)
        : ident(o.ident), distancia(o.distancia) {}
    std::string Vehiculo::id() const {
        return ident;
    }
    void Vehiculo::estacionar() {
        distancia = 0 ;
    }
    void Vehiculo::mover() {
        ++distancia ;
    }
    void Vehiculo::mover(int x) {
        distancia += x ;
    }
    Vehiculo* Vehiculo::clone() const {
        return new Vehiculo(*this) ;
    }
}
```

- El destructor y todos los métodos virtuales.
- El operador de asignación desactivado.
- El método `clone` devuelve una copia del objeto actual.
- Método `mover` protegido para que las clases derivadas puedan modificar el atributo `distancia`.

DEFINICIÓN E IMPLEMENTACIÓN DE CLASES POLIMÓRFICAS (III)

```
#ifndef automovil_hpp_
#define automovil_hpp_
#include "vehiculo.hpp"
#include <string>
namespace umalcc {
    class Automovil : public Vehiculo {
    public:
        Automovil(const std::string& i) ;
        virtual ~Automovil() ;
        virtual void mover() ;
        virtual void repostar(int litros) ;
        virtual Automovil* clone() const ;
    protected:
        Automovil(const Automovil& o) ;
    private:
        int deposito ;
    };
}
#endif
```

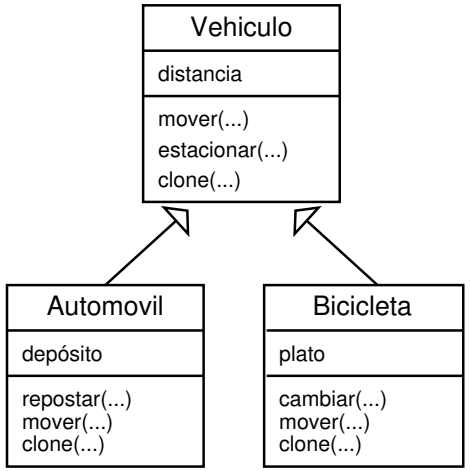



```
#include "automovil.hpp"
namespace umalcc {
    Automovil::~Automovil() {}
    Automovil::Automovil(const std::string& i)
        : Vehiculo(i), deposito(0) {}
    Automovil::Automovil(const Automovil& o)
        : Vehiculo(o), deposito(o.deposito) {}
    void Automovil::mover() {
        if (deposito > 0) {
            Vehiculo::mover(50) ;
            --deposito ;
        }
    }
    void Automovil::repostar(int litros) {
        deposito += litros ;
    }
    Automovil* Automovil::clone() const {
        return new Automovil(*this) ;
    }
}
```

- La clase `Automovil` hereda de la clase `Vehiculo`.
- Hereda los métodos `id` y `estacionar`, redefine los métodos `mover` y `clone`, y añade el nuevo método `repostar` (y el atributo `deposito`).
- La lista de inicialización del constructor de `Automovil` invoca al constructor de `Vehiculo`.
- El destructor de `Automovil` invoca automáticamente al destructor de `Vehiculo`.
- El método `mover` invoca al método protegido `mover` de la clase base (`Vehiculo::mover(...)`).

DEFINICIÓN E IMPLEMENTACIÓN DE CLASES POLIMÓRFICAS (VI)

```
#ifndef bicicleta_hpp_
#define bicicleta_hpp_
#include "vehiculo.hpp"
#include <string>
namespace umalcc {
    class Bicicleta : public Vehiculo {
    public:
        Bicicleta(const std::string& i) ;
        virtual ~Bicicleta() ;
        virtual void mover() ;
        virtual void cambiar() ;
        virtual Bicicleta* clone() const ;
    protected:
        Bicicleta(const Bicicleta& o) ;
    private:
        int plato ;
    };
}
#endif
```




```
#include "bicicleta.hpp"
namespace umalcc {
    Bicicleta::~Bicicleta() {}
    Bicicleta::Bicicleta(const std::string& i)
        : Vehiculo(i), plato(0) {}
    Bicicleta::Bicicleta(const Bicicleta& o)
        : Vehiculo(o), plato(o.plato) {}
    void Bicicleta::mover() {
        Vehiculo::mover(plato + 1) ;
    }
    void Bicicleta::cambiar() {
        plato = (plato + 1) % 3 ;
    }
    Bicicleta* Bicicleta::clone() const {
        return new Bicicleta(*this) ;
    }
}
```

- La clase `Bicicleta` hereda de la clase `Vehiculo`.
- Hereda los métodos `id` y `estacionar`, redefine los métodos `mover` y `clone`, y añade el nuevo método `cambiar` (y el atributo `plato`).
- La lista de inicialización del constructor de `Bicicleta` invoca al constructor de `Vehiculo`.
- El destructor de `Bicicleta` invoca automáticamente al destructor de `Vehiculo`.
- El método `mover` invoca al método protegido `mover` de la clase base (`Vehiculo::mover(...)`).

T4-Anexo: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

UTILIZACIÓN DE CLASES POLIMÓRFICAS (I)

- Para que el polimorfismo y la vinculación dinámica sean efectivas, se debe trabajar con los objetos a través de **punteros** (o de referencias).

  - El **polimorfismo** permite que un puntero a un objeto de una **clase derivada** pueda ser considerado como un puntero a un objeto de la **clase base**. *Principio de sustitución*.

```
Bicicleta* ptr_bicicleta = new Bicicleta("B123") ;
Vehiculo* ptr_vehiculo = ptr_bicicleta ;
```

  - La dirección de correspondencia opuesta **no** se mantiene, ya que **no** todos los punteros a objetos de la clase base son también punteros a objetos de la clase derivada.

  - En contextos polimórficos, sólo es válido invocar a los métodos especificados por el *tipo aparente* del puntero al objeto.

  - La **vinculación dinámica** permite, en *contextos polimórficos*, que los métodos **virtuales** invocados se seleccionen adecuadamente, en tiempo de ejecución, dependiendo del *tipo real* del objeto, y no del *tipo aparente*.

```
ptr_bicicleta->mover() ;   ptr_vehiculo->mover() ;   delete ptr_vehiculo ;
```

UTILIZACIÓN DE CLASES POLIMÓRFICAS (II)

```
// ... includes y using namespace
void proceso(Vehiculo* v)
{
    cout << "Estacionar " << v->id() << endl;
    v->estacionar() ;
}
int main()
{
    Bicicleta* bx = new Bicicleta("B001") ;
    proceso(bx) ;
    Vehiculo* b = bx ;
    Vehiculo* a = new Automovil("A002") ;
    proceso(a) ;
    a->mover() ;
    b->mover() ;
    bx->cambiar() ;
    // b->cambiar() ;    // No es posible
    // a->repostar(10) ; // No es posible
    Vehiculo* v = a->clone() ;
    v->mover() ;
    delete v ;
    delete a ;
    delete b ;
}
```

- El subprograma `proceso()` puede recibir como parámetro cualquier objeto de la clase `Vehiculo` o de sus clases derivadas (`Bicicleta` y `Automovil`) gracias al *polimorfismo*.
- Las invocaciones a los métodos `id()`, `estacionar()` y `mover()` son adecuadas gracias a la *vinculación dinámica*.
- La invocación al método `cambiar()` o `repostar()` sólo puede realizarse a través de punteros a la clase `Bicicleta` o `Automovil` (y derivadas).
- Un puntero a un objeto de la clase `Bicicleta` o de la clase `Automovil` puede ser asignado a un puntero de la clase `Vehiculo` gracias al *polimorfismo*.
- El método `clone()` realiza una duplicación y copia del objeto real gracias a la *vinculación dinámica*.
- La invocación a `delete` destruye cada objeto adecuadamente según su *tipo real* gracias a la *vinculación dinámica*.

T4-Anexo: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

EJEMPLO (I)

```
#ifndef parking_hpp_
#define parking_hpp_
#include "vehiculo.hpp"
#include <array>
#include <string>
namespace umalcc {
    class Parking {    // CLASE NO-POLIMORFICA
    public:
        ~Parking();
        Parking();
        Parking(const Parking& o);
        Parking& operator=(const Parking& o);
        void mostrar() const;
        void anyadir(Vehiculo* v, bool& ok);
        Vehiculo* extraer(const std::string& id);
    private:
        static const int MAX = 100;
        typedef std::array<Vehiculo*, MAX> Park;
        //--
        int buscar(const std::string& id) const;
        void destruir();
        void copiar(const Parking& o);
        //--
        int n_v;
        Park parking;
    };
}
#endif
```

EJEMPLO (II)

```
#include "parking.hpp"
#include <iostream>
using namespace std;
namespace umalcc {
  Parking::~Parking() { destruir(); }
  Parking::Parking() : n_v(0), parking() {}
  Parking::Parking(const Parking& o)
      : n_v(), parking() { copiar(o); }
  Parking& Parking::operator=(const Parking& o) {
      if (this != &o) {
          destruir();
          copiar(o);
      }
      return *this;
  }
  void Parking::anyadir(Vehiculo* v, bool& ok) {
      ok = ((v != NULL)&&(n_v < MAX));
      if (ok) {
          v->estacionar();
          parking[n_v] = v;
          ++n_v;
      }
  }
  void Parking::destruir() {
      for (int i = 0; i < n_v; ++i) {
          delete parking[i];
      }
      n_v = 0;
  }
  void Parking::mostrar() const {
      for (int i = 0; i < n_v; ++i) {
          cout << parking[i]->id() << endl;
      }
  }
  Vehiculo* Parking::extraer(const std::string& id) {
      Vehiculo* v = NULL;
      int i = buscar(id);
      if (i < n_v) {
          v = parking[i];
          --n_v;
          parking[i] = parking[n_v];
          v->mover();
      }
      return v;
  }
  int Parking::buscar(const std::string& id) const {
      int i = 0;
      while ((i < n_v)&&(id != parking[i]->id())) {
          ++i;
      }
      return i;
  }
  void Parking::copiar(const Parking& o) {
      n_v = o.n_v;
      for (int i = 0; i < o.n_v; ++i) {
          parking[i] = o.parking[i]->clone();
      }
  }
}
```

EJEMPLO (III)

```
#include "vehiculo.hpp"
#include "bicicleta.hpp"
#include "automovil.hpp"
#include "parking.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;
int main()
{
    bool ok;
    Parking park;
    Automovil* a1 = new Automovil("A1");
    Bicicleta* b1 = new Bicicleta("B1");
    Vehiculo*  a2 = new Automovil("A2");
    park.anyadir(a1, ok);
    park.anyadir(b1, ok);
    park.anyadir(a2, ok);
    Parking aux = park;
    Vehiculo*  v = aux.extraer("A1");
    if (v != NULL) {
        cout << "Vehiculo: " << v->id() << endl;
        v->mover();
    }
    aux.mostrar();
    park.mostrar();
    delete v;
}
```

T4-Anexo: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

RESUMEN DE PROGRAMACIÓN ORIENTADA A OBJETOS (I)

- La Orientación a Objetos (OO) añade **herencia**, **polimorfismo** y **vinculación dinamica** a la abstracción y encapsulación proporcionada por los TADS.
- La **herencia** modela relaciones **ES-UN** entre clases, definiendo jerarquías de clases relacionadas.
  - Las clases derivadas heredan tanto los atributos como los métodos de las clases base.
  - Las clases derivadas pueden definir nuevos atributos y nuevos métodos.
  - Las clases derivadas pueden redefinir los métodos (y el comportamiento) definidos por las clases base.
- El **polimorfismo** sustenta el principio de sustitución, donde los objetos de las clases derivadas pueden sustituir y tomar el papel de los objetos de la clase base. La dirección opuesta no es adecuada.
- La **vinculación dinamica** permite que la redefinición de métodos de las clases derivadas sea efectiva en contextos polimórficos, es decir, se invoque adecuadamente al método **redefinido** en contextos polimórficos.
- En contextos polimórficos, sólo se pueden invocar a los métodos públicos especificados por el tipo de la referencia polimórfica.

RESUMEN DE PROGRAMACIÓN ORIENTADA A OBJETOS (II)

- La OO se implementa en C++ como **Clases Polimórficas**, donde tanto el destructor como los métodos son **virtual** (se especifica en el HPP).
- La herencia se especifica en la definición de la clase:
  `class Derivada : public Base { /* ... */ };`
- La lista de inicialización de los constructores debe invocar a los constructores de la clase base.
- El constructor de copia se define en el ámbito `protected`.
- Se desactiva el operador de asignación definiéndolo en el ámbito `private` (sin implementación).
- El método `clone()` devuelve un puntero a un nuevo objeto creado en memoria dinámica (`new`) como copia del objeto actual.
- En la clase derivada, los métodos heredados no se deben especificar, salvo que se vayan a redefinir.
- Cuando se redefine un método en la clase derivada, se puede invocar al método de la clase base cualificándolo con el nombre de la clase base: `Base::metodo()`
- Los objetos polimórficos se crean siempre en memoria dinámica (*heap*) con `new`, se utilizan a través de punteros (`*`, `->`), y se destruyen con `delete`.

COMPARATIVA EN LA IMPLEMENTACIÓN DE TADs y POO (I)

- La Orientación a Objetos (OO) añade **herencia**, **polimorfismo** y **vinculación dinamica** a la abstracción y encapsulación proporcionada por los Tipos Abtractos de Datos (TADS).

- Los TADS se implementan como **Clases No-Polimórficas** mientras que la OO se implementa como **Clases Polimórficas** (destructor y métodos **virtual**).

- Los objetos **no-polimórficos** se alojan en el **stack** o dentro (empotrado) de una estructura (**nodo** de lista enlazada) en el heap. Los objetos **polimórficos** se alojan siempre en el **heap** directamente, de forma aislada e independiente.

- No existirá un puntero a un objeto de una clase **no-polimórfica**, y los punteros a nodos de lista enlazada se definen con **typedef**. Los punteros a objetos de clases **polimórficas** se definen directamente con **asterisco**.

- Los objetos **no-polimórficos** siempre se pasan como parámetros por **referencia**. Los objetos **polimórficos** siempre se pasan como parámetros de tipo puntero (con asterisco) por valor (o por referencia en caso de ser necesario).

- Una función puede devolver una copia (por valor) de un objeto **no-polimórfico** (o un puntero a un nodo de lista enlazada donde se encuentra ese objeto). Una función puede devolver un puntero a un objeto **polimórfico**.

COMPARATIVA EN LA IMPLEMENTACIÓN DE TADs y POO (II)

- La invocación a métodos de objetos **no-polimórficos** se realiza a través del operador punto (.). La invocación a métodos de objetos **polimórficos** se realiza a través del operador flecha (->).

- Los objetos de clases **no-polimórficas** se pueden copiar y asignar. Los objetos de clases **polimórficas** se pueden clonar.

```
Dato func(Dato& d) {
    d.metodo2();
    Dato aux = d;
    return aux;
}
int main() {
    Dato d1;
    d1.metodo1();
    Dato d2 = func(d1);
}
```

```
Vehiculo* func(Vehiculo* v) {
    v->metodo2();
    Vehiculo* aux = v->clone();
    return aux;
}
int main() {
    Vehiculo* v1 = new Bicicleta();
    v1->metodo1();
    Vehiculo* v2 = func(v1);
    delete v2;
    delete v1;
}
```

T4-Anexo: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS