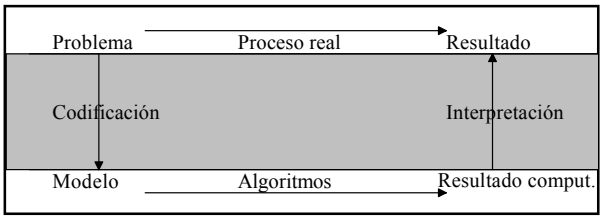
TEMA 2

PROGRAMACIÓN ORIENTADA A OBJETOS

- Conceptos básicos
  - *Objeto, mensaje*
  - *Clase, Herencia*
  - *Polimorfismo*
- Ecuación fundamental de la POO
- Clasificación de LOO
- Algunos lenguajes y ejemplos
  - *Simula, Eifell, etc.*
  - *Smalltalk*

Modelado del mundo real

- La Programación orientada a objetos (POO) se basa en estructuras modulares basadas en datos (**objetos**) que incluyen los procedimientos de acceso a esos datos (**métodos**)
- Los objetos en la POO deben modelar objetos del mundo real.
- El Diseño orientado a objetos (DOO) es un modelo de construcción de software basado no en la función que dicho software debe realizar sino en los objetos que se manipulan.



Pasos en el Diseño Orientado a Objetos

1. Localizar los objetos
2. Describir los objetos (su conducta)
3. Describir las relaciones entre los objetos
4. Usar los objetos

¿Qué puede ser un objeto?

- Un número
- Una cola
- Un diccionario
- Un compilador
- Una ventana
- Una urna
- .....

- Es decir, un objeto pertenece al dominio del problema

Objetos

- Representan a los datos del problema real
- Booch: "Entidad tangible que representa un comportamiento bien definido"
- Desde la perpestiva del conocimiento humano:
  - *Algo tangible y/o visible*
  - *Alguna cosa que pueda ser comprendida intelectualmente*
  - *Algo hacia lo que se dirige el pensamiento o la acción*
- Representa un elemento individual e identificable, real o abstracta, con un comportamiento bien definido en el dominio del problema

objeto = estado + comportamiento + identidad

Objeto. Estado

- **Propiedades** o **atributos** que caracterizan al objeto
- Cada atributo debe tener un **valor** en algún dominio
- Los valores de los atributos pueden variar a lo largo de la vida del objeto
- Los atributos de un objeto son tenidos en cuenta según el dominio del problema
  - *Si quiero vender un coche, los atributos que intereresan son el precio, color, potencia, terminación,...*
  - *Si lo que quiero es participar en un rally, lo que interesa es aceleración, potencia, velocidad, anchura de ruedas,*
- Los atributos de un objeto **deben** ser privados al objeto

Objeto. Comportamiento

- Viene determinado por la forma en la que el objeto interactúa con el sistema.
- La forma de actuar sobre un objeto es enviándole un **mensaje**
- El mensaje activará un comportamiento del objeto (**método**) que será el que determine su forma de actuar
- Los métodos son los comportamientos del objeto
- Pueden generarse métodos para
  - *Permitir consultas sobre aspectos internos del objeto*
  - *Modificar atributos del objeto*
  - *Envíe mensajes a otros objetos*
  - *...*

Objeto. Identidad

- Propiedad característica que los distingue del resto de los objetos
- Dos objetos con
  - *los mismos atributos*
  - *los mismos valores en sus atributos*
    - Son iguales pero no idénticos
- Para distinguir objetos se utilizan varias técnicas
  - *Utilizar dirección de memoria o referencia*
  - *Utilizar nombres definidos por el usuario*
  - *Utilizar claves identificadoras internas o externas*
- La identidad de un objeto permanece con él durante toda su vida
- La identidad no se la proporciona el usuario. Simplemente, la tiene

Objeto

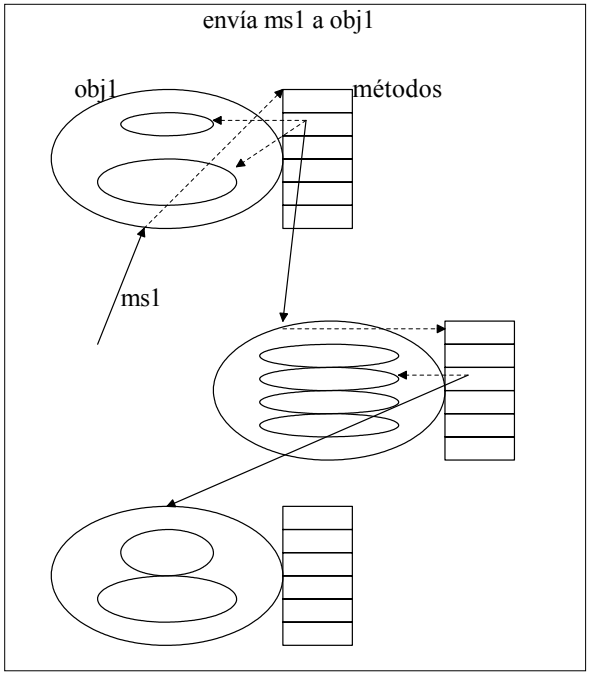
- Implementación
  - *Mantiene una **memoria privada** que describe las propiedades del objeto (su estado). Sus atributos (variables de instancias, slot o datos miembro)*
  - *Disponen de un conjunto de **operaciones** que actúan sobre dicha memoria privada. Son las que definen su comportamiento (métodos o funciones miembro)*
  - *Un objeto tiene una identidad. (Normalmente una dirección)*
- La forma en la que un método actúa sobre un objeto es a través del envío de **mensajes**. En un mensaje intervienen:
  - ***Receptor*** *Es el objeto que recibe el mensaje*
  - ***Selector*** *Es el comportamiento que está involucrado en el mensaje*

Resultado del envío de un mensaje

- El resultado esperado del envío de un mensaje dependerá:
  - Del estado en que se encuentre dicho objeto
  - Del método involucrado en el mensaje
  - De la información que este mensaje pueda portar
- Un **método** tiene **total visibilidad** sobre los **atributos del objeto** al cual le han enviado el mensaje.
  - *Cuando a un objeto se le envía un mensaje, el método activado puede utilizar como variables los atributos del objeto receptor pudiendo incluso modificarlos*
- El único camino para acceder al estado de un objeto **debe ser** por la activación de algún método, es decir, por el envío de un mensaje

☐ El conjunto de métodos que un objeto es capaz de responder es su **protocolo** o **interfaz** y define su conducta

Primera visión de un Sistema Orientado a Objetos



Ejemplo

Se desea realizar el siguiente experimento.

- De una urna que contiene inicialmente un número determinado de bolas blancas y otro número determinado de bolas negras, se pretende realizar lo siguiente:

```
Mientras en la urna quede más de una bola,
   Sacar dos bolas de la misma,
   Si ambas son del mismo color
      Introducir una bola negra en la urna,
   Si ambas son de distinto color
      Introducir una bola blanca en la urna,
Extraer la bola que queda y determinar su color.
```

- Vamos a estudiar el objeto urna, su estado y su comportamiento.

Objeto: Una urna

Memoria privada del objeto:

- Número de bolas blancas
- Número de bolas negras

Interfaz (métodos)

sacaBola() Devolverá el color de la bola sacada. Decrementa en uno el número de bolas de ese color

meteBola(Color) Incrementa en uno el número de bolas del color dado

quedanBolas() Devuelve cierto si hay bolas en la urna

quedaMasDeUnaBola() Devuelve cierto si hay más de una bola en la urna

Int totalBolas() Devuelve el número total de bolas. (privado)

Creación y destrucción de un objeto

- Crear un objeto es como crear una variable de tipo, por tanto tienen las mismas propiedades que los datos de cualquier tipo. Destruirlo debe ser igual
- Además hay que inicializarlo
- Todo esto es algo muy **dependiente** del lenguaje. Se verá en cada lenguaje como hacerlo

Mientras no se hable de lenguajes, supongamos:

- Que todo se puede escribir en "PseudoC"
- Que el envío de un mensaje a un objeto **u** con selector **m** se escribe por **u.m**
- Que la manera de crear e inicializar un objeto de algún tipo, es simplemente creando una variable de ese tipo y que en la creación se le propociona el valor inicial de sus variables de instancias como argumentos.
- Para crear el objeto urna u con 34 bolas bancas y 56 negras se escribirá **Urna u(34,56)**.

- El ejemplo resuelto en PseudoC

```
main()
{
   /*u es un objeto urna con 34 bolas
       blancas y 56 negras */
   Urna u(34,56);
   char a,b;

   while (u.quedaMasdeUnaBola())
   {  // en cada pasada, el número
     // de bolas disminuye en uno
      a=u.sacaBola();
      b=u.sacaBola();
      if (a==b)
         u.meteBola('n');
      else
         u.meteBola('b');
   }
   printf("La bola final es de color
          %c \n",u.sacaBola());
}
```


Clases

La definición de urna anterior no define el comportamiento de una urna, sino de todas las urnas. (Se ha definido una clase)

- **u** es sólo un ejemplo de ese comportamiento (Un objeto Urna o una **instancia** de la **clase Urna**)

Una clase describe el comportamiento de una familia de objetos.

- Es una plantilla que se utiliza para definir los objetos.
- Puede verse como un tipo (que además define métodos)
- Los objetos que se definen a partir de esa clase son sus instancias.
- Son las instancias las que reciben los mensajes definidos en su clase.

En el ejemplo:

- Urna puede verse como la clase que describe el comportamiento de todas las urnas
  - *u es una instancia de esa clase.*

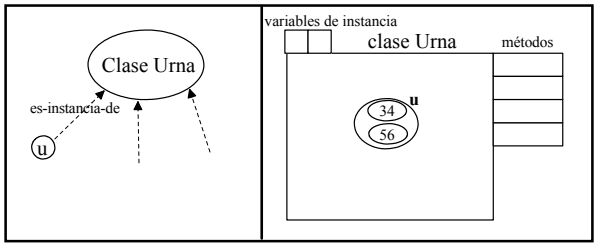
Los atributos que describen la memoria privada de un objeto se llaman

- variables de instancia, slot, datos miembro

Los comportamientos se llaman

- métodos, funciones miembro

Dos formas de ver una clase



Instancias

El número de bolas blancas y el número de bolas negras son dos variables de instancia definidas en la clase.

- los valores 34 y 56 inicialmente son los valores de las variables de instancia que definen el estado de la urna u.

Es evidente que podemos crear varias urnas

```
main()
{
   Urna u(34,67),v(89,23);

   u.meteBola('n');
   // u tiene ahora 34 blancas y 68 negras
   v.meteBola('b');
   // v tiene ahora 90 blancas y 23 negras
   ....
```

- Ambas responden a los mismos mensajes con los mismos métodos pero su actuación depende de su estado que es particular

Descripción de una clase

```
Clase Urna
   Variables de Instancia privadas
      Número de bolas blancas (blancas)
      Numero de bolas negras   (negras)
   Métodos públicos
      sacaBola()
      meteBola(color)
      quedanBolas()
      quedaMasDeUnaBola()
   Métodos privados
      totalBolas()
Fin Clase
```

Las clases son las unidades básicas de la POO

- La implementación depende del lenguaje. Si utilizamos el PseudoC, se antepondrá el nombre de la clase seguido de :: al nombre del método en su definición para indicar a la clase a la que pertenece dicho método

```
char Urna::sacaBola()
{
   ....
}
```


Referencias al propio objeto

- Un objeto puede enviarse un mensaje a sí mismo

```
int Urna::totalBolas()
{
   return (blancas+negras);
}
int Urna::quedaMasDeUnaBola()
{
   return (1 < mimismo.totalBolas());
}
```

- mimismo = self, current, this

- Hay lenguajes que la referencia a mimismo puede suprimirse

```
int Urna::quedaMasDeUnaBola()
{
   return (1 < totalBolas());
}
```


Ventajas de la utilización de clases

- Cada clase puede ser creada de modo independiente

- Cada clase puede probarse de modo independiente

- Asegura la consistencia de los datos pues ofrece un interfaz para su manejo

- La implementación queda escondida al usuario de la clase (lo mismo que la implementación de los enteros queda oculta a los que los usan)

- Puede variarse la implementación sin tener que cambiar los programas que las utilizan.

- Es altamente reutilizable

Resumen

- Todas las instancias de una clase responden al mismo conjunto de mensajes con los mismos métodos

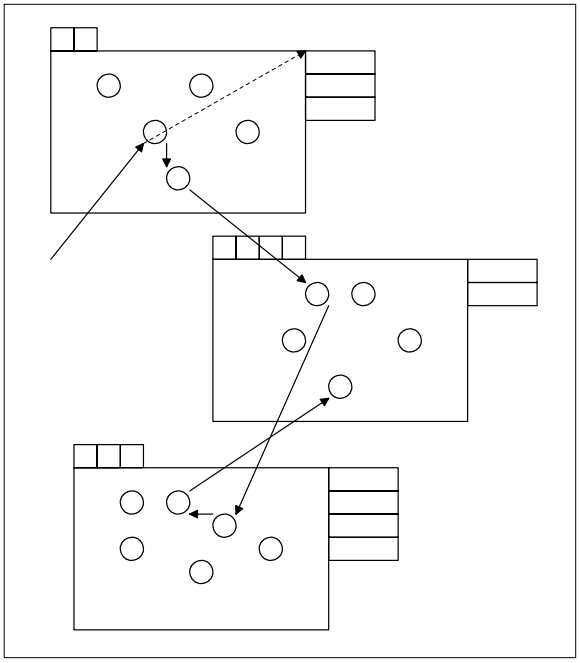
- Todas las instancias de una clase tienen las mismas variables de instancias pero cada una con sus valores

Programadores en la Programación Orientada a Objetos

1. Productores de clases
2. Consumidores de clases

- Un programa puede contener instancias de clases previamente definidas y clases definidas específicamente para este programa.
- La relación entre variables de instancia y los objetos de una clase se define como una relación de tipo *"..es parte de.."*
  - *Así, el atributo **número de bolas blancas** de la urna u es parte de la urna u*
- Un objeto puede contener como una parte suya a otros objetos. **Agregación**
- Un objeto puede contener como una parte suya referencias a otros objetos. **Asociación**
- **Composición: Agregación o Asociación**

Segunda visión de un Sistema Orientado a Objetos



Ecuación fundamental de la POO

PROGRAMACION ORIENTADA A OBJETOS

=

TIPOS ABSTRACTOS DE DATOS

+

HERENCIA

+

POLIMORFISMO

Tipos Abstractos de datos

Permiten:

Encapsulación Guardar conjuntamente datos y operaciones que actuan sobre ellos

Ocultación Proteger los datos de manera que se asegura del uso de las funciones definidas para su manejo

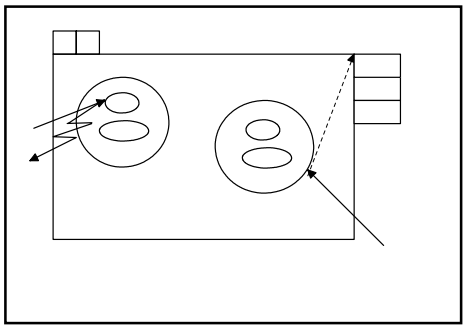
Ventajas:

- Implementación escondida al cliente
- Los tipos abstractos se generan independientemente
- Los tipos abstractos se pueden probar independientemente
- Aseguran la consistencia de los datos
- Aumentan la reutilización de código

Las posibilidades de ocultación de la información dependen del lenguaje en el que se implemente.

- Hay lenguajes que ponen diferentes niveles de privacidad
- Hay lenguajes que no pueden hacer las variables de instancias privadas

- Una clase se implementa como un tipo abstracto de datos



Ejercicio1:

- Definir el problema anterior con urnas trampas. Son urnas que despues de 10 extracciones, tienen una probabilidad del 0.2 de cambiar todas las bolas blancas por negras y viceversa.

Ejercicio2:

- Dos jugadores disponen cada uno de una urna.
- Inicialmente, cada uno toma 20 bolas de los colores que quiera (blancas o negras). Por ejemplo, uno puede meter en la suya (10,10) y el otro (2,18)
- Cada jugador conoce la identidad de su oponente

```
Mientras quede mas de una bola
        en cualquiera de las dos urnas,
   El jugador A realiza su jugada
   El jugador B realiza su jugada
   Si en las urnas de A y de B queda una bola
        empatan
   Si en la urna de A queda una bola, A pierde
   Si en la urna de B queda una bola, B pierde
```


La jugada de A consiste en:

```
Sacar una bola de su urna
Sacar una bola de la urna de su contrincante
Si son iguales
      meter una bola negra al contrincante
Si son distintas
      devolver al contrincante su bola
```

La jugada de B consiste en:

```
Sacar una bola de su urna
Sacar una bola de su contrincante
Si son iguales
      meter una bola blanca al contrincante
Si son distintas
      devolver la bola del contrincante
```

- Pista
  - *Cada jugador deberá tener en propiedad una urna y conocer al adversario*

Solución ejercicio1:

```
Clase UrnaTrampa
   Variables de Instancia privadas
      blancas
      negras
      contExtracciones
   Métodos públicos
      sacaBola()
      meteBola(color)
      quedanBolas()
      quedaMasDeUnaBola()
   Métodos privados
      totalBolas()
Fin Clase
```

y el programa

```
main()
{
   UrnaTrampa v(23,34);
   char a,b;
   while (v.quedaMasDeUnaBola())
   {// cada pasada, decrementa una bola
      a=v.sacaBola();
      b=v.sacaBola();
      if (a==b)
         v.meteBola('n');
      else
         v.meteBola('b');
   }
   printf("La bola final es de color %c
          \n",v.sacaBola());
}
```

➢ Ahora hay que implementar la clase UrnaTrampa

- *Es igual que Urna salvo el método sacarBola()*

Solución ejercicio2

```
Clase JugadorA
   Variables de instancia privadas
      Urna     urna
      JugadorB contrincante
   Métodos públicos
      contra(JugadorB)
      jugada()
      urna()
Fin clase

Clase JugadorB
   Variables de instancia privadas
      Urna     urna
      JugadorA contrincante
   Métodos públicos
      contra(JugadorA)
      jugada()
      urna()
Fin clase
```


y el programa

```
main(){
   JugadorA J1(10,15);
   JugadorB J2(13,12);
   J1.contra(J2);
   J2.contra(J1);
   while (J1.urna().quedaMasDeUnaBola()
       && J2.urna().quedaMasDeUnaBola())
   {
      J1.jugada();
      J2.jugada();
   }
   if (J1.urna().quedaMasDeUnaBola())
      printf( "GANA J1\n");
   else if (J2.urna().quedaMasDeUnaBola())
      printf( "GANA J2\n");
   else
      printf( "EMPATE\n");
}
```


Veamos la descripcion de los métodos **jugada** en cada clase

```
JugadorA::jugada()
{
      char m, c;
      // Un método de una clase, ve las
       variables de instancia
      // del objeto al cual se le envía
         el mensaje
      m=urna.sacaBola();
      c=contrincante.urna().sacaBola();
      if (m==c)
         contrincante.urna().meteBola('n');
      else
         urna.meteBola(c);
}
```

```
JugadorB::jugada()
{
   char m, c;
   m=urna.sacaBola();
   c=contrincante.urna().sacaBola();
   if (m==c)
      contrincante.urna().meteBola('b');
   else
      urna.meteBola(c);
}
```

Otro caso

```
JugadorA   juan,pepe,antonio
```

El mensaje

```
juan.urna().sacaBola(),
      se refiere a una bola de la urna de juan
antonio.urna().sacaBola(),
      se refiere a una bola de la urna de antonio
pepe.urna().meteBola('n'),
      mete una bola negra en la urna de pepe
```

➢ Si las variables de instancia fueran públicas

```
pepe.contrincante.urna.meteBola('n')
      mete una bola negra en la urna del
      contrincante de pepe
```


Herencia

Es una técnica que permite incrementar la reusabilidad.

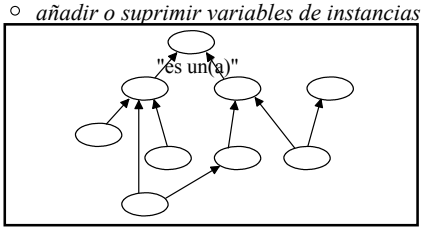
➢ Maneja eficientemente relaciones *"..es como un..."*

➢ Crea nuevas clases a partir de generalizar o especializar otras clases ya existentes.

Para ello a la hora de crear una clase puede reutilizar parte de la conducta de otra clase.

➢ Esto se hace por medio de:

- *añadir, suprimir o modificar métodos*
- *añadir o suprimir variables de instancias*



La clase así resultante sería una clase que es heredera de la inicial.

Si A hereda de B,

➢ A es hija de B, A es subclase de B, A es derivada de B

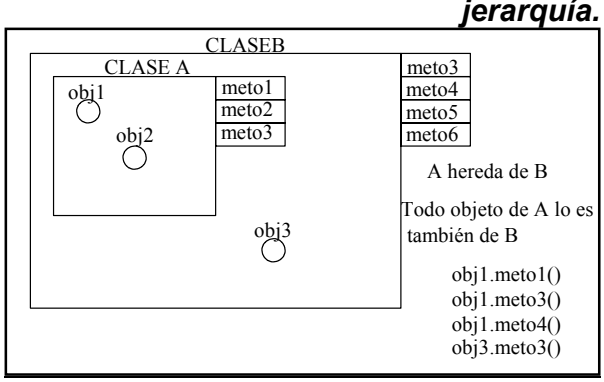
➢ B es padre A, B es superclase de A, B es ancestro de A y de sus subclases

La herencia puede ser

simple. Una clase hereda exclusivamente de otra clase

múltiple. Una clase hereda de varias clases

La herencia clasifica a las clases en una jerarquía.



Una urna heredada

Sea UrnaTrampa una clase que hereda las propiedades de la clase Urna.

La clase UrnaTrampa es como una Urna pero con un comportamiento especial

```
Clase UrnaTrampa hereda Urna
   Variables de Instancias Privadas
      contExtracciones
   Métodos privados
      cambiaBolas()
Fin Clase
```

- No se indican los métodos que hereda de la clase Urna
- Una vez definida, se usa como una clase más
- Una clase puede redefinir los métodos de la clase de la que hereda. La nueva definición puede apoyarse en la antigua.

Se puede redefinir el ejercicio1 basada en esta nueva definición de la clase UrnaTrampa

Referencias al padre

```
char Urna::sacaBola()
{
   if (random()*mimismo.totalBolas() < blancas)
   {
      blancas--;
      return 'b';
   }
   else
   {
      negras--;
      return 'n';
   }
}
```

- Definimos sacaBola() en UrnaTrampa

```
int UrnaTrampa::sacaBola()
{
   if (contExtracciones==10)
   {
      contExtracciones = 0;
      if (random()>=0.2)
         mimismo.cambiaBolas();
   }
   return mipadre.sacaBola();
}
```


- mipadre = super, nombre de la clase::
- En este caso, no hay que modificar ningún método más en la clase UrnaTrampa
- Si en la clase Urna se define el método

```
bool Urna::dosIguales()
{
   char a,b;
   a = mimismo.sacaBola();
   b = mimismo.sacaBola();
   mimismo.meteBola(a);
   mimismo.meteBola(b);
   return (a==b);
}
```

- tampoco habría que modificarlo en UrnaTrampa.
- Por supuesto, habría que crear el método

```
UrnaTrampa::cambiaBolas()
{
   int temp;
   temp = blancas;
   negras = blancas;
   blancas = temp;
}
```


Herencia frente a composición

- Herencia: A hereda de B cuando A es como un B ....
  - Relación de inclusión entre clases
- Composición: B contiene a A cuando A es parte de B.
  - Relación de cliente-servidor entre clases
  - Una clase usa o se basa en otra para su definición
- Ejemplo:
  - Herencia: clase Punto y clase Pixel (es un punto con color)
  - Composición: clase Chasis, clase Motor, clase Ruedas, etc. forman parte de la clase Coche

Clase Abstracta

Clases que proporcionan un interfaz **común** y **básico** a sus herederas.

De ella no se obtendrá ninguna instancia

Definirá métodos con el cuerpo vacío o métodos con comportamientos generales a todas las subclases

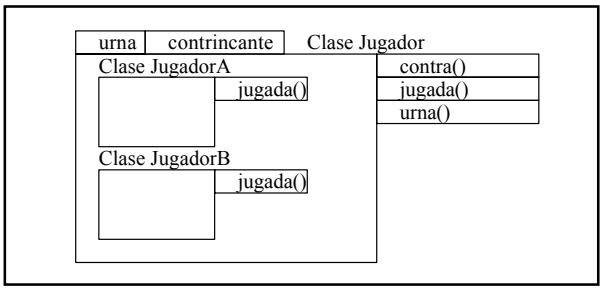
- Supongamos que se define una clase Jugador que define las características generales de cualquier jugador

```
Clase Jugador
   Variables de instancia privadas
      Urna    urna
      Jugador contrincante
   Métodos públicos
      contra(Jugador)
      jugada()
      urna()
Fin clase
```


Clase Abstracta

Esta clase abstracta definirá todos sus métodos excepto el método jugada() que será definido de forma particular por cada subclase.

```
Clase JugadorA hereda Jugador
Clase JugadorB hereda Jugador
```



- Jugada será un método vacio en la clase Jugador
- Cada subclase JugadorA y JugadorB definen su propio método jugada()

Polimorfismo

Capacidad de una entidad de referenciar distintos elementos en distintos instantes.

Dos tipos de polimorfismo

- Por sobrecarga de funciones
  - *ad hoc*
  - *a medida*
- Por vinculación dinámica
  - *paramétrico*

Polimorfismo por sobrecarga de funciones

Se comentó al principio cuando se habló de sobrecarga de funciones

- Dos funciones con el mismo nombre y distintos argumentos son funciones distintas
- Dos funciones con el mismo nombre y con los mismos argumentos pero definidas en distintas clases son funciones distintas.

☐ Este tipo de polimorfismo es resuelto en tiempo de compilación

- Permite especificar un mismo nombre a funciones que realizan una misma actividad en distintas clases
- Objetos de distintas clases, pueden recibir el mismo mensaje. Cada clase dispone de un método para aplicar a dicho mensaje

Polimorfismo por vinculación dinámica

Vinculación estática:

- En tiempo de compilación se conoce el tipo de todas sus variables y expresiones.

Vinculación dinámica:

- En tiempo de compilación no puede determinarse el tipo de algunas variables o expresiones.
- El compilador puede que no sepa a qué clase pertenece un objeto antes de la ejecución.
- Por tanto hasta tanto no se determine la clase a la que pertenece el objeto, no podrá sustituir el cuerpo. Por tanto, esta vinculación tiene que hacerse en tiempo de ejecución.

Polimorfismo dinámico

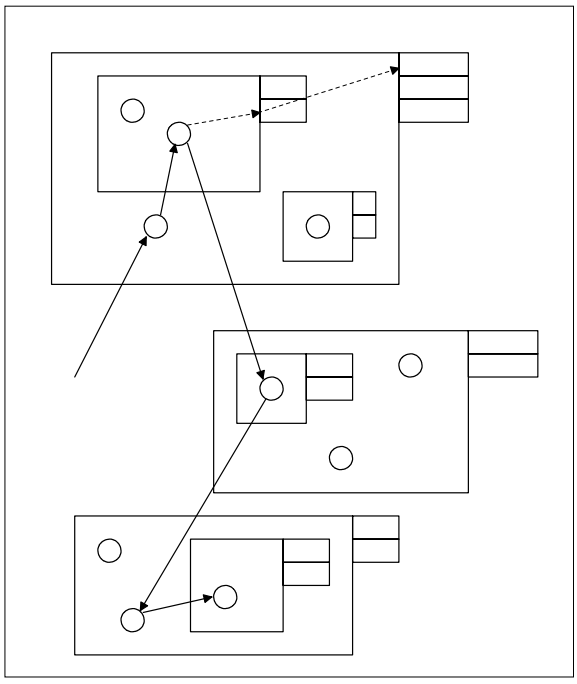
- ***"Un identificador de variable que referencie a un objeto de una clase, puede pasar a referenciar a otro objeto de otra clase"***

- C++ impone ciertas restricciones a esta frase:
  - ***"siendo esta otra clase una subclase o derivada de la anterior"***
  - *Es decir, donde se espera un objeto de una clase, se puede recibir un objeto de una clase derivada.*

- **Así, por ejemplo**

```
JugadorA J1(20,20);
JugadorB J2(15,25);
Jugador N;
if (condicion)
   N=J1;
else
   N=J2;
N.jugada(); // Qué jugada es?
```


Tercera visión de un Sistema Orientado a Objetos



Ejemplo

Supongamos que queremos realizar un sistema que trabaje con figuras geométricas planas.

El sistema debe poder mostrar información de una figura y computar su área.

Se manejaran varios tipos de figuras, entre ellas, círculos y rectángulos

Dos aproximaciones:

- Sin utilizar las técnicas de la POO
  - *Se utilizará un código C para la descripción del programa*
- Utilizando las técnicas de la POO
  - *Se utilizará un pseudocódigo para describir el programa. En el capítulo dedicado a C++ se matizaran todos los aspectos de pseudocódigo utilizados*

Sin POO

```
#define TCirculo  1
#define TRectangulo  2
typedef struct Circulo
{
   short tipo;   // tipo de figura
   double x,y;   // centro del circulo
   double radio; // radio del circulo
} Circulo
typedef struct Rectangulo
{
   short tipo;    // tipo de figura
   double x1,y1;  // corner inicial
   double x2,y2;  // corner final
} Rectangulo
typedef union Figura
{
   short tipo;      // tipo de figura
   Circulo  cir;    // datos del circulo
   Rectangulo  rec; // datos del rectangulo
} Figura

// prototipo de funciones
double area(Figura *p_figura);
void muestra(Figura *p_figura);
```


Sin POO

```
// Funcion que computa el area
double area(Figura *p_figura)
{
   double varea;
// se maneja el area dependiendo del tipo
switch(p_figura->tipo)
{
      case TCirculo:
         varea=M_PI*p_figura->cir.radio*
                 *p_figura->cir.radio;
         break;
      case TRectangulo:
         varea=fabs(
            (p_figura->rec.x2-
             p_figura->rec.x1)*
            (p_figura->rec.y2-
             p_figura->rec.y1));
         break;
      default: printf("Figura desconocida\n");
               return -1;
   }
   return varea;
}
```


Sin POO

```
// Funcion que muestra la figura
void  muestra(Figura *p_figura)
{
   printf("Figura: ");
   switch(p_figura->tipo)
   {
      case TCirculo:
         printf("Círculo de radio %f y
            de centro (%f,%d)\n",
            p_figura->cir.radio,
            p_figura->cir.x,
            p_figura->cir.y);
         break;
      case Trectangulo:
         printf("Rectángulo corner (&f,%f)
            y (%f,%d)\n",
            p_figura->rec.x1,
            p_figura->rec.y1,
            p_figura->rec.x2,
            p_figura->rec.y2);
         break;
      default: printf("Desconocida \n");
   }
}
```


Sin POO

```
int main()
{
   int i;
   Figura s[2];  //array con dos figuras
   // inicializa s[0] con un rectangulo
   s[0].tipo=TRectangulo;
   s[0].rec.x1=80.0;
   s[0].rec.y1=30.0;
   s[0].rec.x2=300.0;
   s[0].rec.y2=50.0;
   // inicializa s[1] con un circulo
   s[1].tipo=TCirculo;
   s[1].cir.radio=80.0;
   s[1].cir.y=40.0;
   s[1].cir.x=30.0;
   // Computando areas
   for (i=0;i<2;i++)
      printf("Area figura[%d]=%f\n",i,
            area(&s[i]));
   // Mostrando figuras
   for (i=0;i<2;i++)
      muestra(&s[i]);
   return 0;
}
```

Sin POO

Añadiendo una nueva figura. Queremos incorporar triángulos.

➢ 1 Añadir el tipo Triangulo

```
#define TTriangulo 3
typedef struct Triangulo
{
   short tipo;   // tipo de figura;
   double x1,y1; // Coor. de punto 1
   double x2,y2; // Coor. de punto 2
   double x3,y3; // Coor. de punto 3
}
```

➢ 2 Añadirlo a la unión de figuras

```
typedef union Figura
{
   short tipo;      // tipo de figura
   Circulo  cir;    // datos del circulo
   Rectangulo  rec; // datos del rectangulo
   Triangulo tri;   // datos del triangulo
} Figura
```

➢ 3 Modificar las funciones muestra y area

Con POO

Creamos una clase abstracta Figura. Esta clase soporta los métodos muestra y area

```
Clase Figura
   Metodos de instancia publicos
      muestra()
      area()
Fin clase
Figura::muestra()
{
   printf("Debe implementarla la subclases\n");
}
Figura::area()
{
   printf("Debe implementarla la subclases\n");
   return 0.0;
}
```


Con POO

Creamos Circulo como subclase de Figura

```
Clase Circulo hereda Figura
   Variables de instancia publicas
      radio, x, y;
Fin Clase

// Implementacion de muestra para Circulo
Circulo::muestra()
{
   printf("Circulo de radio %f y
           de centro (%f,%d)\n",
           radio,x,y);
}

// Implementacion de area para Circulo
Circulo::area()
{
   return M_PI*radio*radio;
}
```


Con POO

Creamos Rectangulo como subclase de Figura

```
Clase Rectangulo hereda Figura
   Variables de instancia publicas
      x1,y1,x2,y2;
Fin Clase

// Implementacion de muestra para Rectangulo
Rectangulo::muestra()
{
   printf("rectangulo corner (&f,%f)
         y (%f,%d)\n",
        x1,y1,x2,y2);
}

// Implementacion de area para Rectangulo
Rectangulo::area()
{
   return fabs((x2-x2)*(y2-y1));
}
```

Con POO

```
int main()
{
   int i;
   Figura s[2];
   // Inicializacion de un rectangulo
   s[0]=Rectangulo(80.0,30.0,300.0,50.0);
   // inicializa de un circulo
   s[1]=Circulo(80.0,40.0,30.0);
   // Computando areas
   for (i=0;i<2;i++)
      printf("Area figura[%d]=%f\n",i,
           s[i].area()); //vinculación dinamica
   // Mostrando figuras
   for (i=0;i<2;i++)
      s[i].muestra(); //vinculación dinamica
   return 0;
}
```

➢ Este código no es C++. Está subrayado lo que se debe matizar

Con POO

Añadiendo una nueva figura. Queremos incorporar Triangulo.

➢ 1 Creamos la clase como derivada de Figura

```
Clase Triangulo hereda Figura
   Variables de instancia publicas
      x1,y1,x2,y2,x3,y3
Fin Clase

Triangulo::muestra()
{
   printf( ....
}
Triangulo::area()
{
   return  ....
}
```

➢ y usarla del mismo modo que las otras

Clasificación de los lenguajes Orientados a Objetos

Nivel 1. Estructura modular basada en objetos. El sistema está modularizado en función de las estructuras de datos

Nivel 2. Abstracción de datos. Los objetos están descritos como tipos abstractos de datos.

Nivel 3. Recolección automática de basuras. Los objetos no referenciados deben ser recolectados automáticamente por el sistema.

Nivel 4. Clases y módulos. Coherencia sintáctica.

Nivel 5. Herencia. Una clase puede ser definida como extensión o restricción de otra.

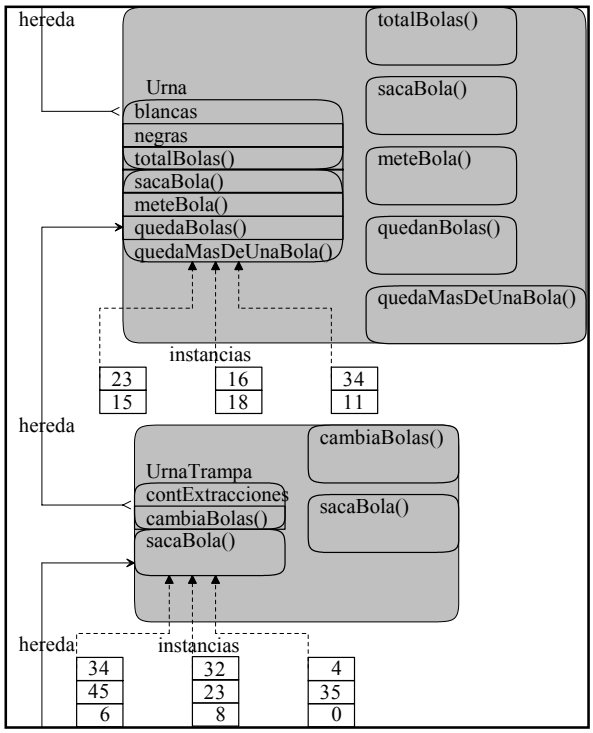
Nivel 6. Polimorfismo. Debe permitirse que las entidades se refieran a objetos de más de una clase.

Nivel 7. Herencia múltiple. Debe permitirse declarar una clase como heredera de varias clases.

	1	2	3	4	5	6	7
ADA							
C++							
SMALLTALK							
EIFELL							
JAVA							(*)

(*) Incluye el concepto de interfaz

CLASE,SUBCLASES E INSTANCIAS



Simula

- 1967. Universiad de Oslo
- Proviene de Simula-1 (simulación)
- Extensión orientada a objetos de Algol-60
- Introduce el concepto de clase
- Herencia simple
- Compilado
- Tipado estático
- Los objetos tienen vida. Son procesos
- Está bien adaptado para simular sistemas paralelos (clases Timer, métodos detach y resume, etc.)

Ejemplo en Simula

- Definición de una clase

```
Class A
   rutinas
   begin
      cuerpo
   end
   ...
end
```

- referencia a un objeto

```
ref(A) obj          obj es una referencia
```

- Creación de un objeto

```
obj :- new A        creación de un objeto
```

- Ejemplo de herencia

```
A class B; begin ... end
```

- Está vivo. Refientemente se le incorporó protección y ocultación

Smalltalk

- Modelo ortogonal basado en los principios de:
  - *Objeto, Clase, Herencia, Mensaje*
- Concebido en su primeras versiones en los 70 por A. Kay y A.Golberg en Xerox Parc.
- Versiones Smalltalk-72, Smalltalk-76, Smalltalk-80
  - *Semicompilado (bytecodes)*
  - *Tipado dinámico*
  - *Herencia simple*
  - *Recogida de basura*
- Es un entorno integrado. Su entorno es el precursor de todos los entornos gráficos actuales
- Buenas herramientas para edición, depurarción e inspección
- Librería de clases extensa

Eifell

- Concebido por Bertrand Meyer
- Es un lenguaje nuevo (No basado en otros)
- Pensado para enfatizar los conceptos de
  - *reusabilidad (clases paramétricas, renombrado)*
  - *fiabilidad (pre-postcondiciones, invariantes)*
- Herencia múltiple
- Tipado estricto
- Recolector de basuras

C++

En 1967 Martin Richars diseña BCPL

En 1970 Ken Thompson (Bell Lab.) diseña B

En 1972 Dennis Ritchie diseña C

C mantiene una versión estándar diseñada por los comites

ANSI (CX3J11) y ISO (JTC1SC22WG14)

En 1989 se estandariza la versión ANSI

Basado en C, se construye un C con clases.

El nombre C con clases, es cambiado por C++

C++ lo diseña Bjarne Stroustrup en 1980

En 1983 sale del entorno de Stroustrup

En 1989 se crea el comite para la normalización de C++ (ANSI)

- ANSI X3J16

A este se une en junio de 1991 otro comite de la ISO

- ISO WG21

Tres versiones:

- 1985
- 1990

Marzo de 1992. La versión 3.0 de AT&T incluye la versión ANSI C.


```
class PilaChar {
   char datos[];
   int indice;
public:
   PilaChar() {
      indice = 0;
      datos = new char[100];
      for(int i=0;i<100;i++) datos[i]=0;
   }
   push(char s) {
       if (!this->isFull())
         datos[indice++]=s;
       else error("Pila llena");
   }
   char pop() {
       if (!this->isEmpty())
         return datos[--indice];
       else error("Pila vacia");
   }
   int isEmpty() {return indice==0;}
   int isFull()  {return indice==100;}
   ~PilaChar() {
      delete [] datos;
   }
};
```



```
main()
{
   // Un objeto "estático"
   PilaChar p;
   p.push('h');
   p.push('o');
   p.pop();
   // Una referencia a un objeto
   PilaChar *q;
   q = new PilaChar();
   q->push('h');
   q->push('o');
   q->pop();
   delete q;
}
```

Java

- Definido por Sun Microsystems en 1.991
- Basado en C++
  - *Eliminando cosas poco agradables*
  - *Añadiendo otras que consideraban que faltaban*
- Es un lenguaje portable
  - *Pseudocompilado*
- Admite trheads
- Tomado como base para el desarrollo de las Web's
- Incluye una gran librería de clases
  - *I/O, contenedores, sistema gráfico (AWT, SWING)*
- Herencia simple. Interface
- Recolector automático de basuras

Pila.java

```
class PilaChar   {
   private char datos[];
   private int indice;
   public PilaChar() {
      datos = new char[100];
      indice = 0;
      for (int i=0;i<100;i++)
            datos[i]=0;
   }
   public void push(char s) {
      if (!this.isFull())
         datos[indice++]=s;
      else
         System.out.println("Error. Pila llena");
   }
    public char pop() {
      if (!this.isEmpty())
         return datos [--indice];
      else {
         System.out.println("Error. Pila Vacia");
        return 0;
      }
   }
    public boolean isEmpty() {
      return indice ==0;
   }
    public boolean isFull() {
      return indice==100;
   }
}
```


Ejpila.java

```
import PilaChar;
class Ejpila
{
    public static void main(String[] args)
    {
        PilaChar p = new PilaChar();
        p.push('h');
        p.push('k');
        p.pop();
    }
}
```

Las figuras en Modula 2

```
DEFINITION MODULE Figuras;

TYPE TIPO = (Circulo, Rectangulo);

FIGURA =
RECORD
   CASE tipo: TIPO OF
     Circulo:     x,y,radio  : LONGREAL;
      | Rectangulo: x1,y1,x2,y2 : LONGREAL
   END;
END

(* Calcula el area de una figura *)
PROCEDURE area(VAR Figura : FIGURA) : LONGREAL;

(* Muestra datos de una figura *)
PROCEDURE muestra(VAR Figura : FIGURA) ;

(* Crea una figura circular *)
PROCEDURE creaCirculo(var figura:FIGURA; vx, vy, vr :
          LONGREAL);

(* Crea una figura cuadrangular *)
PROCEDURE creaRectangulo(var figura:FIGURA;

END Figuras.
```



```
IMPLEMENTATION MODULE Figuras;
IMPORT IO;

PROCEDURE area(VAR Figura : FIGURA) : LONGREAL;
BEGIN
   WITH Figura DO
          CASE tipo OF
            Circulo: RETURN (2*PI*radio);
          | Rectangulo: RETURN ((x2-x1)*(y2-y1));
          END;
       END;
END area;

PROCEDURE muestra(VAR Figura : FIGURA) ;
BEGIN
 WITH Figura DO
     CASE tipo OF
       Circulo: WrStr("El area del circulo es ");
                WrLongReal(area(Figura));
                WrLn;
      Rectangulo:
                WrStr("El area del Rectangulo es ");
                WrLongReal(area(Figura));
                WrLn;
     END;
 END;
END muestra;

PROCEDURE creaCirculo(var figura:FIGURA; vx, vy,
                      vr : LONGREAL);
BEGIN
    figura.tipo  := Circulo;
    figura.x     := vx;
    figura.y     := vy;
    figura.radio := vr;
END creaCirculo;

PROCEDURE creaRectangulo(var figura:FIGURA;
                         vx1, vy1, vx2, vy2: LONGREAL);
BEGIN
    figura.tipo  := Rectangulo;
    figura.x1    := vx1;
    figura.y1    := vy1;
    figura.x2    := vx2;
    figura.y2    := y2;
END creaRectangulo;

END Figuras.
```



```
MODULE PruebaFiguras;
IMPORT Figuras;

VAR m : ARRAY [1..2] OF FIGURA;
    i : CARDINAL;

BEGIN

   creaCirculo(m[1],10,15,90);
   creaRectangulo(m[2],10,15,20,25);

   FOR i := 1 TO 2 DO
      muestra(m[i]);
   END;
END PruebaFiguras.
```