2.- PROGRAMACIÓN ORIENTADA A OBJETOS

El éxito de la Progr. Orientada a Objetos

- **Permite una mejor organización del software:**
  - ✓ Es más "fácil" (??) desarrollar aplicaciones (muy) complejas.
- **Especialmente adecuada para determinadas aplicaciones**
- **La POO soporta un alto grado de reutilización de código:**
  - ✓ Se abarata el proceso de desarrollo

Seamos Realistas

- **Utilizar un lenguaje Orientado a Objetos NO asegura el éxito.**
- **La POO permite hacer las cosas de otra manera, pero lo mismo se puede hacer con lenguajes no-Orientados a Objetos:**
  - ✓ Todo programa se traduce a código máquina, luego cualquier concepto de programación se puede traducir a Ensamblador
  - ✓ Lo importante es utilizar con buen criterio las herramientas que proporciona el lenguaje (cualquiera).

Abstracción y Programación

- **Todos los paradigmas de programación se basan en mecanismos de abstracción.**
- **Programación Orientada a Objetos → Mecanismo básico de abstracción → Objeto.**
- **La única forma de manejar la complejidad de los problemas al resolverlos (programar) es emplear la abstracción:**
  - ✓ Centrarse en los detalles importantes
  - ✓ Ocultar (eliminar) los detalles irrelevantes

  } **Menos información**

La Abstracción es un proceso natural

"Los humanos hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad: abstraernos de ella. Incapaces de dominar en su totalidad los objetos complejos, se ignora los detalles no esenciales, tratando en su lugar con el modelo ideal de objeto y centrándonos en los aspectos esenciales" (Wulft).

- **Las personas construimos modelos (abstracciones) mentales para comprender el mundo y nos servimos de ellos para resolver los problemas.**
  - ✓ Ejemplo:
    - Un mapa es un modelo del territorio que representa, NO es el territorio, es una abstracción que lo simplifica y lo hace manejable.
    - El mapa nos permite: localizar lugares, orientarnos en los desplazamientos, etc..

Mecanismos de abstracción en Programación

- **Procedimiento:**
  - ✓ Permite "ocultar" algoritmos en su interior, también datos útiles, exclusivamente, para estos algoritmos (variables locales).
  - ✓ Una vez construido nos olvidamos de su contenido (está oculto), sólo nos interesa cómo se puede usa:

```
nombre_procedimiento (argumentos)
```

  - ✓ Programación procedimental:
    - Descomponer el problema en procedimientos.
    - Diseño conducido por el procesamiento.

Ejemplo Procedimientos

Procedimientos

Programa:
Usa la definición de procedimientos, es independiente del algoritmo que contienen

```
#include <stdio.h>

typedef int pila[100];

int pop (pila p)
{ /* algoritmo */ }

void push (pila p, int x)
{ /* algoritmo */ }

int main()
{
      pila mis_datos;
      int i;

      for (i=0; i<100; i++)
            push(mis_datos, i);

      for (i=0; i<100; i++)
            printf ("%d", pop(mis_datos));

      return 0;
}
```


Problemas

- **Relación débil entre procedimientos y datos:**
  - ✓ ¿Cómo restringir qué el tipo "`pila`" se use sólo asociado a las operaciones "`push`" y "`pop`"?

    `mis_datos[3] = 67; /* ¡ Es válido! */`

Mecanismos de abstracción en Programación

- **Módulos:**
  - ✓ Archivos que contienen colecciones de procedimientos y datos.
  - ✓ Permite "ocultar" algoritmos y datos en su interior.
  - ✓ Mecanismos de "visibilidad" dentro y fuera del módulo:
    - Elementos públicos: son accesibles desde archivos externos al módulo.
    - Elementos privados: sólo son accesibles desde el interior del módulo.
  - ✓ Programación modular:
    - Descomponer el problema en módulos (compilación separada).
    - Diseño conducido por la organización de los datos.

Ejemplo Módulo

```
/* PILA.H */
/* Sólo declaraciones de */
/* elementos públicos */
#define max 100
int pop ();
void push (int x);
```

Módulo PILA

```
/* PILA.C */
/* Implementación del módulo */
/* Contiene elementos privados * /
#include "pila.h"
static int mis_datos[max];

int pop ()
{ /* algoritmo que usa mis_datos */ }

void push (int x)
{ /* algoritmo que usa mis_datos */ }
```

Módulo:
Oculta "mis_datos"

```
/* Programa que usa la PILA */
#include <stdio.h>
#include "pila.h"

int main()
{
      int i;

      for (i=0; i<100; i++) push(i);
      for (i=0; i<100; i++) printf ("%d", pop());

      return 0;
}
```

Programa:
No hay referencias a "mis_datos",
por lo tanto, no se puede usar incorrectamente

Problemas

- **No es posible instanciar nuevos datos:**
  - ✓ ¿Qué ocurre si necesito más de 1 pila?
- **Solución: si un módulo intenta representar un T.A.D. → que el lenguaje permita definir tipos de datos con esa estructura:**

  Objetos = Datos + Operaciones

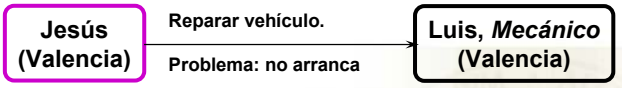

```
TIPO pila
  ELEMENTOS PÚBLICOS:
    int pop ()
    { /* algoritmo que usa mis_datos */ }

    void push (int x)
    { /* algoritmo que usa mis_datos */ }

  ELEMENTOS PRIVADOS:
    int mis_datos[100];
```

```
/* Programa que usa PILAS */
int main()
{
      int i;
      pila p, q;

      for (i=0; i<100; i++) p.push(i);
      for (i=100; i>0; i--) q.push(i);

      for (i=0; i<100; i++)
        printf ("%d %d", p.pop(), q.pop());

      return 0;
}
```

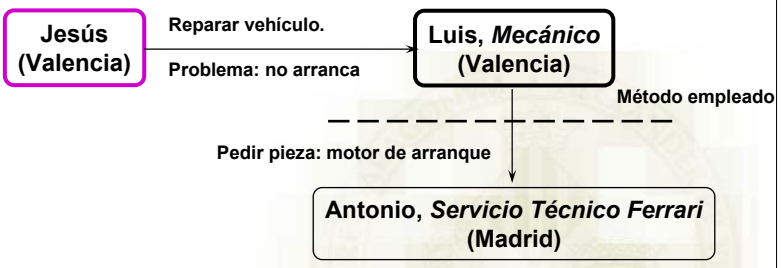
Procedural vs. Orientación a Objetos

- **Programación procedural:**
  - ✓ Interés en la descomposición en subrutinas (funcionalidad).
  - ✓ Más inestable → Depende de los cambios en los requisitos funcionales.
- **Programación Orientada a Objetos:**
  - ✓ Interés en la organización basada en los datos.
  - ✓ Más estable y robusta frente a cambios en los requisitos funcionales.

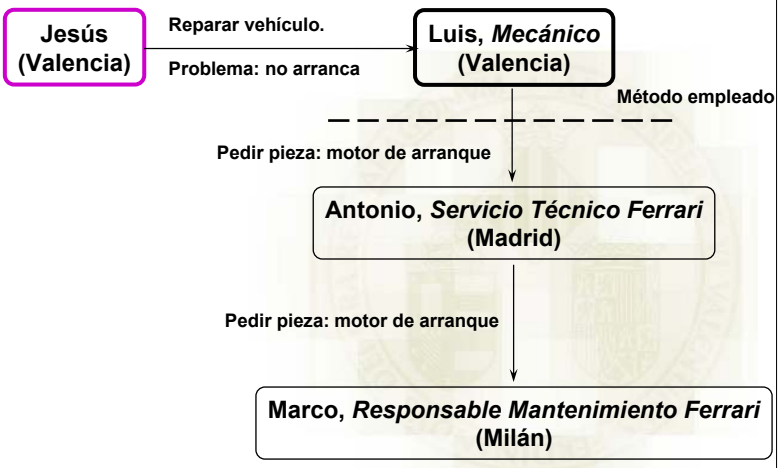
Problema: Reparar el coche

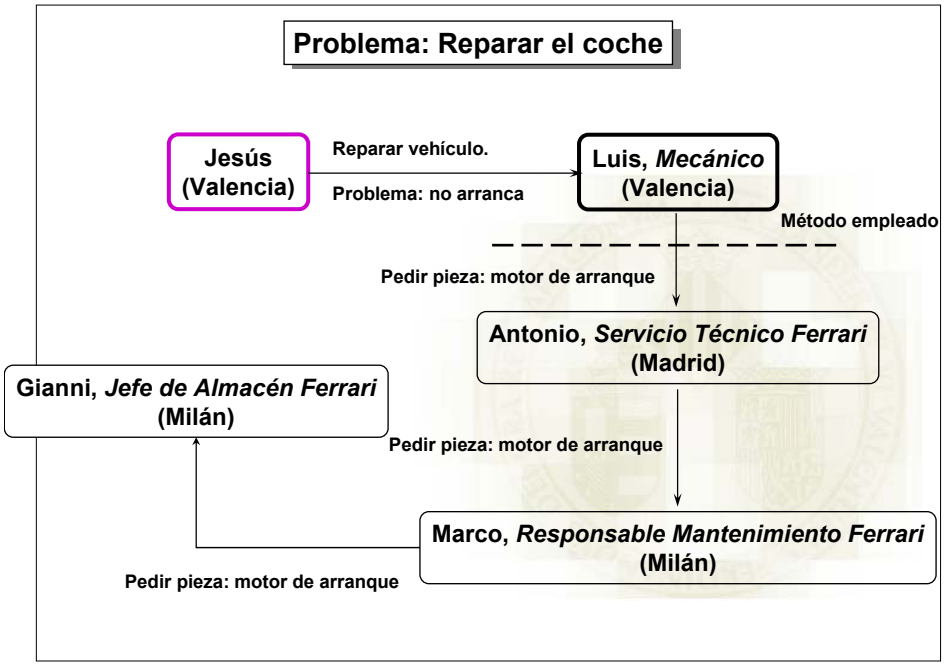


Problema: Reparar el coche



Problema: Reparar el coche





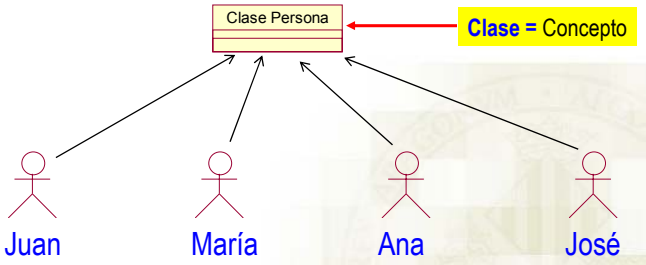
Mensajes y Métodos

- En POO, la acción se inicia mediante la transmisión de un mensaje a un agente (objeto) responsable de la acción.
- El mensaje tiene codificada la petición de una acción y puede contener información adicional (argumentos) para realizarla.
- El receptor es el agente al cual se envía el mensaje. Si el receptor acepta el mensaje, está aceptando la responsabilidad de llevar a cabo la acción indicada.
- En respuesta al mensaje, el receptor ejecutará algún método para satisfacer la petición.

Clases y Objetos

- Todos los objetos son ejemplares de una clase.
- El método aplicado por un objeto en respuesta a un mensaje queda determinado por la clase del receptor.
- Todos los objetos de una clase usan el mismo método en respuesta a mensajes similares.
- Objetos de distinta clase pueden responder al mismo mensaje, aunque aplicando distintos métodos (polimorfismo).

Clases y Objetos (2)



Objetos = Ejemplos concretos de la clase que responden al concepto.

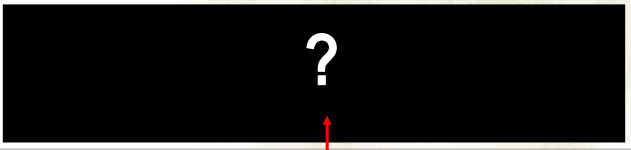
Ejemplo: Concepto de Silla

SILLA

Métodos (Acciones, Operaciones):

Cómo se puede usar una Silla

	Entrada	
• *sentar*	**nuevo_Peso**	//"sienta" nuevo_Peso en la silla
• *levantar*	**viejo_Peso**	//"levanta" viejo_Peso de la silla
• *dibujar*	**(nada)**	// dibuja la silla en pantalla



Cómo se ha fabricado la silla:
- ✓ Depende del fabricante
- ✓ El usuario no puede cambiar estas características

Ejemplo: Concepto de Silla

SILLA

Métodos (Acciones, Operaciones):

Cómo se puede usar una Silla

	Entrada	
• *sentar*	**nuevo_Peso**	//"sienta" nuevo_Peso en la silla
• *levantar*	**viejo_Peso**	//"levanta" viejo_Peso de la silla
• *dibujar*	**(nada)**	// dibuja la silla en pantalla

Información (Atributos):

• *color*	//color de la silla
• *estilo*	//tipo de la silla
• *carga_máxima*	//carga máxima que puede soportar la silla
• *está_rota*	//indica si la silla está rota o no
• *carga_actual*	//carga que está soportando la silla

Ejemplo Java: Clase Silla

```
class Silla {
  public void sentar ( int nuevo_Peso ) {
    /* ... */
  }
  public void levantar ( int viejo_Peso ) {
    /* ... */
  }
  public void dibujar ( ) {
    /* ... */
  }

  private String color;
  private String estilo;
  private int carga_maxima;

  private boolean esta_rota;
  private int carga_actual;
}
```


Java: Crear Objetos

- **Los objetos se declaran en *Java* con la misma sintaxis que las variables en *C/C++*:**
  - ✓ *C/C++*: `<tipo> <id. variable>` → `int x;`
  - ✓ *Java*: `<clase> <id. objeto>` → `Silla x;`

Java: Crear Objetos

- **Los objetos se declaran en *Java* con la misma sintaxis que las variables en *C*:**
  - ✓ *C/C++*: `<tipo> <id. variable>` → `int x;`
  - ✓ *Java*: `<clase> <id. objeto>` → `Silla x;`

- **Atención: `x` es sólo un identificador que puede referenciar a un objeto de la clase `Silla` pero la declaración anterior no le asocia ningún objeto:**
  - ✓ La declaración de `x` NO crea ninguna silla.

```
x.sentar(56); //NO es posible, no existe ninguna silla.
```


Java: Crear Objetos (2)

- **Java maneja referencias a objetos.**
- **La declaración `Silla x;` está indicando que `x` es una referencia que en algún momento, no ahora, permitirá trabajar con un objeto de la clase `Silla`.**
- **El objeto debe crearse explícitamente cuando sea necesario mediante la sentencia `new`:**

```
x = new Silla(); //Ahora SÍ que hay una Silla
x.sentar(56); //SÍ es posible.
```

Java: Manejo de Objetos

```
Silla mi_silla, otra_silla; //declaración

mi_silla = new Silla(); //creación objeto
mi_silla.sentar(80);
mi_silla.dibujar();

otra_silla = new Silla(); //creación objeto
otra_silla.sentar(90);
otra_silla.dibujar();
```


Constructores de Objetos

Constructor

```
class Silla {
  public Silla ( ) {
    /* ... */
  }
  public void sentar ( int nuevo_Peso ) {
    /* ... */
  }
  public void levantar ( int viejo_Peso ) {
    /* ... */
  }
  public void dibujar ( ) {
    /* ... */
  }

  private String color;
  private String estilo;
  private int carga_maxima;

  private boolean esta_rota;
  private int carga_actual;
}
```

Constructores de Objetos (2)

```
class Silla {

   public Silla ( ) {
      color = new String ("negro");
      estilo = new String ("despacho");
      carga_maxima = 100;
      esta_rota = false;
      carga_actual = 0;
   }

/* …El resto de los elementos de la clase */
}
```

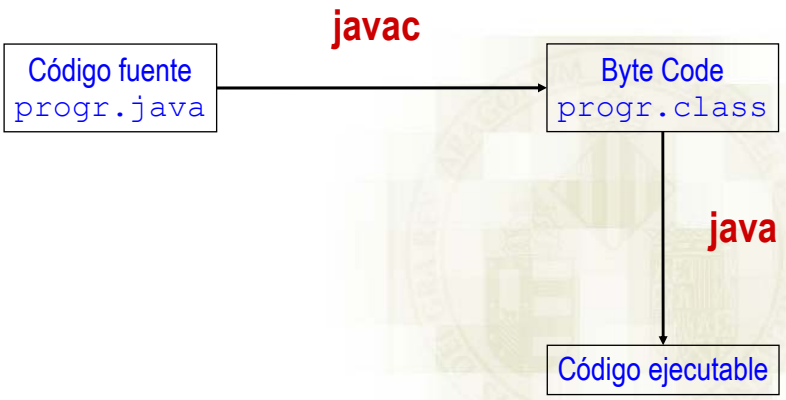

Constructores de Objetos (3)

```
class Silla {

   public Silla (String c, String e, int cg) {
      color = c;
      estilo = e;
      carga_maxima = cg;
      esta_rota = false;
      carga_actual = 0;
   }

/* …El resto de los elementos de la clase */
}
```

Utilización:

```
Silla mi_silla; //declaración
mi_silla = new Silla("blanco", "cocina", 125); //creación objeto
```

Java

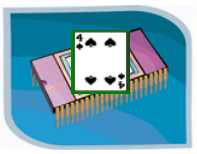


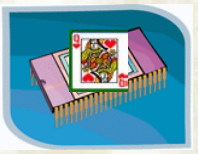
Ejemplo

```
class Carta {
   private int numero; // 1..12
   private char palo; // 'O', 'C', 'E', 'B'

   public Carta ( char p, int n ) {
      palo = p;
      numero = n;
   }

   public int obtenerNumero ( ) {
      return ( numero );
   }

   public char obtenerPalo ( ) {
      return ( palo );
   }

   public void ponerNumero ( int n ) {
      numero = n;
   }

   public void ponerPalo ( char p ) {
      palo = p;
   }

}
```

Ejemplo

```
class Carta {
  public Carta ( char p, int n )
  public int obtenerNumero ( )
  public char obtenerPalo ( )
  public void ponerNumero ( int n )
  public void ponerPalo ( char p )
}
```



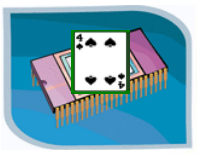


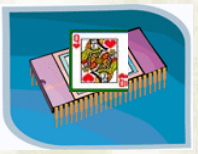
Ejemplo

```
class Carta { //Version 2
  private int codigo;
  // 1..12 = oros, 13..24 = copas, etc…

  public Carta ( char p, int n ) {
    if ( p == 'O' ) codigo = n;
    else
      if ( p == 'C' ) codigo = 12 + n;
      else
        if ( p == 'E' ) codigo =24 + n;
        else codigo = 36 + n;
  }

  public int obtenerNumero ( ) {
    if ( p == 'O' ) return ( n );
    else
      if ( p == 'C' ) return ( n - 12 );
      else
        if ( p == 'E' ) return ( n – 24 );
        else return ( n -36 );
  }

  public char obtenerPalo ( ) { //…}

  public void ponerNumero ( int n ) { //…}

  public void ponerPalo ( char p ) { //…}
}
```

Ejemplo

```
class Carta {
  public Carta ( char p, int n )
  public int obtenerNumero ( )
  public char obtenerPalo ( )
  public void ponerNumero ( int n )
  public void ponerPalo ( char p )
}
```





Ejercicio

- **Construir una clase Pila para representar el tipo de datos Pila, caracterizado por las operaciones:**
  - ✓ Apilar
  - ✓ Desapilar
  - ✓ Cima
  - ✓ esVacia
  - ✓ Para simplificar, consideremos que se almacenan datos enteros en la pila.

Ejercicio

```
class Pila {
   public Pila () {
      max = 100;
      tope = -1;
      datos = new int [max];
   }
   public void Apilar (int x){
      if ( tope < (max -1) ) {
         tope ++;
         datos [tope] = x;
      }
   }
   public void Desapilar () {
      if ( ! esVacia() ) tope --;
   }
   public int Cima () {
      if ( ! esVacia() ) return (datos [tope]);
   }
   public boolean esVacia ( ) {
      if ( tope < 0 ) return (true);
      else return(false);
   }

   private int[]  datos;  //array para almacenar datos
   private final int max; //nº maximo de elementos
   private int  tope;     //hasta donde hay datos en el array
}
```


Ejercicio

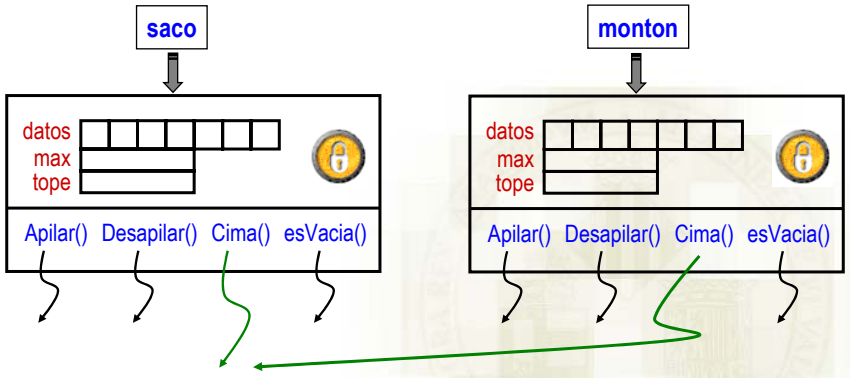
```
class miPrograma {

   public static void main (String[] args) {
      Pila saco, monton;

      saco = new Pila ();
      for (int i=0; i<25; i++)
         saco.Apilar(i+10);

      monton = new Pila ();
      for (int i=25; i<100; i++)
         monton.Apilar(i);

      System.out.println ("Contenido del saco:");
      visualizar (saco);
      System.out.println ("Contenido del monton:");
      visualizar (monton);
   }

   private static void visualizar (Pila p) {
      while ( ! p.esVacia() ) {
         System.out.println ( p.Cima() );
         p.Desapilar();
      }
   }

}
```

Programa principal

Ejercicio

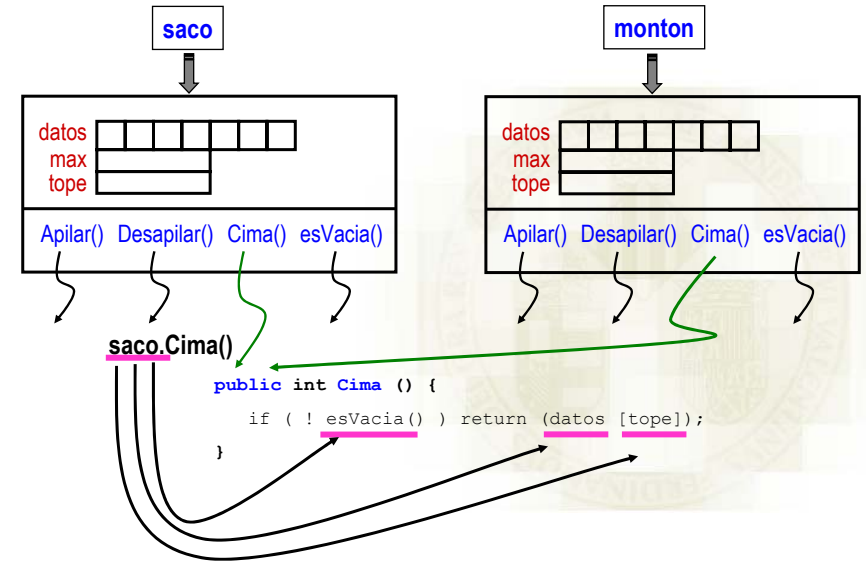
```
class Pila { //Versión con 2 constructores
   public Pila () {
      max = 100;
      tope = -1;
      datos = new int [max];
   }
   public Pila (int tamano) {
      max = tamano;
      tope = -1;
      datos = new int [max];
   }

   public void Apilar (int x){
      ...
   }
   public void Desapilar () {
      ...
   }
   public int Cima () {
      ...
   }
   public boolean esVacia ( ) {
      ...
   }
}
```


Representación de objetos

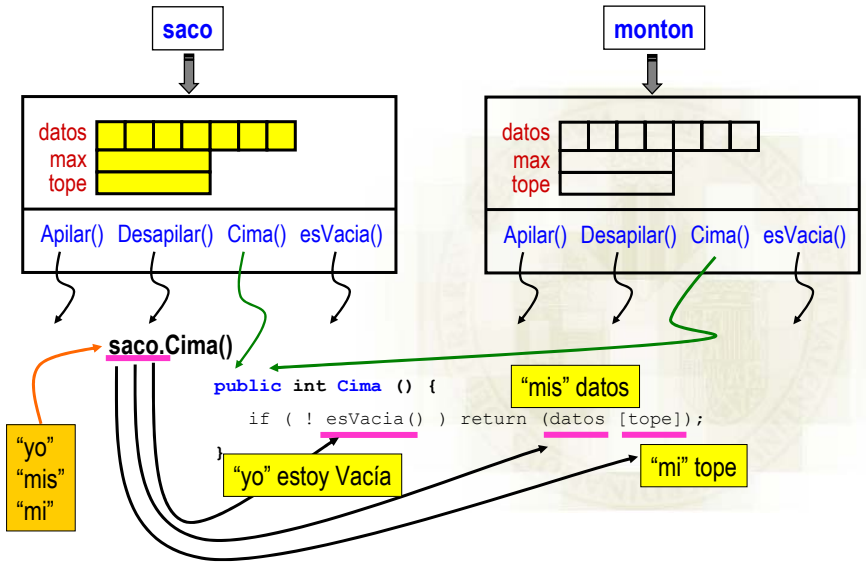



```
public int Cima () {
   if ( ! esVacia() ) return (datos [tope]);
}
```

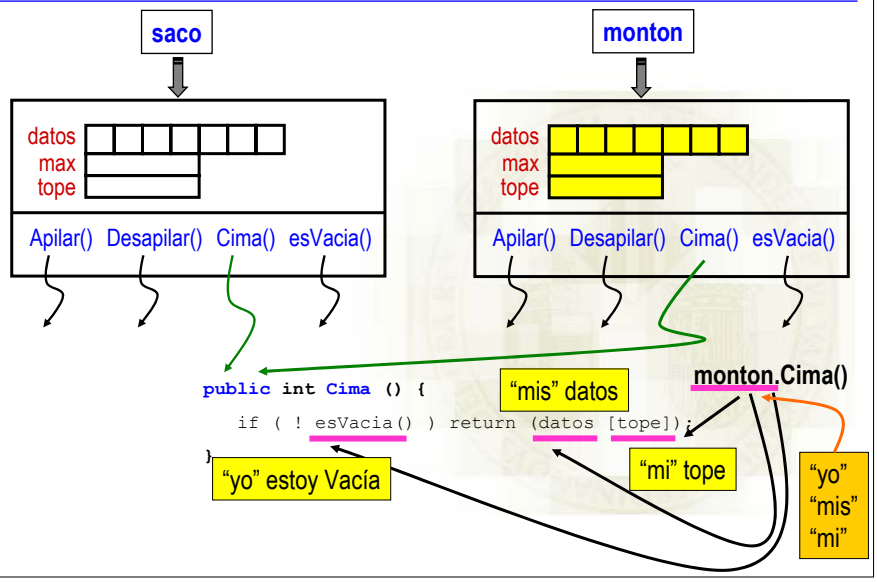
Representación de objetos



Representación de objetos



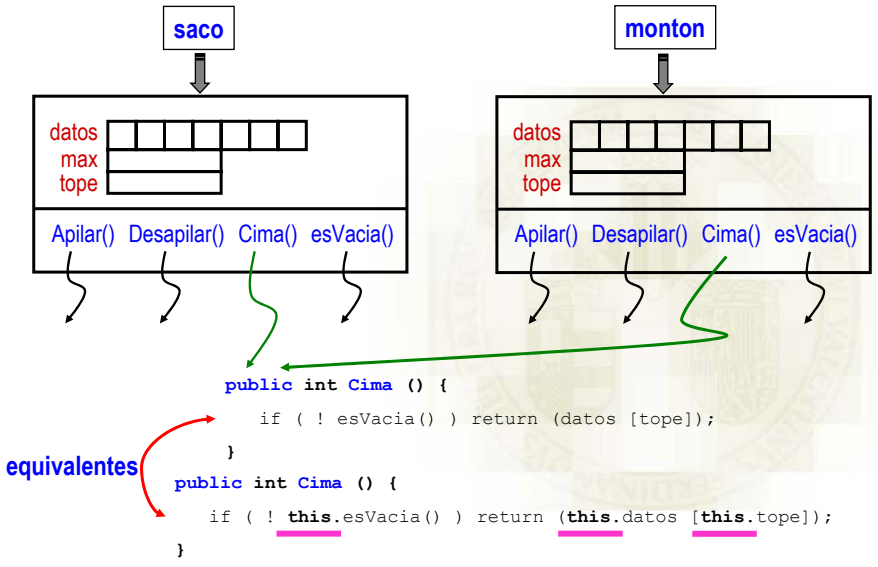
Representación de objetos



this

- La referencia "`this`" (éste) permite nombrar al objeto que recibe el mensaje que se está realizando.
- "`this`" equivale al sujeto "yo"
- "`this`" es una referencia constante, no puede asignarse a otro objeto.
  - ✓ "yo soy yo, no puedo ser otro"

this (Ejemplo)



Ejercicio

```
class Pila { //Versión con 2 constructores

    public Pila () {
        this (100);
    }

    public Pila (int tamano) {
        max = tamano;
        tope = -1;
        datos = new int [max];
    }

        ...

}
```

Ejercicio

```
class Pila { //Versión con 2 constructores

   public Pila () {
      this (100);
   }

   public Pila (int tamano) {
      max = tamano;
      tope = -1;
      datos = new int [max];
   }

      ...

}
```

?

```
public Pila () {
   this = new Pila (100);
}
```


Ejercicio

```
class Pila { //Versión con 2 constructores

   public Pila () {
      this (100);
   }

   public Pila (int tamano) {
      max = tamano;
      tope = -1;
      datos = new int [max];
   }

      ...

}
```

X

```
public Pila () {
   this = new Pila (100);
}
```

No se puede,
`this` es una referencia constante.
No puede asignarse.