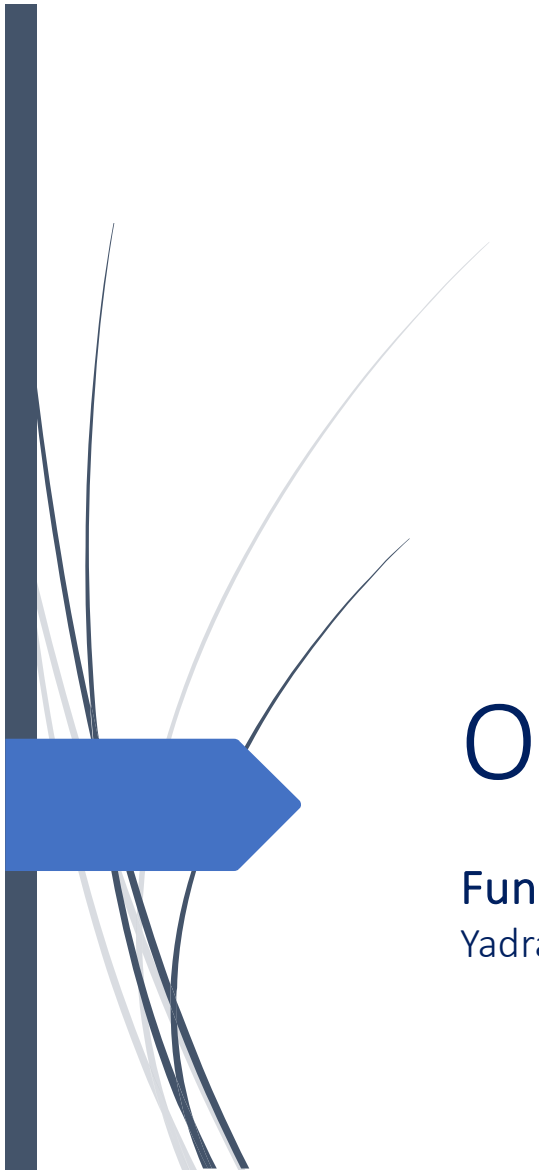


Orientación a objetos

Fundamentos de Lenguajes de Programación – IIC2560

Yadran Eterovic S. (yadran@uc.cl)

```
type Int_Stack = struct{
                    int P[100];  // the stack proper
                    int top;     // first readable element
}
Int_Stack create_stack(){
   Int_Stack s = new Int_Stack();
   s.top = 0;
   return s;
}
Int_Stack push(Int_Stack s, int k){
   if (s.top == 100) error;
   s.P[s.top] = k;
   s.top = s.top + 1;
   return s;
}
int top(Int_Stack s){
   return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
   if (s.top == 0) error;
   s.top = s.top - 1;
   return s;
}
bool empty(Int_Stack s){
   return (s.top == 0);
}
```

```
type Int_Stack = struct{
                    int P[100];   //
                    int top;      //
}
Int_Stack create_stack(){
   Int_Stack s = new Int_Stack();
   s.top = 0;
   return s;
}
Int_Stack push(Int_Stack s, int k){
   if (s.top == 100) error;
   s.P[s.top] = k;
   s.top = s.top + 1;
   return s;
}
int top(Int_Stack s){
   return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
   if (s.top == 0) error;
   s.top = s.top - 1;
   return s;
}
bool empty(Int_Stack s){
   return (s.top == 0);
}
```

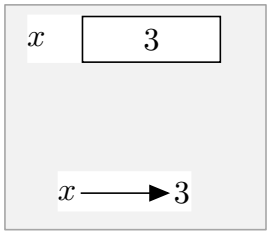
Únicas operaciones definidas por el programador

Definición del tipo de datos `Int_Stack` (variables en el modelo de referencia)

El propósito es que un stack de enteros sea una estructura manejada por sólo 5 operaciones

Variable modificable

Variable en el modelo de referencia



```
type Int_Stack = struct{
                    int P[100];   //
                    int top;      //
}
Int_Stack create_stack(){
   Int_Stack s = new Int_Stack();
   s.top = 0;
   return s;
}
Int_Stack push(Int_Stack s, int k){
   if (s.top == 100) error;
   s.P[s.top] = k;
   s.top = s.top + 1;
   return s;
}
int top(Int_Stack s){
   return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
   if (s.top == 0) error;
   s.top = s.top - 1;
   return s;
}
bool empty(Int_Stack s){
   return (s.top == 0);
}
```

Únicas operaciones definidas por el programador

Definición del tipo de datos `Int_Stack` (variables en el modelo de referencia)

El propósito es que un stack de enteros sea una estructura manejable por sólo 5 operaciones

Pero el lenguaje no garantiza que éstas sean las únicas operaciones posibles:

- p.ej., nada previene que alguien defina …

```
int second_from_top(Int_Stack c){
    return c.P[c.top-1];
}
```

```
type Int_Stack = struct{
                    int P[100];   //
                    int top;      //
}
Int_Stack create_stack(){
   Int_Stack s = new Int_Stack();
   s.top = 0;
   return s;
}
Int_Stack push(Int_Stack s, int k){
   if (s.top == 100) error;
   s.P[s.top] = k;
   s.top = s.top + 1;
   return s;
}
int top(Int_Stack s){
   return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
   if (s.top == 0) error;
   s.top = s.top - 1;
   return s;
}
bool empty(Int_Stack s){
   return (s.top == 0);
}
```

Únicas operaciones definidas por el programador

Definición del tipo de datos `Int_Stack` (variables en el modelo de referencia)

El propósito es que un stack de enteros sea una estructura manejable por sólo 5 operaciones

Pero el lenguaje no garantiza que éstas sean las únicas operaciones posibles:

- p.ej., nada previene que alguien defina …

```
int second_from_top(Int_Stack c){
    return c.P[c.top-1];
}
```

Los lenguajes tienen tipos de datos predefinidos que ocultan sus implementaciones

… pero los programadores no pueden hacer lo mismo

"abstype" introduce un TDA

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int P[100];
                        int n;
                        int top;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;
         s.n = s.n - 1;
         s.top = s.top - 1;
         return s;
      }
      bool empty(Int_Stack s){
         return (s.n == 0);
      }
}
```

Hay lenguajes que permiten la definición de abstracciones de datos

… que se comportan como los tipos pre-definidos en cuanto a la (in)accesibilidad de las representaciones

→ **tipo de datos abstracto (TDA)**

Nombre del TDA

Firma o interfaz:
Nombres y tipos de las operaciones para manejar un `Int_Stack`

Implementación:
Para cada operación, una implementación que usa la representación o tipo concreto

Implementación o representación o *tipo concreto*

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int P[100];
                        int n;
                        int top;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;
         s.n = s.n - 1;
         s.top = s.top - 1;
         return s;
      }
      bool empty(Int_Stack s){
         return (s.n == 0);
      }
}
```

Hay lenguajes que permiten la definición de abstracciones de datos

… que se comportan como los tipos pre-definidos en cuanto a la (in)accesibilidad de las representaciones

→ **tipo de datos abstracto** (**TDA**)

Nombre del TDA

Firma o *interfaz*:
Nombres y tipos de las operaciones para manejar un `Int_Stack`

Implementación:
Para cada operación, una implementación que usa la representación o tipo concreto

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int P[100];
                        int n;
                        int top;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;
         s.n = s.n - 1;
         s.top = s.top - 1;
         return s;
      }
      bool empty(Int_Stack s){
         return (s.n == 0);
      }
}
```

Implementación o representación o *tipo concreto*

Un TDA es una cápsula opaca:

- en la superficie exterior, visible para todos, están el nombre del nuevo tipo y los nombres y tipos de las operaciones —*firma* (*signature*) o *interfaz*
- dentro, invisibles para el mundo exterior, están las *implementaciones* del tipo y de sus operaciones
- el acceso al interior es controlado por la cápsula que garantiza la consistencia de la información que encierra

Nombre del TDA

Firma o interfaz:
Nombres y tipos de las operaciones para manejar un `Int_Stack`

Implementación:
Para cada operación, una implementación que usa la representación o tipo concreto

```
abstype Int_Stack{
   type Int_Stack = struct{
                       int P[100];
                       int n;
                       int top;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;
         s.n = s.n - 1;
         s.top = s.top - 1;
         return s;
      }
      bool empty(Int_Stack s){
         return (s.n == 0);
      }
}
```

Implementación o representación o *tipo concreto*

Dentro de la declaración, `Int_Stack` es sinónimo de su representación concreta:

- las operaciones manejan un stack como una `struct` con un array y dos enteros

Fuera de la declaración —en el resto del programa— no hay ninguna relación entre un `Int_Stack` y su tipo concreto:

- la única forma de manejar un `Int_Stack` son sus operaciones

P.ej., la función `second_from_top()` anterior, ahora es *imposible*

→ el chequeo de tipos no permite la aplicación de un selector de campo (`c.P`, `c.top`) a un `Int_Stack` (ya que `Int_Stack` no es una *struct* fuera de su declaración)

La separación entre interfaz e implementación es una separación entre el uso de un TDA y su implementación

… generalizando las abstracciones de control:

- ahora no sólo está oculto cómo se implementa una operación

… sino también cómo se representan los datos

La separación entre interfaz e implementación es una separación entre el uso de un TDA y su implementación

… generalizando las abstracciones de control: → Subprogramas

- ahora no sólo está oculto cómo se implementa una operación

… sino también cómo se representan los datos

La separación entre interfaz e implementación es una separación entre el uso de un TDA y su implementación

… generalizando las abstracciones de control:

- ahora no sólo está oculto cómo se implementa una operación

… sino también cómo se representan los datos

Subprogramas

Ocultación de información

La separación entre interfaz e implementación es una separación entre el uso de un TDA y su implementación

… generalizando las abstracciones de control: → Subprogramas

- ahora no sólo está oculto cómo se implementa una operación

… sino también cómo se representan los datos

Ocultación de información

Es posible sustituir la implementación de un ADT por otra implementación del mismo TDA

… manteniendo la misma interfaz

… y no debería haber ningún efecto observable en los programas que usan el TDA

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int P[100];
                        int n;
                        int top;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;
         s.n = s.n - 1;
         s.top = s.top - 1;
         return s;
      }
      bool empty(Int_Stack s){
         return (s.n == 0);
      }
}
```

P.ej., aquí está nuevamente la definición que ya vimos del TDA `Int_Stack`

... y aquí una nueva definición que usa otra implementación

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int info;
                        Int_stack next;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         return null;
      }
      Int_Stack push(Int_Stack s, int k){
         Int_Stack tmp = new Int_Stack();
         tmp.info = k;
         tmp.next = s;
         return tmp;
      }
      int top(Int_Stack s){
         return s.info;
      }
      Int_Stack pop(Int_Stack s){
         return s.next;
      }
      bool empty(Int_Stack s){
         return (s == null);
      }
}
```

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int P[100];
                        int n;
                        int top;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;
         s.n = s.n - 1;
         s.top = s.top - 1;
         return s;
      }
      bool empty(Int_Stack s){
         return (s.n == 0);
      }
}
```

Especificación del ADT `Int_Stack`

`create_stack` crea un stack vacío

`push` inserta un elemento en el stack

`top` devuelve el elemento en el tope del stack, sin modificar el stack; el stack no debe estar vacío

`pop` saca el elemento en el tope del stack; el stack no debe estar vacío

`empty` es verdadeo si y sólo si el stack está vacío

Descripción de la semántica de las operaciones del ADT

… en términos de relaciones abstractas generales

… y no en términos de tipos concretos

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int info;
                        Int_stack next;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         return null;
      }
      Int_Stack push(Int_Stack s, int k){
         Int_Stack tmp = new Int_Stack();
         tmp.info = k;
         tmp.next = s;
         return tmp;
      }
      int top(Int_Stack s){
         return s.info;
      }
      Int_Stack pop(Int_Stack s){
         return s.next;
      }
      bool empty(Int_Stack s){
         return (s == null);
      }
}
```

Un TDA Counter

```
abstype Counter{
   type Counter = int;
   signature
      void reset(Counter x);
      int get(Counter x);
      void inc(Counter x);
   operations
      void reset(Counter x){
         x = 0;
      }
      int get(Counter c){
         return x;
      }
      void inc(Counter c){
         x = x+1;
      }
}
```

Un nuevo TDA `NewCounter1`

```
abstype NewCounter1{
   type NewCounter1 = struct{
                        int c;
                        int num_reset = 0;
                      }
   signature
      void reset(NewCounter1 x);
      int get(NewCounter1 x);
      void inc(NewCounter1 x);
      int howmany_resets(NewCounter1 x);
   operations

      void reset(NewCounter1 x){
         x.c = 0;
         x.num_reset = x.num_reset+1;
      }
      int get(NewCounter1 x){
         return x.c;
      }
      void inc(NewCounter1 x){
         x.c = x.c+1;
      }
      int howmany_resets(NewCounter1 x){
         return x.num_reset;
      }
}
```

`NewCounter1` es una versión "enriquecida" de `Counter`:

- agrega una función que cuenta el número de veces que el contador ha sido "reseteado"

Un nuevo TDA `NewCounter1`

```
abstype NewCounter1{
   type NewCounter1 = struct{
                        int c;
                        int num_reset = 0;
                      }
   signature
      void reset(NewCounter1 x);
      int get(NewCounter1 x);
      void inc(NewCounter1 x);
      int howmany_resets(NewCounter1 x);
   operations

      void reset(NewCounter1 x){
         x.c = 0;
         x.num_reset = x.num_reset+1;
      }
      int get(NewCounter1 x){
         return x.c;
      }
      void inc(NewCounter1 x){
         x.c = x.c+1;
      }
      int howmany_resets(NewCounter1 x){
         return x.num_reset;
      }
}
```

`NewCounter1` es una versión "enriquecida" de `Counter`:

- agrega una función que cuenta el número de veces que el contador ha sido "reseteado"

Pero

- requiere redefinir todas las operaciones de `Counter` —mismos nombres *overloaded**, diferentes tipos de argumentos
- no tiene ninguna relación con `Counter` —cualquier cambio a `Counter` (e.g., corrección de un *bug*, un mejor algoritmo) no tiene efecto en `NewCounter1`, en donde hay que rehacer los cambios a mano

Un nuevo TDA `NewCounter1`

```
abstype NewCounter1{
   type NewCounter1 = struct{
                        int c;
                        int num_reset = 0;
                      }
   signature
      void reset(NewCounter1 x);
      int get(NewCounter1 x);
      void inc(NewCounter1 x);
      int howmany_resets(NewCounter1 x);
   operations

      void reset(NewCounter1 x){
         x.c = 0;
         x.num_reset = x.num_reset+1;
      }
      int get(NewCounter1 x){
         return x.c;
      }
      void inc(NewCounter1 x){
         x.c = x.c+1;
      }
      int howmany_resets(NewCounter1 x){
         return x.num_reset;
      }
}
```

`NewCounter1` es una versión "enriquecida" de `Counter`:

- agrega una función que cuenta el número de veces que el contador ha sido "reseteado"

Pero

- requiere redefinir todas las operaciones de `Counter` —mismos nombres *overloaded**, diferentes tipos de argumentos
- no tiene ninguna relación con `Counter` —cualquier cambio a `Counter` (e.g., corrección de un *bug*, un mejor algoritmo) no tiene efecto en `NewCounter1`, en donde hay que rehacer los cambios a mano

*Un nombre está *overloaded* cuando corresponde a más de un objeto y es necesario usar información de contexto para determinar cuál objeto es denotado por una instancia específica de ese nombre

Un nuevo TDA `NewCounter2`


```
abstype NewCounter2{
   type NewCounter2 = struct{
                        Counter c;
                        int num_reset = 0;
                      }
   signature
      void reset(NewCounter2 x);
      int get(NewCounter2 x);
      void inc(NewCounter2 x);
      int howmany_resets(NewCounter2 x);
   operations
      void reset(NewCounter2 x){
         reset(x.c);
         x.num_reset = x.num_reset+1;
      }
      int get(NewCounter2 x){
         return get(x.c);
      }
      void inc(NewCounter2 x){
         inc(x.c);
      }
      int howmany_resets(NewCounter2 x){
         return x.num_reset;
      }
}
```

`NewCounter2` es otra versión "enriquecida" de `Counter`:

… pero su definición usa `Counter` —mejor que `NewCounter1`:

- las operaciones que se repiten son simplemente llamadas desde dentro de `NewCounter2`

… aunque hay que llamarlas explícitamente, incluso en los casos de `get` e `inc`

¿Qué pasa si queremos "resetear" una colección de contadores, algunos simples, otros enriquecidos?

- la declaración `Counter V[100]` no permite almacenar contadores de tipo `NewCounter2` en `V` (y vice versa)

¿Qué pasa si queremos "resetear" una colección de contadores, algunos simples, otros enriquecidos?

- la declaración `Counter V[100]` no permite almacenar contadores de tipo `NewCounter2` en `V` (y vice versa)

Pero, ¿son `Counter` y `NewCounter2` tipos compatibles?

> `T` es compatible con `S` cuando todas las operaciones sobre valores de tipo `S` también son posibles sobre valores de tipo `T`

Una de las formas de *compatibilidad de tipos*

¿Qué pasa si queremos "resetear" una colección de contadores, algunos simples, otros enriquecidos?

- la declaración `Counter V[100]` no permite almacenar contadores de tipo `NewCounter2` en `V` (y vice versa)

Pero, ¿son `Counter` y `NewCounter2` tipos compatibles?

> `T` es compatible con `S` cuando todas las operaciones sobre valores de tipo `S` también son posibles sobre valores de tipo `T`



Exactamente el caso:

- todas las operaciones en un contador `Counter` son posibles también en un contador `NewCounter2`

Supongamos entonces que la declaración `Counter V[100]` **sí** permite almacenar contadores de tipo `NewCounter2` en `V`

Entonces podemos escribir:

```
for (int i=1; i<100; i=i+1)
    reset(V[i]);
```

Supongamos entonces que la declaración **`Counter V[100]` sí** permite almacenar contadores de tipo **`NewCounter2`** en **`V`**

Entonces podemos escribir:

```
for (int i=1; i<100; i=i+1)
    reset(V[i]);
```

Pero, ¿cuál **`reset`** es este **`reset`**?

```
abstype Counter {
     type Counter = int;
     signature
          ...
     operations
          void reset(Counter x){
               x = 0;
          }
          ...
}
```

```
abstype NewCounter2 {
     type NewCounter2 = struct{
                              Counter c;
                              int num_reset = 0;
                         }
     signature
          ...
     operations
          void reset(NewCounter2 x){
               reset(x.c);
               x.num_reset = x.num_reset+1;
          }
          ...
}
```

Supongamos entonces que la declaración `Counter V[100]` **sí** permite almacenar contadores de tipo `NewCounter2` en `V`

Entonces podemos escribir:

```
for (int i=1; i<100; i=i+1)
    reset(V[i]);
```

Pero, ¿cuál `reset` es este `reset`?

- la solución estática (decidida por el compilador) al *overloading* de `reset` implica que es el `reset` de `Counter`, independiente del tipo de `V[i]`

→ los objetos de tipo `NewCounter2` son "reseteados" correctamente

… pero sus campos `num_reset` no son actualizados

```
abstype Counter{
     type Counter = int;
     signature
          ...
     operations
          void reset(Counter x){
               x = 0;
          }
          ...
}
```

```
abstype NewCounter2{
     type NewCounter2 = struct{
                              Counter c;
                              int num_reset = 0;
                         }
     signature
          ...
     operations
          void reset(NewCounter2 x){
               reset(x.c);
               x.num_reset = x.num_reset+1;
          }
          ...
}
```

```
for (int i=1; i<100; i=i+1)
    reset(V[i]);
```

Requiere la aplicación de la *solución dinámica* al *overloading*:

- decidir en tiempo de ejecución cuál `reset` aplicar

… dependiendo del tipo "efectivo" del argumento

⟶ la asociación del nombre de la operación con su significado debe resolverse en tiempo de ejecución

El código que debe ejecutarse: el de `Counter` o el de `NewCounter2`

`reset`

Así, podemos prever constructos tales que:

- permiten la *encapsulación* y ocultación de información (TDAs)
- tienen mecanismos que permiten la *herencia* de la implementación de operaciones a partir de otras definiciones ya dadas
- están enmarcados en una noción de *compatibilidad* definida en términos de las operaciones admisibles para un cierto constructo
- permiten la *selección dinámica* de operaciones, dependiendo del "tipo efectivo" de los argumentos a los que son aplicadas

Así, podemos prever constructos tales que:

- permiten la *encapsulación* y ocultación de información (TDAs)
- tienen mecanismos que permiten la *herencia* de la implementación de operaciones a partir de otras definiciones ya dadas
- están enmarcados en una noción de *compatibilidad* definida en términos de las operaciones admisibles para un cierto constructo
- permiten la *selección dinámica* de operaciones, dependiendo del "tipo efectivo" de los argumentos a los que son aplicadas



Paradigma orientado a objetos

Objetos

Principal constructo de los lenguajes O-O

Un **objeto** es una cápsula que contiene *datos* y *operaciones* para manejar esos datos

La cápsula ofrece una interfaz al mundo exterior a través de la cual el objeto es accesible

Objetos

Principal constructo de los lenguajes O-O

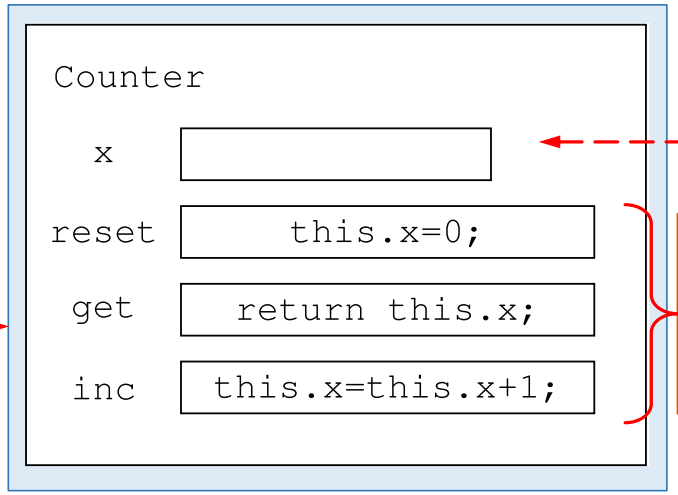
Un **objeto** es una cápsula que contiene *datos* y *operaciones* para manejar esos datos

La cápsula ofrece una interfaz al mundo exterior a través de la cual el objeto es accesible

Cada objeto es un contenedor (al menos conceptualmente) que encapsula tanto datos como operaciones

P.ej., un objeto `Counter`

… que podemos imaginar como un record



Atributo —dato modificable

Operaciones o métodos —que se ejecutan enviando al objeto un *mensaje*: nombre del método + posibles parámetros

Así, podemos prever constructos tales que:

- permiten la *encapsulación* y ocultación de información (TDAs)
- tienen mecanismos que permiten la *herencia* de la implementación de operaciones a partir de otras definiciones ya dadas
- están enmarcados en una noción de *compatibilidad* definida en términos de las operaciones admisibles para un cierto constructo
- permiten la *selección dinámica* de operaciones, dependiendo del "tipo efectivo" de los argumentos a los que son aplicadas



Paradigma orientado a objetos

Objetos

Principal constructo de los lenguajes O-O

Un **objeto** es una cápsula que contiene:

- *datos*
- *operaciones* para manejar esos datos

La cápsula ofrece una interfaz al mundo exterior a través de la cual el objeto es accesible

Un **objeto** es una cápsula que contiene:

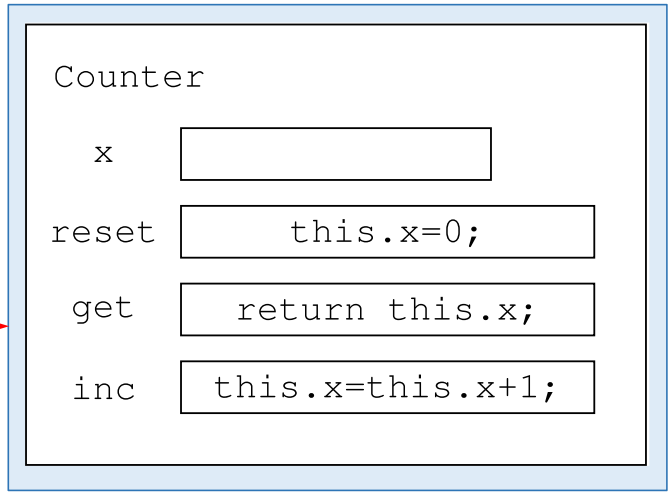
- *datos*
- *operaciones* para manejar esos datos

La cápsula ofrece una interfaz al mundo exterior a través de la cual el objeto es accesible

Cada objeto es un contenedor (al menos conceptualmente) que encapsula tanto datos como operaciones

P.ej., un objeto `Counter`

… que podemos imaginar como un record



Un **objeto** es una cápsula que contiene:

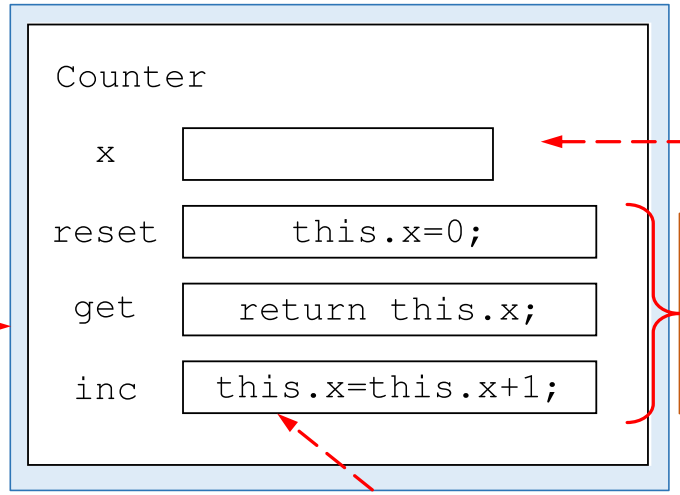
- *datos*
- *operaciones* para manejar esos datos

La cápsula ofrece una interfaz al mundo exterior a través de la cual el objeto es accesible

Cada objeto es un contenedor (al menos conceptualmente) que encapsula tanto datos como operaciones

P.ej., un objeto `Counter`

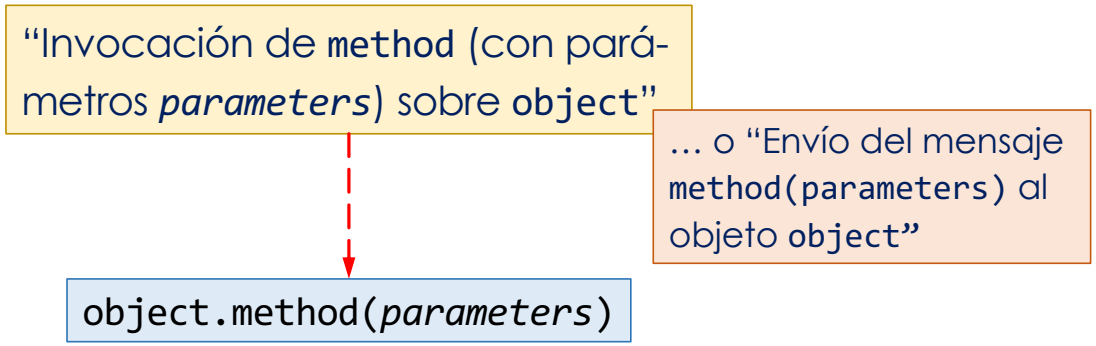
… que podemos imaginar como un record

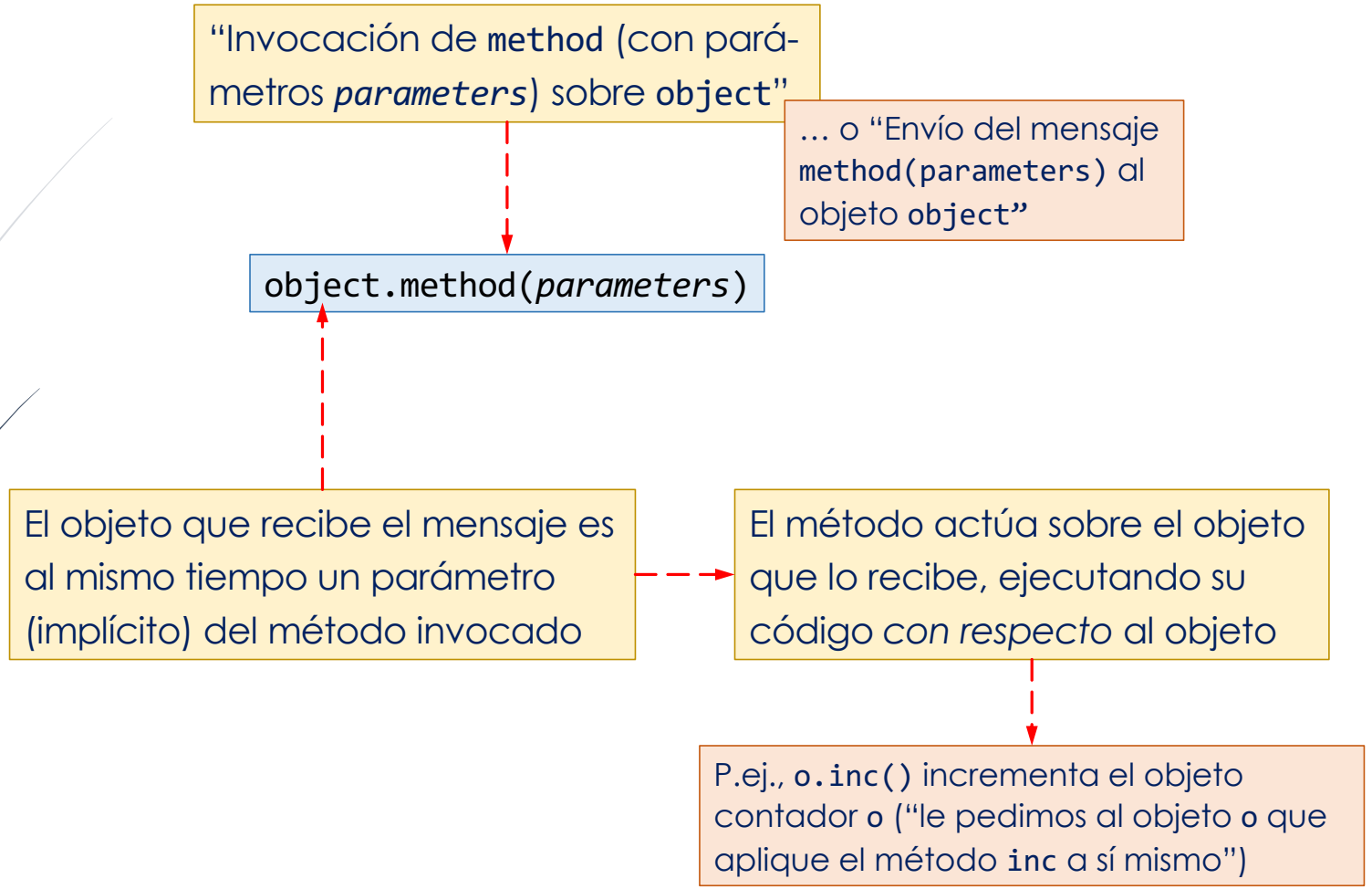


Atributo —dato modificable

Operaciones o métodos —que se ejecutan enviando al objeto un *mensaje*: nombre del método + parámetros

"`this`" (también "`self`") se refiere al objeto que recibe el método


"Invocación de method (con parámetros parameters) sobre object"
... o "Envío del mensaje method(parameters) al objeto object"
object.method(parameters)


"Invocación de method (con parámetros parameters) sobre object"
... o "Envío del mensaje method(parameters) al objeto object"
object.method(parameters)
El objeto que recibe el mensaje es al mismo tiempo un parámetro (implícito) del método invocado
El método actúa sobre el objeto que lo recibe, ejecutando su código con respecto al objeto
P.ej., o.inc() incrementa el objeto contador o ("le pedimos al objeto o que aplique el método inc a sí mismo")

Los lenguajes O-O ofrecen mecanismos para la organización de los objetos

Agrupación de objetos con la misma estuctura (o con estructura similar)

Los lenguajes O-O ofrecen mecanismos para la organización de los objetos

Agrupación de objetos con la misma estuctura (o con estructura similar)

Clases

Si bien es conceptualmente válido que cda objeto contenga sus propios datos y métodos, esto sería un gran desperdicio:

- en un mismo programa, se van a usar muchos objetos que se diferencian entre ellos sólo por los valores de sus datos y no por su estructura o por sus métodos

Una **clase** es un modelo para un conjunto de objetos:

- cada objeto pertenece a una clase
- los objetos son creados dinámicamente mediante *instanciación* de sus clases
- en C++, Java y Python, las clases corresponden a un tipo de datos

Una **clase** es un modelo para un conjunto de objetos:

- cada objeto pertenece a una clase (al menos)
- los objetos son creados dinámicamente mediante *instanciación* de sus clases

El objeto es una *instancia* de la clase

- en C++, Java y Python, las clases corresponden a un tipo de datos

Todos los objetos que instancian una clase *A* son valores de tipo *A*

Una **clase** es un modelo para un conjunto de objetos:

- cada objeto pertenece a una clase
- los objetos son creados dinámicamente mediante *instanciación* de sus clases
- en C++, Java y Python, las clases corresponden a un tipo de datos

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

Una **clase** es un modelo para un conjunto de objetos:

- cada objeto pertenece a una clase
- los objetos son creados dinámicamente mediante *instanciación* de sus clases
- en C++, Java y Python, las clases corresponden a un tipo de datos

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

Tipo y *visibilidad* de los atributos

Nombre, firma, *visibilidad* e implementación de cada método

Una **clase** es un modelo para un conjunto de objetos:

- cada objeto pertenece a una clase
- los objetos son creados dinámicamente mediante *instanciación* de sus clases
- en C++, Java y Python, las clases corresponden a un tipo de datos

`private` : inaccesible desde fuera del objeto; accesible en la implementación de los métodos

`public` : visible para todos

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

Tipo y *visibilidad* de los atributos

Nombre, firma, *visibilidad* e implementación de cada método

Una **clase** es un modelo para un conjunto de objetos:

- cada objeto pertenece a una clase
- los objetos son creados dinámicamente mediante *instanciación* de sus clases
- en C++, Java y Python, las clases corresponden a un tipo de datos

`private` : inaccesible desde fuera del objeto; accesible en la implementación de los métodos

`public` : visible para todos

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

Tipo y *visibilidad* de los atributos

Nombre, firma, *visibilidad* e implementación de cada método

`private` : inaccesible desde fuera del objeto; accesible en la implementación de los métodos

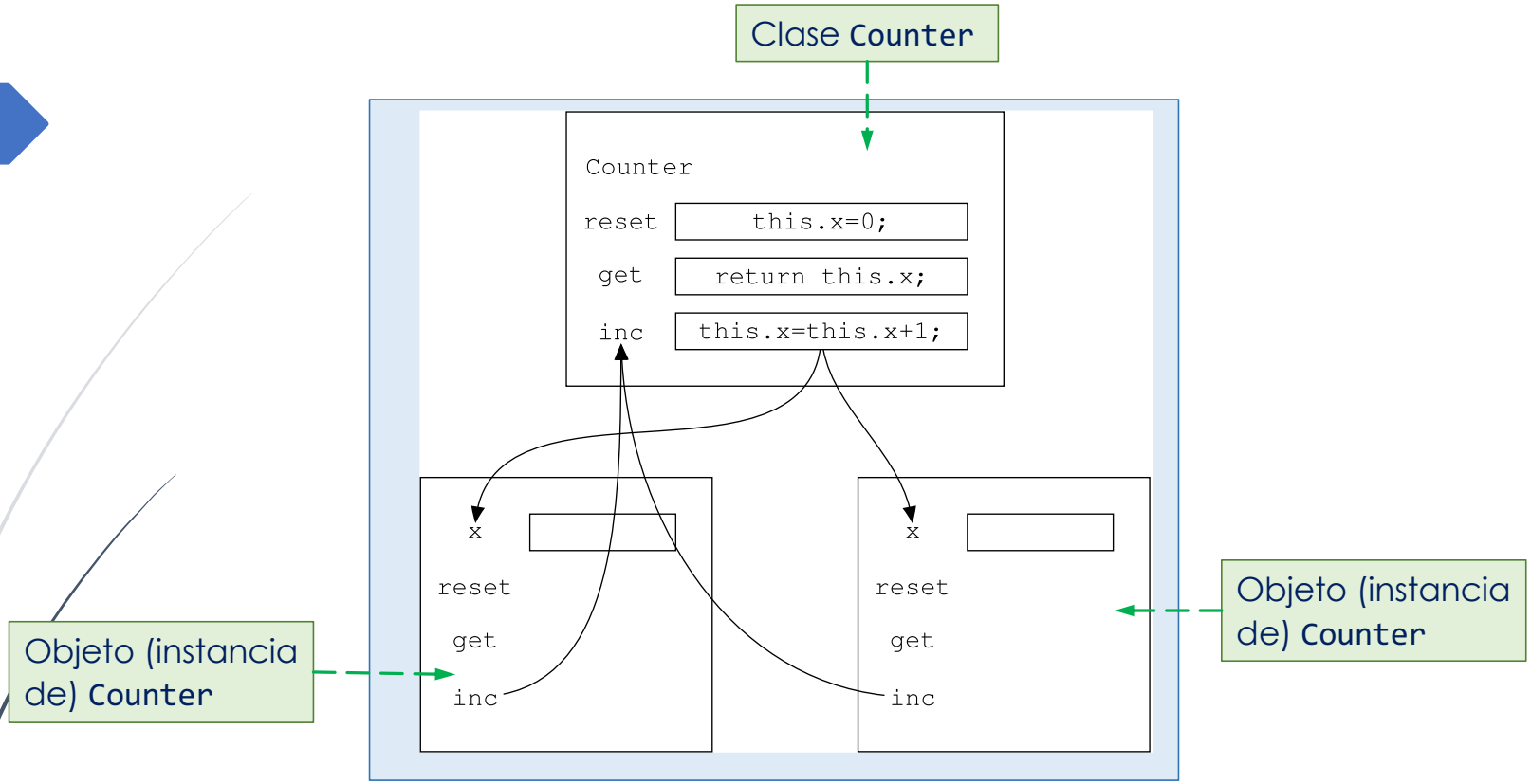
`public` : visible para todos

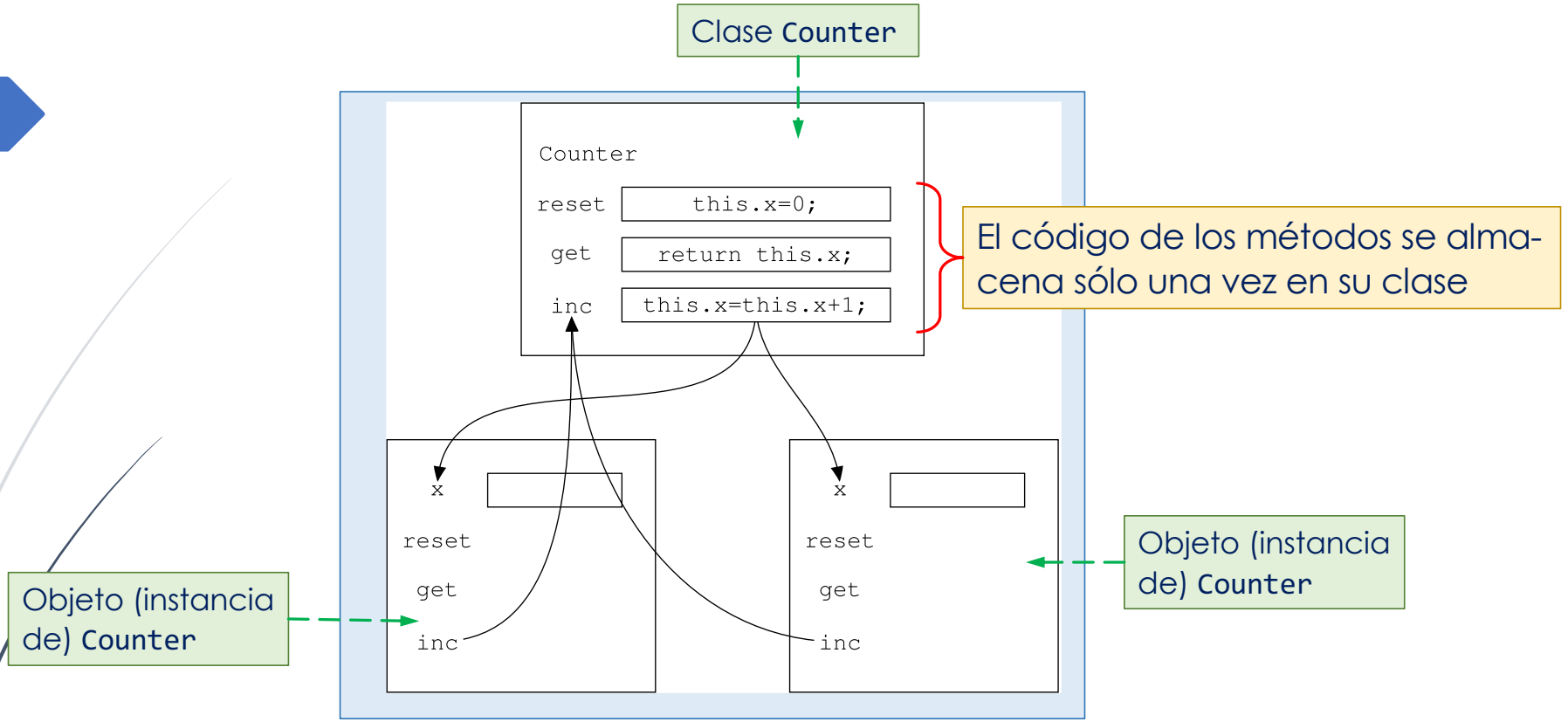
```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

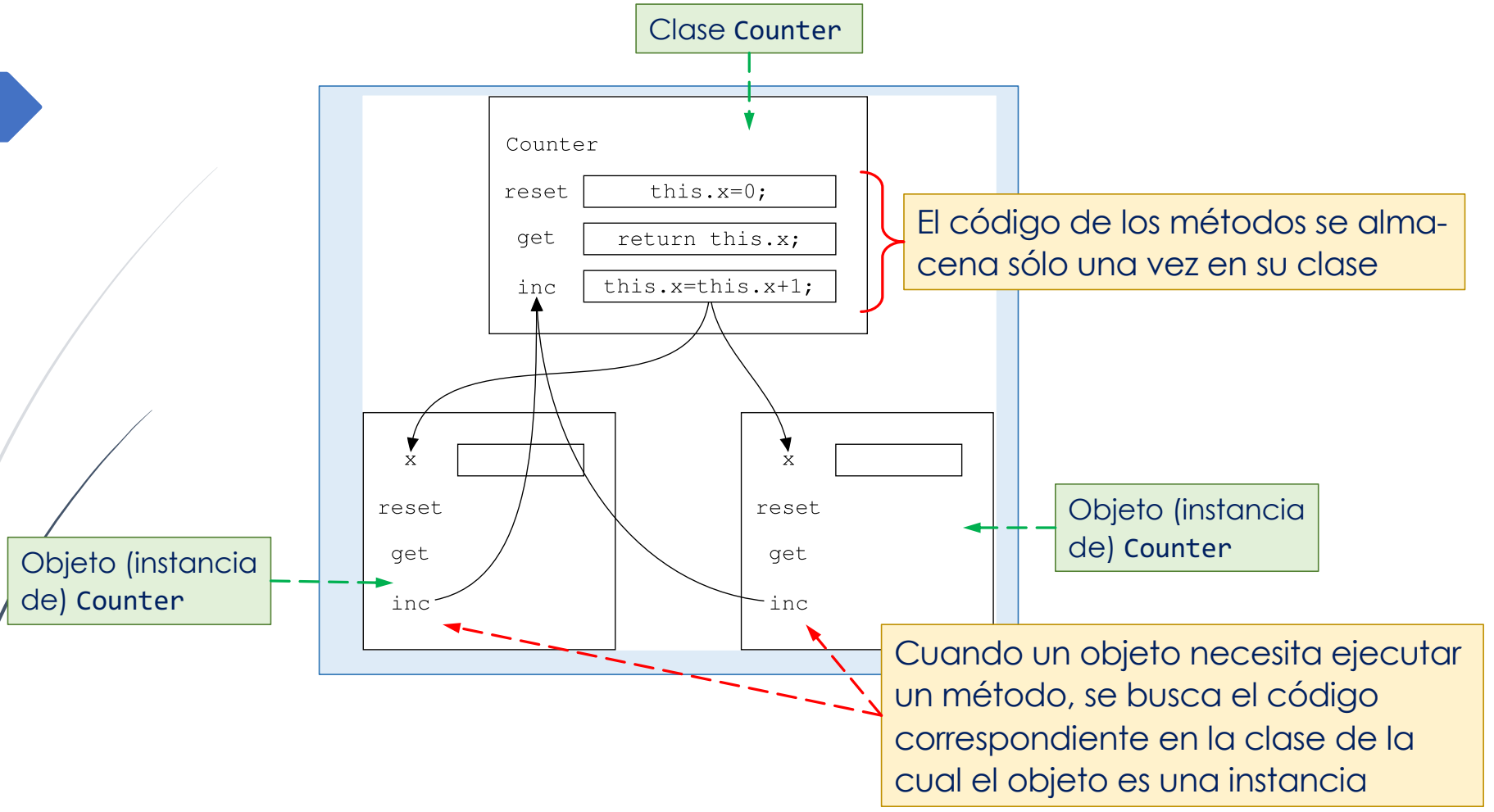
Tipo y *visibilidad* de los atributos

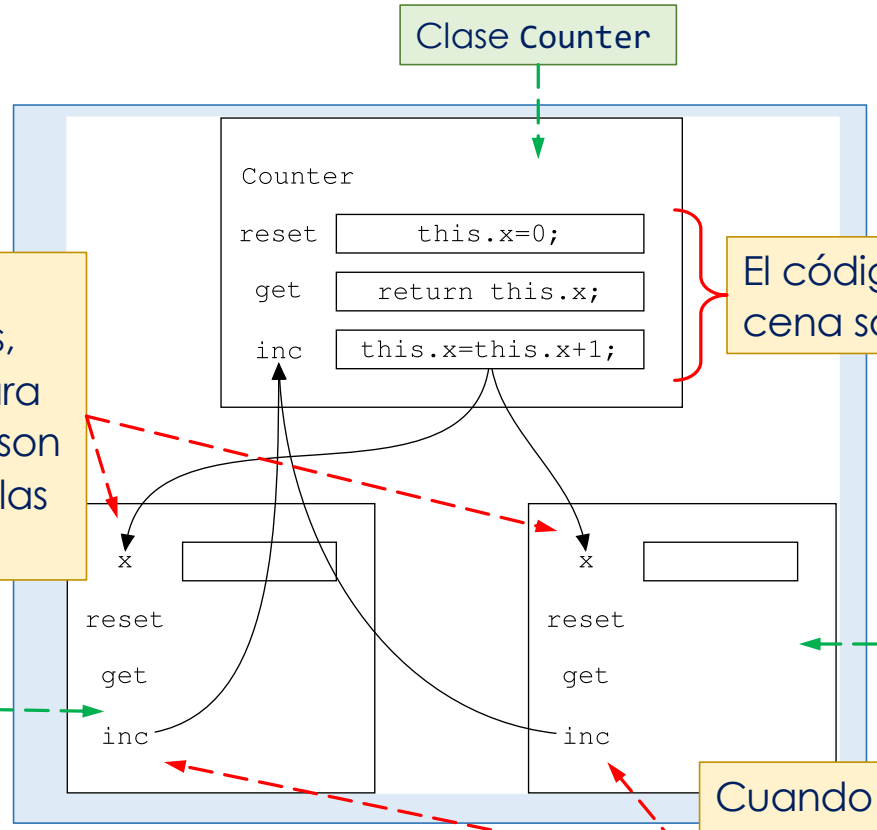
Nombre, firma, *visibilidad* e implementación de cada método

El conjunto de atributos y métodos `public` constituye la **interfaz** de la clase


Clase Counter
Counter
reset
this.x=0;
get
return this.x;
inc
this.x=this.x+1;
x
reset
get
inc
x
reset
get
inc
Objeto (instancia de) Counter
Objeto (instancia de) Counter


Clase Counter
Counter
reset
this.x=0;
get
return this.x;
inc
this.x=this.x+1;
El código de los métodos se alma-
cena sólo una vez en su clase
x
reset
get
inc
x
reset
get
inc
Objeto (instancia
de) Counter
Objeto (instancia
de) Counter


Clase Counter
Counter
reset
this.x=0;
get
return this.x;
inc
this.x=this.x+1;
El código de los métodos se alma-
cena sólo una vez en su clase
x
reset
get
inc
x
reset
get
inc
Objeto (instancia
de) Counter
Objeto (instancia
de) Counter
Cuando un objeto necesita ejecutar
un método, se busca el código
correspondiente en la clase de la
cual el objeto es una instancia



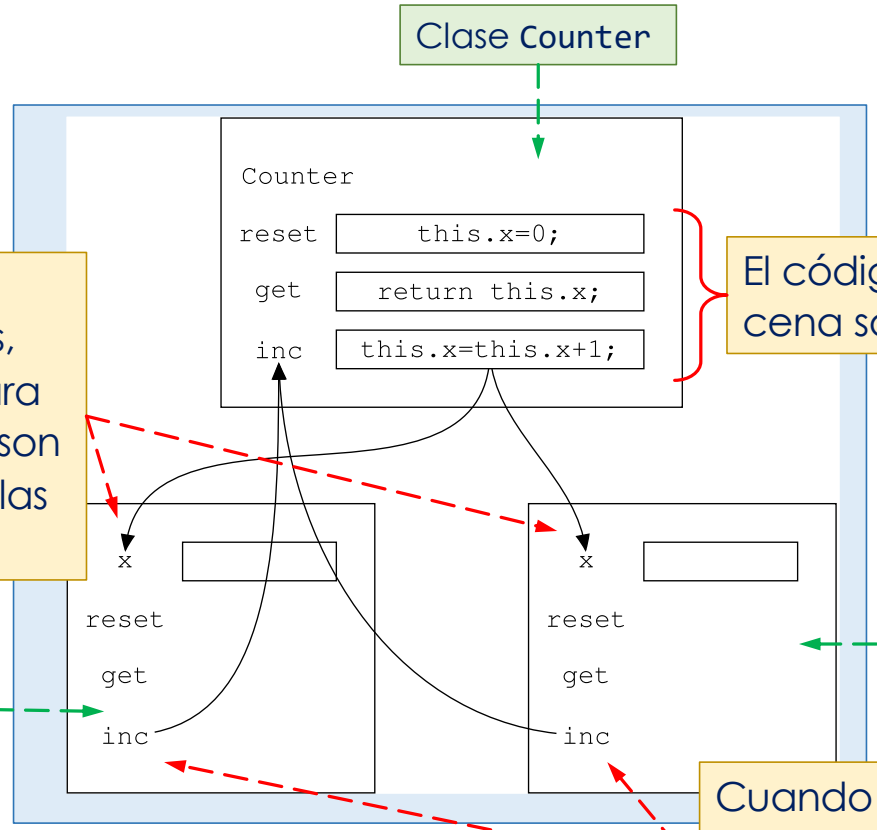
El método debe accesar correctamente los atributos, los cuales son diferentes para cada objeto y por lo tanto son almacenados en éstos (en las instancias) y no en la clase

El código de los métodos se alma-cena sólo una vez en su clase

Objeto (instancia de) `Counter`

Objeto (instancia de) `Counter`

Cuando un objeto necesita ejecutar un método, se busca el código correspondiente en la clase de la cual el objeto es una instancia



El método debe accesar correctamente los atributos, los cuales son diferentes para cada objeto y por lo tanto son almacenados en éstos (en las instancias) y no en la clase

El código de los métodos se alma-cena sólo una vez en su clase

Objeto (instancia de) **Counter**

Objeto (instancia de) **Counter**

Cuando un objeto necesita ejecutar un método, se busca el código correspondiente en la clase de la cual el objeto es una instancia

```
public void inc(){
    x = x+1;
}
```

≡

```
public void inc(){
    this.x = this.x+1;
}
```

Una clase corresponde al conjunto de sus instancias

Una definición de una clase también introduce la definición de un tipo cuyos valores son las instancias de la clase

Entre los tipos obtenidos a través de clases, se define una relación de compatibilidad en términos de las operaciones posibles sobre los valores del tipo:

- El tipo asociado con la clase *T* es un **subtipo** de *S* cuando todo mensaje entendido por objetos de *S* también es entendido por los objetos de *T*

Entre los tipos obtenidos a través de clases, se define una relación de compatibilidad en términos de las operaciones posibles sobre los valores del tipo:

- El tipo asociado con la clase *T* es un **subtipo** de *S* cuando todo mensaje entendido por objetos de *S* también es entendido por los objetos de *T*

Diap. 23

Si representamos un objeto como un record que contiene datos y funciones, entonces

… la relación subtipo corresponde a:

- *T* es un record que contiene todos los campos de *S* (y posiblemente otros campos adicionales)

Entre los tipos obtenidos a través de clases, se define una relación de compatibilidad en términos de las operaciones posibles sobre los valores del tipo:

- El tipo asociado con la clase *T* es un **subtipo** de *S* cuando todo mensaje entendido por objetos de *S* también es entendido por los objetos de *T*

Diap. 23

Si representamos un objeto como un record que contiene datos y funciones, entonces

… la relación subtipo corresponde a:

- *T* es un record que contiene todos los campos de *S* (y posiblemente otros campos adicionales)

Si tomamos en cuenta que algunos campos de los dos tipos podrían ser privados, entonces:

- *T* es un subtipo de *S* cuando la interfaz de *S* es un subconjunto de la interfaz de *T*

En C++, Java y Python, los subtipos deben ser introducidos explícitamente → *subclases* o *clases derivadas*

```
class NamedCounter extending Counter{
    private String name;
    public void set_name(String n){
        name = n;
    }
    public String get_name(){
        return name;
    }
}
```

`NamedCounter` es una subclase (o clase derivada) de `Counter`

`Counter` es una superclase (o clase base) de `NamedCounter`

Las instancias de `NamedCounter` contienen todos los campos de `Counter`, además de campos nuevos introducidos por la definición

Característica fundamental del mecanismo de subclases → capacidad de una subclase de modificar la definición (implementación) de un método de su superclase

Característica fundamental del mecanismo de subclases → capacidad de una subclase de modificar la definición (implementación) de un método de su superclase

Los contadores extendidos de la clase pasada pueden ser definidos como una subclase de **Counter**

```
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int howmany_resets(){
        return num_reset;
    }
}
```

Característica fundamental del mecanismo de subclases → capacidad de una subclase de modificar la definición (implementación) de un método de su superclase

Los contadores extendidos de la clase pasada pueden ser definidos como una subclase de **Counter**

NewCounter extiende la interfaz de **Counter** con nuevos campos

... y redefine el método **reset**

```
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int howmany_resets(){
        return num_reset;
    }
}
```

Un método **reset** enviado a una instancia de **NewCounter** va a producir la invocación de la nueva implementación

Cuando una subclase no redefine los métodos de su superclase

... la subclase hereda los métodos de la superclase

La implementación del método en la superclase se vuelve disponible para la subclase

```
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int howmany_resets(){
        return num_reset;
    }
}
```

`NewCounter` hereda de `Counter` `x`, `inc` y `get` (pero no `reset`, que es redefinido)

En general, herencia es un mecanismo que permite la definición de nuevos objetos basada en el reuso de objetos existentes

La herencia permite el reuso de código:

- al modificar la implementación de un método, una clase automáticamente deja la modificación disponible para todas sus subclases

… sin requerir ninguna acción del programador

La herencia permite el reuso de código:

- al modificar la implementación de un método, una clase automáticamente deja la modificación disponible para todas sus subclases

… sin requerir ninguna acción del programador

Distinguimos *subtipos* de *herencia*

Posibilidad de usar un objeto en otro contexto:

- relación entre las *interfaces* de dos clases
- herencia de interfaz

Posibilidad de reusar el código que maneja a un objeto:

- relación entre las *implementaciones* de dos clases
- herencia de implementación

La herencia permite el reuso de código:

- al modificar la implementación de un método, una clase automáticamente deja la modificación disponible para todas sus subclases

… sin requerir ninguna acción del programador

La herencia permite el **reuso de código**:

- al modificar la implementación de un método, una clase automáticamente deja la modificación disponible para todas sus subclases

… sin requerir ninguna acción del programador

Distinguimos *subtipos* de *herencia*

Posibilidad de usar un objeto en otro contexto:

- relación entre las *interfaces* de dos clases
- *herencia de interfaz*

Posibilidad de reusar el código que maneja a un objeto:

- relación entre las *implementaciones* de dos clases
- *herencia de implementación*

Identificamos dos tipos de campos, según su visibilidad:

- `private`
- `public`

Pero si una subclase —un tipo particular de cliente de la superclase— necesita acceso a un dato no público

… un tercer tipo de campos:

- protegido (`protected`)

En todo caso, ¡ cuidado con el nivel acoplamiento !

Una clase puede heredar de una única superclase inmediata:

- la jerarquía de herencia es un árbol
- el lenguaje tiene *herencia simple*

pero a veces

Una clase puede heredar métodos de más de una superclase inmediata:

- la jerarquía de herencia es un grafo direcccional acíclico (*dag*)
- el lenguaje tiene *herencia múltiple*
- menos común que herencia simple

Una clase puede heredar de una única superclase inmediata:

- la jerarquía de herencia es un árbol
- el lenguaje tiene *herencia simple*

pero a veces

Una clase puede heredar métodos de más de una superclase inmediata:

- la jerarquía de herencia es un grafo direcccional acíclico (*dag*)
- el lenguaje tiene *herencia múltiple*
- menos común que herencia simple

```
class A{
    int x;
    int f(){
        return x;
    }
}
class B{
    int y;
    int f(){
        return y;
    }
}
class C extending A,B{
    int h(){
        return f();
    }
}
```

¿Cuál es el método `f` heredado por `C`?

Problemas de **colisión de nombres** → no tienen una solución elegante, ni conceptual ni de implementación:

- cuando una clase *C* hereda simultáneamente de *A* y *B*, tal que ambas proporcionan implementaciones para métodos con la misma firma

Un método definido en una clase puede ser redefinido (*overridden*) en subclases:

- cuando un método *m* es invocado sobre un objeto o, puede haber varias versiones posibles de *m* para *o*

La selección de cuál implementación de *m* debe usarse en una invocación

o.m(*parameters*)

… es una función del tipo de objeto que recibe el mensaje —no del tipo de la referencia (o nombre) a ese objeto

```
class Counter{
   private int x;
   public void reset(){
      x = 0;
   }
   public int get(){
      return x;
   }
   public void inc(){
      x = x+1;
   }
}
class NewCounter extending Counter{
   private int num_reset = 0;
   public void reset(){
      x = 0;
      num_reset = num_reset + 1;
   }
   public int howmany_resets(){
      return num_reset;
   }
}
```

... y ejecutamos ...

```
Counter c = new NewCounter();
c.reset();
```

```
class Counter{
   private int x;
   public void reset(){
      x = 0;
   }
   public int get(){
      return x;
   }
   public void inc(){
      x = x+1;
   }
}
class NewCounter extends Counter{
   private int num_reset = 0;
   public void reset(){
      x = 0;
      num_reset = num_reset + 1;
   }
   public int howmany_resets(){
      return num_reset;
   }
}
```

… y ejecutamos …

```
Counter c = new NewCounter();
c.reset();
```

El tipo estático del nombre `c` es `Counter`

… pero se refiere dinámicamente a una instancia de `NewCounter`

→ se llama al `reset` de `NewCounter`

Si en `Counter V[100]` se almacenan tanto objetos `Counter` como `NewCounter` y ejecutamos …

```
for (int i=1; i<100; i=i+1)
    V[i].reset();
```

… entonces la búsqueda (*lookup*) dinámica asegura que el método correcto va a ser invocado sobre cualquier contador

En general, el compilador no tiene cómo saber cuál va a ser el tipo del objeto cuyo método va a ser invocado

→ la dinamicidad de este mecanismo

Overloading (diap. 26) y búsqueda dinámica:

- ambos resuelven una ambigüedad

→ un mismo nombre puede tener más de un significado

… pero:

- overloading la resuleve estáticamente
- búsqueda dinámica la resuelve dinámicamente (en tiempo de ejecución)

… es una operación de *overloading en tiempo de ejecución*

Overloading (diap. 26) y búsqueda dinámica:

- ambos resuelven una ambigüedad

⟶ un mismo nombre puede tener más de un significado

… pero:

- overloading la resuleve estáticamente
- búsqueda dinámica la resuelve dinámicamente (en tiempo de ejecución)

… es una operación de *overloading en tiempo de ejecución*

… basado en el tipo de los nombres involucrados

… basada en el tipo dinámico del objeto (y no del nombre del objeto)

El objeto que recibe el mensaje es el primer argumento del método cuyo nombre debe ser resuelto

```
class A{
    int f(){
        return g();
    }
    int g(){
        return 1;
    }
}
class B extending A{
    int g(){
        return 100;
    }
}
```

Sea `b` una instancia de `B`

… e invocamos `b.f()`

`f` heredado de **A**

`f` llama a `g` : ¿cuál de las dos implementaciones de `g` es ejecutada?

```
class A{
    int f(){
        return g();
    }
    int g(){
        return 1;
    }
}
class B extending A{
    int g(){
        return 100;
    }
}
```

Sea `b` una instancia de `B`

… e invocamos `b.f()`

`f` heredado de **A**

`f` llama a `g` : ¿cuál de las dos implementaciones de `g` es ejecutada?

El objeto que recibe el mensaje es un parámetro del método

La llamada a `g` en `f` puede escribirse `this.g()`

this es el objeto vigente

El objeto vigente es `b` ⟶ el método que será invocado es el de `B`

```
class A{
    int f(){
        return g();
    }
    int g(){
        return 1;
    }
}
class B extending A{
    int g(){
        return 100;
    }
}
```

Sea `b` una instancia de `B`

… e invocamos `b.f()`

`f` heredado de **`A`**

`f` llama a `g` : ¿cuál de las dos implementaciones de `g` es ejecutada?

El objeto que recibe el mensaje es un parámetro del método

La llamada a `g` en `f` puede escribirse `this.g()`

`this` es el objeto vigente

El objeto vigente es `b` ⟶ el método que será invocado es el de `B`

Una llamada `this.g()` en `f` puede referirse a implementaciones de métodos que aún no están escritos

… y que lo estarán sólo más tarde a través de la jerarquía de clases ⟶ *asociación tardía* de `this`

`this` es asociado dinámicamente al objeto vigente

¿Cómo se implementa objetos?

Un objeto instancia de la clase *A* puede ser usado como el valor de cualquier superclase de *A*:

- es posible tener acceso no sólo a las variables de instancia definidas explícitamnete en *A*

… sino también a las definidas en sus superclases

¿Cómo se implementa objetos?

Un objeto instancia de la clase *A* puede ser usado como el valor de cualquier superclase de *A*:

- es posible tener acceso no sólo a las variables de instancia definidas explícitamnete en *A*

… sino también a las definidas en sus superclases

En Java y C++ :

… como un record con tantos campos como variables de instancia hay en la clase de la cual es instancia

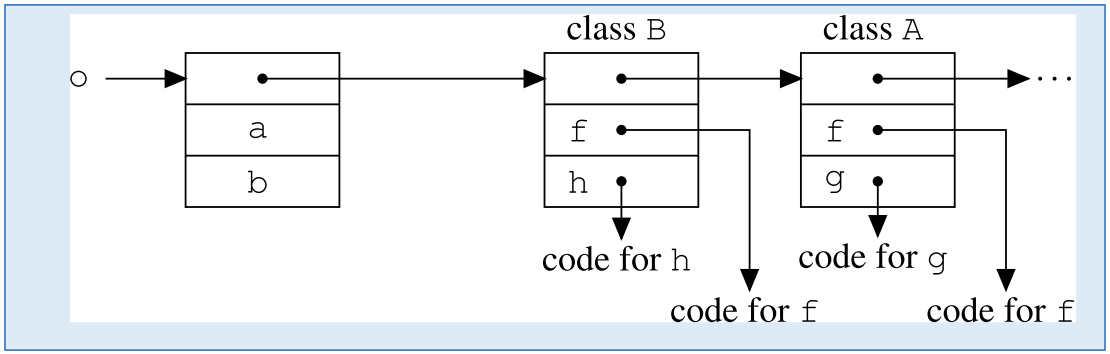
Class instance record (CIR)

… además de todas las que aparecen en sus superclases

+ puntero al descriptor de la clase

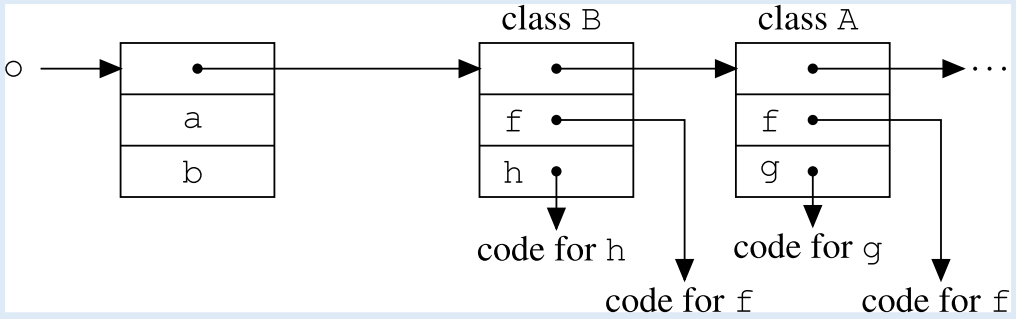
¿Cómo se implementa clases?

Intuitivamente, la jerarquía de clases se puede representar usando una lista ligada:



¿Cómo se implementa clases?

Intuitivamente, la jerarquía de clases se puede representar usando una lista ligada:



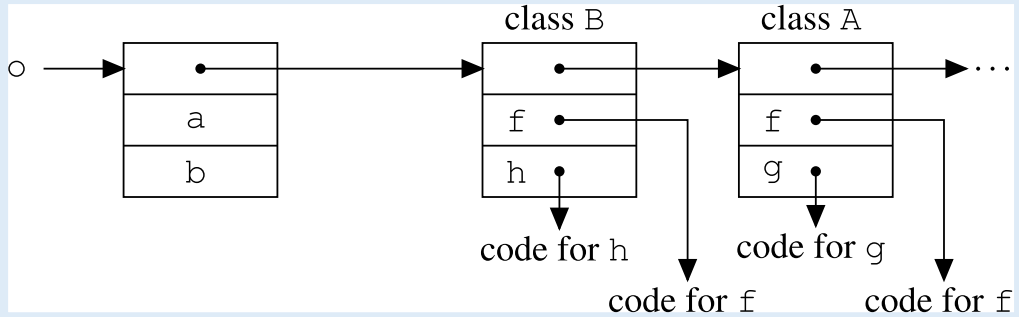
Cuando un método ***m*** de un objeto ***o*** instancia de ***C*** es llamado, el puntero almacenado en ***o*** se usa para tener acceso al descriptor de ***C***

... y determinar si ***C*** contiene o no una implementación de ***m*** :

- en caso negativo se usa la lista de la jerarquía de clases y el procedimiento anterior se repite en la superclase inmediatamente superior de ***C***

¿Cómo se implementa clases?

Intuitivamente, la jerarquía de clases se puede representar usando una lista ligada:



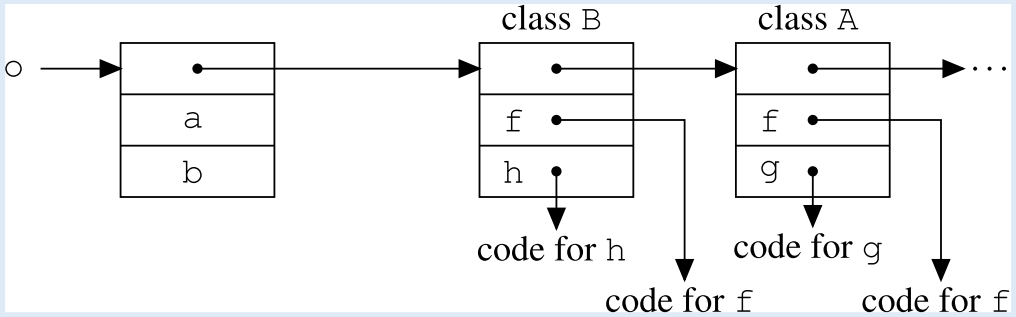
El método se ejecuta similarmente a una fiunción:

- se ingresa al stack el *RA* correspondiente
- y se pasa como parámetro (adicional) un puntero al objeto (**`this`**)

⟶ el método puede tener acceso a las variables de instancia del objeto

¿Cómo se implementa clases?

Intuitivamente, la jerarquía de clases se puede representar usando una lista ligada:



El método se ejecuta similarmente a una fiunción:

- se ingresa al stack el *RA* correspondiente
- y se pasa como parámetro (adicional) un puntero al objeto (**`this`**)

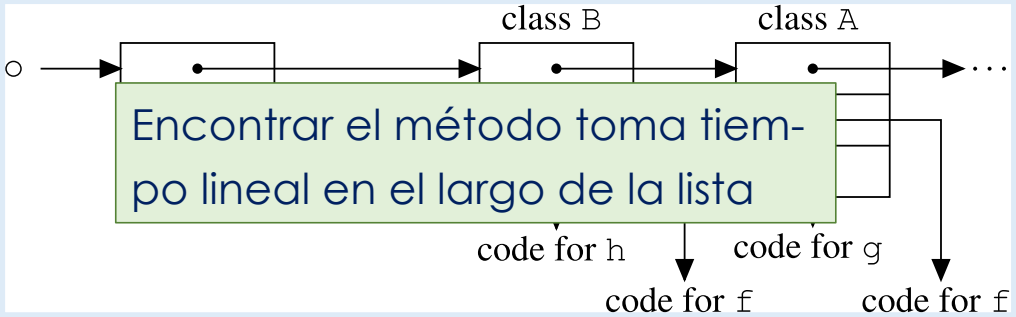
⟶ el método puede tener acceso a las variables de instancia del objeto

... la estructura del objeto es conocida en tiempo de compilación (CIR)

⟶ el *offset* de cada variable de instancia con respecto a este puntero es conocido

¿Cómo se implementa clases?

Intuitivamente, la jerarquía de clases se puede representar usando una lista ligada:



El método se ejecuta similarmente a una fiunción:

- se ingresa al stack el *RA* correspondiente
- y se pasa como parámetro (adicional) un puntero al objeto (**`this`**)

⟶ el método puede tener acceso a las variables de instancia del objeto

… la estructura del objeto es conocida en tiempo de compilación (CIR)

⟶ el *offset* de cada variable de instancia con respecto a este puntero es conocido

El conjunto de métodos que un objeto puede invocar se conoce en tiempo de compilación:

- la lista —*vtable*— se mantiene en el descriptor de la clase

Es la misma para todas las intancias de la clase

Virtual method table (*vtable*):

- Los métodos explícitamente definidos o redefinidos en la clase
- Los métodos heredados de las superclases

El conjunto de métodos que un objeto puede invocar se conoce en tiempo de compilación:

- la lista —*vtable*— se mantiene en el descriptor de la clase

Es la misma para todas las intancias de la clase

Virtual method table (*vtable*):

- Los métodos explícitamente definidos o redefinidos en la clase
- Los métodos heredados de las superclases

Cada clase tiene su propia vtable

… y todas las instancias de la misma clase comparten la misma vtable

Cuando se define una subclase **B** de una clase **A**

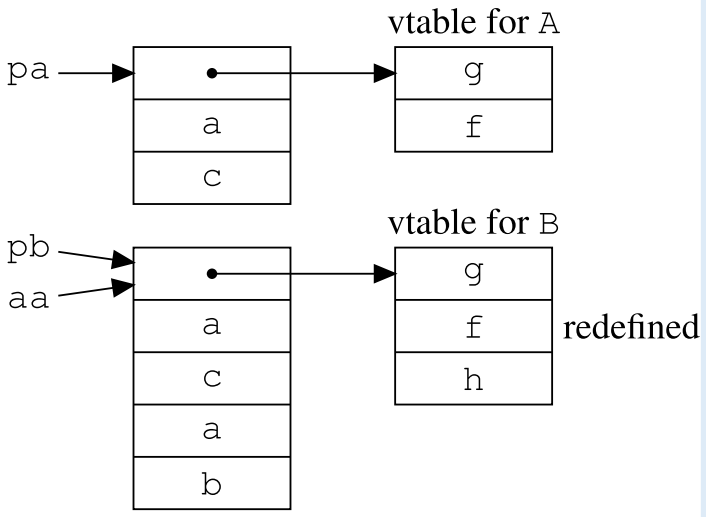
… la vtable de **B** es generada copiando la de **A**

… reemplazando todos los métodos redefinidos en **B**

… y agregando al final los nuevos métodos definidos por **B**

```
class A{
    int a;
    char c;
    void g(){...}
    void f(){...}
}
class B extending A{
    int a;
    int b;
    void h(){...}
    void f(){...}
        // ridefinito

}
B pb = new B();
A pa = new A();
A aa = pb;
aa.f();
```



La invocación de un método toma tres accesos indirectos:

- el offset de cada método en la vtable se conoce estáticamente

Es posible tener acceso a un objeto con una referencia que pertenece estáticamente a una superclase:

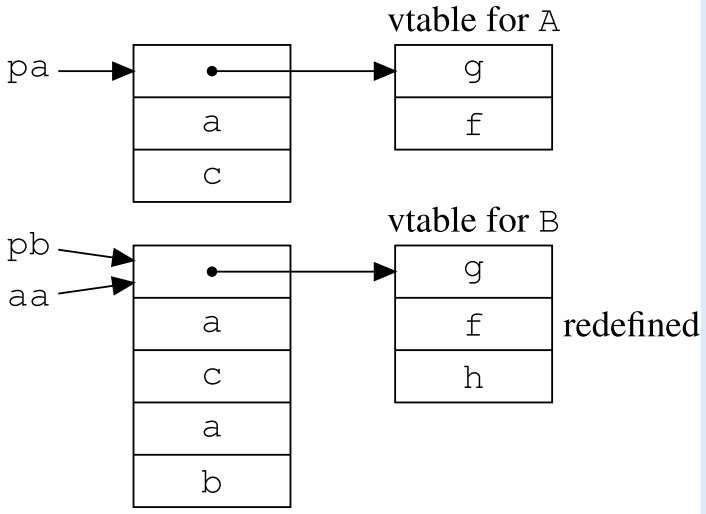
- cuando se invoca a **f**, el compilador calcula un offset que permanece igual

… ya sea que **f** es invocado sobre un objeto de la clase **A**

… o sobre un objeto de la clase **B**

```
class A{
    int a;
    char c;
    void g(){...}
    void f(){...}
}
class B extending A{
    int a;
    int b;
    void h(){...}
    void f(){...}
        // ridefinito

}
B pb = new B();
A pa = new A();
A aa = pb;
aa.f();
```



La invocación de un método toma tres accesos indirectos:

- el offset de cada método en la vtable se conoce estáticamente

Es posible tener acceso a un objeto con una referencia que pertenece estáticamente a una superclase:

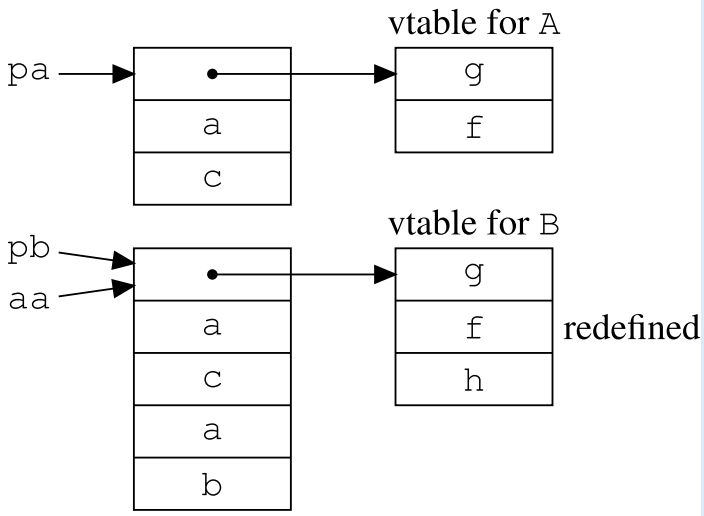
- cuando se invoca a **f**, el compilador calcula un offset que permanece igual

… ya sea que **f** es invocado sobre un objeto de la clase **A**

… o sobre un objeto de la clase **B**

```
class A{
   int a;
   char c;
   void g(){...}
   void f(){...}
}
class B extending A{
   int a;
   int b;
   void h(){...}
   void f(){...}
      // ridefinito

}
B pb = new B();
A pa = new A();
A aa = pb;
aa.f();
```



Compilación de la llamada a **f**:

R1 ⟵ **aa**
R2 ⟵ *(R1)
R3 ⟵ *(R2 + (*n*–1) × *w*)
call *(R3)

f ocupa la *n*-ésima posición en la vtable de **A**

La dirección de un método ocupa *w* bytes

Si tenemos esta definición:

```
class Upper{
    int x;
    int f(){..}
}
class Lower extending Upper{
    int y;
    int g(){return y + f();}
}
```

… y luego hacemos este cambio:

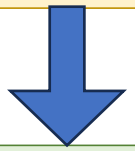
```
class Upper{
    int x;
    int h(){return x;}
    int f(){..}
}
```

… **Lower** deja de funcionar

Una posibilidad es recompilar todas las subclases de la clase modificada

Otra, calcular dinámicamente el offset de los métodos en la vtable

… y de las variables de instancia en el CIR



Dynamic Method Dispatch in JVM