Tecnología de Programación

Martín L. Larrea

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

UML - Diagramas de clases

El diagrama de clases es un diagrama de la estructura estática del sistema.

Un diagrama de clases describe los tipos de objetos en el sistema y las dependencias estáticas que existen entre ellos.

Se describe la estructura de la clase (*nombre, atributos, operaciones, responsabilidades*) y se muestran las relaciones existentes entre las clases.

Las relaciones más comunes son

la asociación,

la agregación,

la composición y

la generalización.

Relaciones entre clases

Existe una relación especial entre clases que toma varias formas:

Una clase puede ser una extensión de otra clase

Una clase PilaRápida con las mismas operaciones de Pila pero ademas con una operación `desapilarDos()` *que desapila dos elementos.*

Una clase puede ser una especialización de otra clase

Matemáticamente un cuadrado es un caso especial de polígonos.
La clase Cuadrado es una especialización de la clase Poligono.

Una clase puede ser una combinación de otras clases

Un ayudante B es un alumno de la Universidad y un docente de la Universidad.
Es una combinación de la clases Alumno y Docente

Reutilización y extendibilidad

Es interesante que las herramientas que disponemos para construir software permitan cumplir a pleno los objetivos de reutilización y extendibilidad.

Un buen acercamiento a la reutilización fué la genericidad, pero se limita únicamente a la parametrización de tipos.

Un mecanismo para lograr un buen grado de reutilización y flexibilidad está relacionada con la noción de generalización.

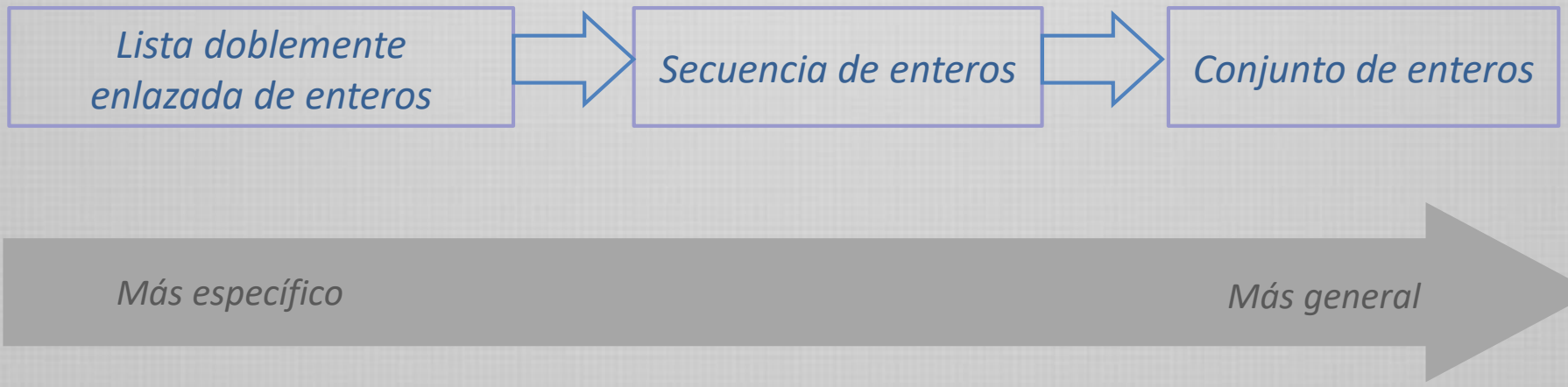
De hecho, la genericidad es una forma de generalización.

Generalización

¿Qué es *generalizar*?

Generalizar es abstraer lo que es común y esencial a muchas cosas, para formar un concepto general que las comprenda a todas.

Una forma de generalización…

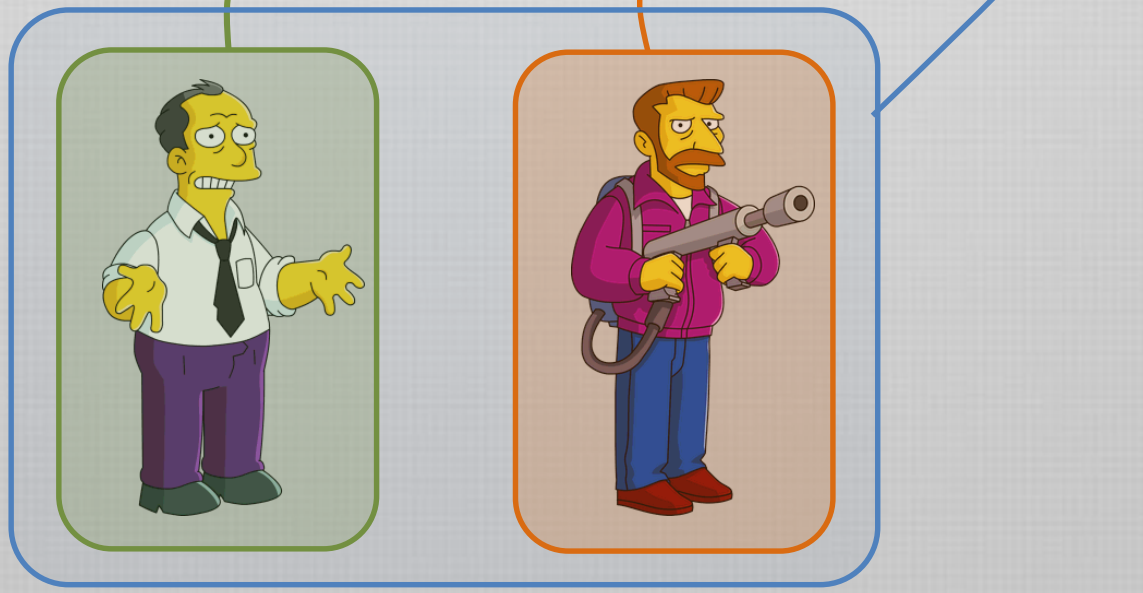


Las nociones más generales describen también las nociones más específicas.
Al menos sus características comunes.

Generalización

Un típico ejemplo de generalización en la literatura: *clientes de un banco*

Supongamos que existen dos clases de clientes:
clientes personales y clientes corporativos.



Generalización

Nombre
DNI
CBU
Saldo

Nombre
DNI
Cuentas
Tarjetas

Los clientes tienen muchas diferencias, pero también muchas similitudes.
Luego, la noción misma de *cliente* es una
generalización de los tipos de clientes existentes.

Generalización

Nombre
DNI
CBU
Saldo

Nombre
DNI
Cuentas
Tarjetas

Podemos decir, por ejemplo, que un *cliente corporativo* es un subtipo de *cliente*, dado que al fin y al cabo, todos los clientes corporativos son clientes.

Luego las características de un cliente corporativo incluyen las características generales de un cliente.

Generalización

Nombre
DNI
CBU
Saldo

Nombre
DNI
Cuentas
Tarjetas

Al mencionar los clientes, podemos
substituir un cliente por cualquier tipo **especial** de cliente.

"A los clientes se les subirá el arancel un 5%"

Cuando hablamos de *clientes* en general, hablamos de todos,
incluso de *clientes corporativos*.

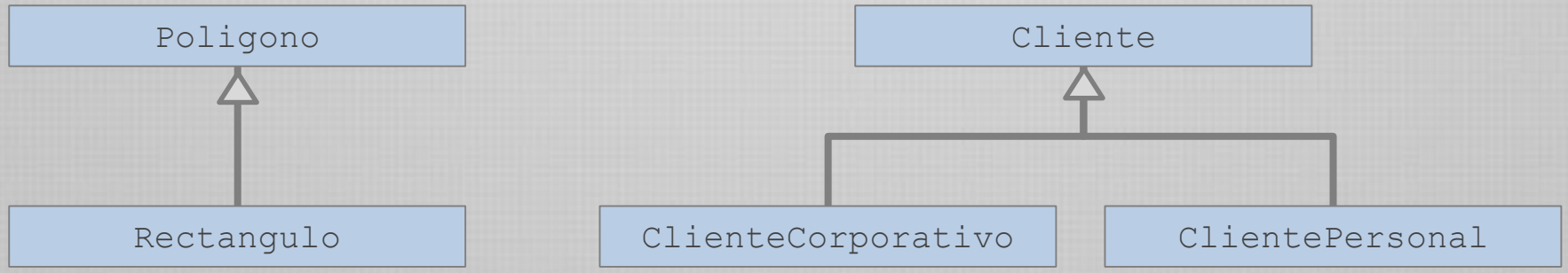
Generalización en UML

Naturalmente identificaremos relaciones de *generalización* en el mundo real.
(generalizar aquí es bueno, pues estructuramos mejor nuestro conocimiento)

Algunas generalizaciones serán naturales, otras surgirán por conveniencia práctica (organización de la información, flexibilidad, etc)

UML nos provee una forma de denotar relaciones de generalización entre clases.
Modela la relación "*es un*".

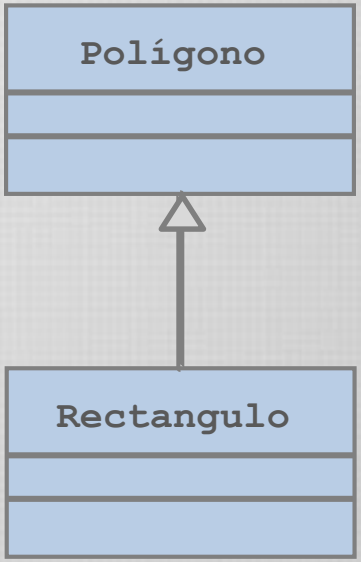
Se utiliza una flecha triangular. 



Generalización, más conocida como herencia

La herencia implica que la clase que hereda dispone de todos los elementos de la otra clase (atributos, operaciones, invariantes, etc). Puede usarlos con total libertad, como si hubiesen sido declarados directamente en la clase hija.

De ahí la idea de “herencia”.

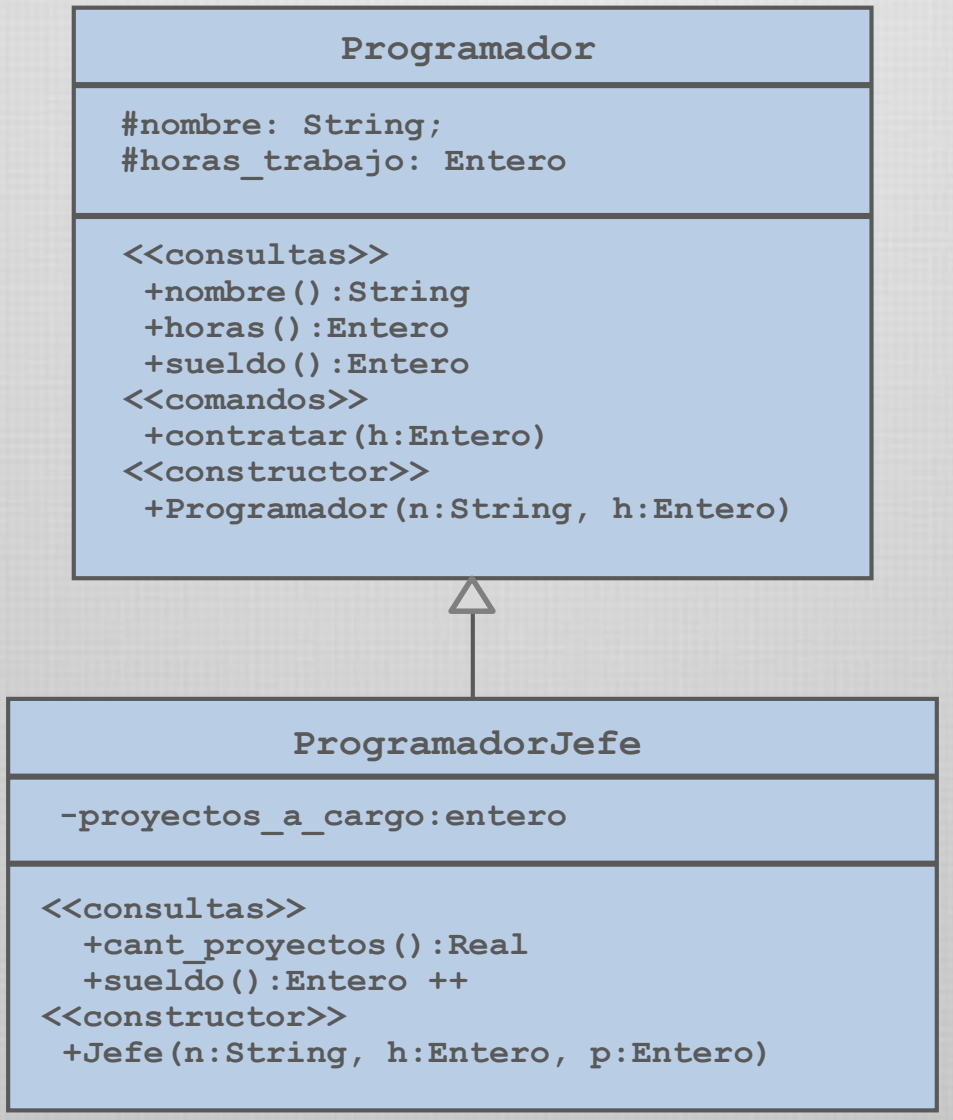


Además, la clase que hereda puede agregar elementos propios que se suman a los heredados.

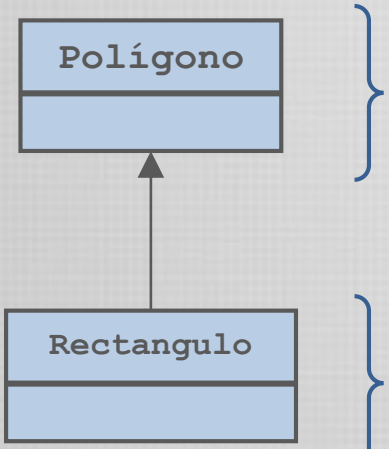
Es posible incluso en la clase que hereda *cambiar* la implementación de una operación por otra más conveniente.

Esto se denomina redefinición de operaciones

Ejemplo


Programador
#nombre: String;
#horas_trabajo: Entero
<<consultas>>
+nombre():String
+horas():Entero
+sueldo():Entero
<<comandos>>
+contratar(h:Entero)
<<constructor>>
+Programador(n:String, h:Entero)
ProgramadorJefe
-proyectos_a_cargo:entero
<<consultas>>
+cant_proyectos():Real
+sueldo():Entero ++
<<constructor>>
+Jefe(n:String, h:Entero, p:Entero)

Terminología



clase padre o superclase o clase base

clase hija o subclase o clase derivada

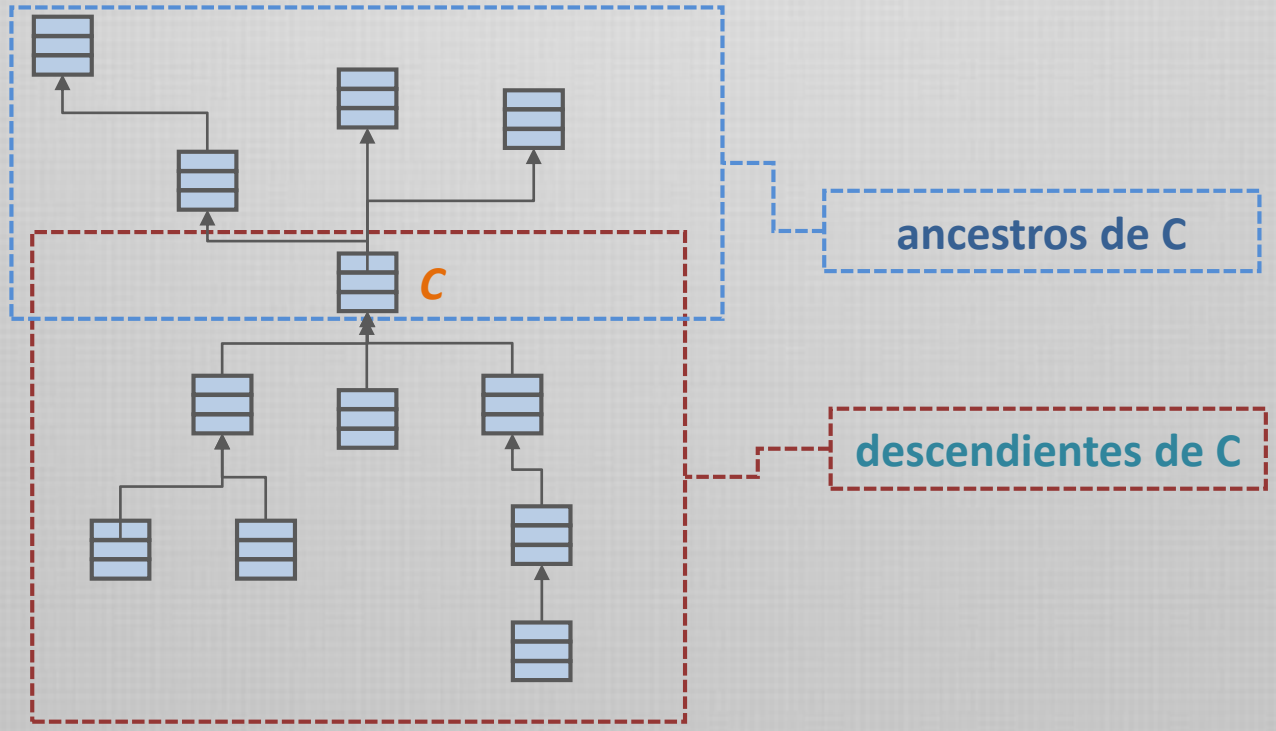
La clase hijo es descendiente directo de la clase padre.
La clase padre es antecesor directo de la clase hijo

Obviamente, la clase **`Rectangulo`** puede tener también *herederos*.
Y estos también, y así sucesivamente, estableciendo toda una
jerarquía de herencia

Terminología

Las clases descendientes de una clase *C* son las clases que heredan directa o indirectamente de *C*, inclusive *C*.
Un descendiente propio de C es un descendiente distinto de C.

Una clases ancestro de una clase *C* es una clase *A* tal que *C* es descendiente de *A*.
Un ancestro propio es una clase *A* tal que *C* es descendiente propio de *A*.



Instancias de una clase

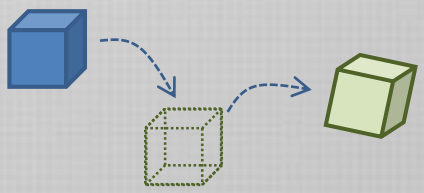
Dado que la generalización habla de aspectos comunes a muchos objetos, es necesario un refinamiento del concepto de *instancias de clases...*

Las instancias de una clase son los objetos que son instancia de algún descendiente de la clase.

Las instancias propias son las instancias de la misma clase

Vale decir,
un **`Rectangulo p`** puede verse como una instancia de la clase **`Polígono`**
un **`ClientePersonal c`** puede verse como una instancia de la clase **`Cliente`**

Todas las instancias de una clase pueden reemplazar (substituir) a una instancia propia, dado que al menos cumplen la misma funcionalidad.



Consequencia importante:
Un objeto puede ser de varios tipos de datos

Herencia y visibilidades

Según la política de visibilidades, en la clase *hija* se heredan atributos y operaciones de la clase *padre*.
Son parte de la clase *hija* y se pueden utilizar como si hubiesen sido declarados en esa clase.

La política de herencia y visibilidad depende en realidad de cada lenguaje.

En general, todas las clases tienen miembros (atributos, operaciones) públicos y privados:

Públicos
pueden ser usados desde cualquier otra clase

Privados
sólo pueden ser usados en la clase en la cual son declarados

Sin embargo, cada lenguaje da interpretaciones diferentes.

Esto naturalmente complica la tarea de diseñar en forma independiente del lenguaje elegido.

Herencia y visibilidades - diversidad

Si bien UML da la libertad de usar cualquier modificador de visibilidad, provee tres tags elementales:

+ *(público)* **-** *(privado)* **#** *(protegido)*

Son modificadores comunes a muchos lenguajes, pero con cierta variación...

En C++:

Un atributo u operación *público* puede ser usado por cualquier objeto del sistema.

Un atributo u operación *privado* puede ser usado en forma directa sólo en la clase que lo define.

Un atributo u operación *protegido* puede ser usado sólo en la clase que lo define y en las clases que de ella heredan.

Herencia y visibilidades - diversidad

En Java:

Un atributo u operación *público* puede ser usado por cualquier objeto del sistema.

Un atributo u operación *privado* puede ser usado
en forma directa sólo en la clase que lo define.

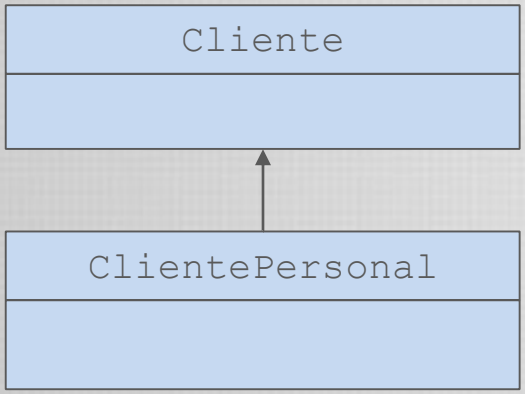
Un atributo u operación *protegido* puede ser usado
sólo en la clase que lo define y en las clases que de ella heredan.

En Smalltalk:

Todos los atributos son siempre *privados* y las operaciones son públicas.

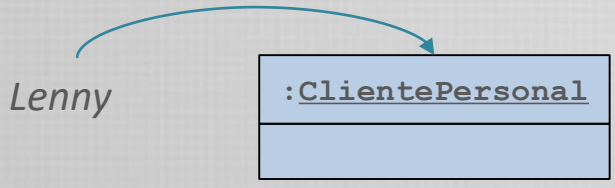
Aquello que es privado igual se hereda y se accede desde los herederos
En ese sentido es parecido al *protegido* de C++/Java

Herencia y visibilidades - diversidad



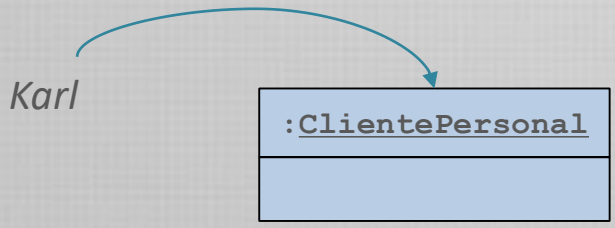
*En **C++***

*En **Java** es similar, pero protegido también exporta al package.*



En la implementación de :ClientePersonal se puede accede a

- cualquier miembro público de cualquier clase
- cualquier miembro privado declarado en *ClientePersonal*
- ningún miembro privado declarado en *Cliente*
- cualquier miembro protegido declarado en *Cliente* o *ClientePersonal*



En la implementación de :ClientePersonal se puede acceder a

- cualquier miembro público, privado o protegido del objeto Lenny definido en *ClientePersonal*.

NOSOTROS ADOPTAREMOS ESTA INTERPRETACION.
Simplemente para unificar posturas durante el cursado.

Redefiniciones

A veces es necesario cambiar la implementación de
ciertas operaciones heredadas.

Para ello se vuelve a declarar la operación en la clase hija,
y se provee una nueva implementación.
El *signature* del servicio debe ser el mismo

Puede reutilizarse el código de la vieja versión de la función
por medio de una palabra reservada.
En nuestro caso usaremos `super` como llamada calificada a dicha versión:

```
super.operacionX(...)
```

Recordemos que esto puede variar entre los lenguajes orientados a objetos
(otros nombres comunes: `base`, `parent`,etc)

Ejemplo

Por ejemplo, para la clase *ProgramadorJefe* la redefinición de la operación **`sueldo()`** puede ser la siguiente:

```
sueldo():entero ++
{
 Resultado ← super.sueldo()+ 500*proyectos_a_cargo
}
```

*"El **sueldo** de un programador Jefe es*

el sueldo que se calcula para los programadores

mas

un monto por la cantidad de proyectos a cargo."

Polimorfismo

Las **instancias** de una clase son los objetos que son instancia de algún descendiente de la clase.

Las instancias propias o directas son las instancias de la misma clase

De acuerdo al postulado anterior los objetos pueden ser tratados de acuerdo a las clases de las cuales son instancia...

Un objeto de tipo *Triangulo* es instancia de la clase *Triangulo*, y también es instancia de la clase *Poligono*.

Un objeto de tipo *ClienteCorporativo* es instancia también de la clase *Cliente*.

¡Es razonable entonces que una referencia declarada como de tipo Polígono pueda ser asociada a Triangulos!

Asociaciones polimórficas

Una asociación polimórfica ocurre cuando a una referencia de una clase se le asocia una referencia a una instancia no directa.

```
p: Poligono ; rec:Rectangulo ; tri:Triangulo
```

Es válido realizar las siguientes asignaciones..

```
p ← rec;
p ← tri;
```

Este tipo de asignaciones se denomina asignación polimórfica.

Una entidad como `p`, que aparece en una asignación polimórfica de este tipo es denominada entidad polimórfica.

El tipo de la expresión que es asociada (lado derecho de la asignación) es descendiente del tipo al que se asocia (lado izquierdo de una asignación)

Estructuras de contenido polimórfico

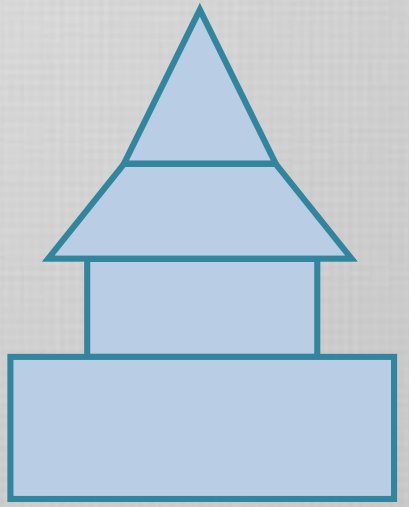
Las estructuras de datos polimórficas surgen de
la combinación de genericidad con herencia en el parámetro genérico formal

A : Arreglo[Poligono]

```
A[1] ← new Rectangulo(...)
A[2] ← new Triangulo(...)
A[3] ← new Cuadrado(...)
A[4] ← new Trapecio(...)

i ← 1
areatotal ← 0
repetir mientras i<=4
{
  areatotal ← areatotal + A[i].getArea()
  i ←i+1

}
```



Chequeo estático de tipos

El control de la validez de las asignaciones se realiza como parte del chequeo de tipos

Este proceso verifica el programa de acuerdo a un conjunto de reglas definidas en el sistema de tipos.

Este sistema indica
Cómo los valores son estructurados en tipos de datos.
Cómo manipular esos tipos de datos.

Los lenguajes basados en clases tienen generalmente un chequeo *estático* de tipos, que se realiza en tiempo de compilación.

Los objetos requieren declaración previa del tipo de dato al que pertenecen.

```
Empleado lenny;
int i;
```

Chequeo estático de tipos

¿Cuáles son las ventajas de tener que declarar los tipos de los objetos?

Nuestro software será más confiable.
Los compiladores detectan los errores y discrepancias antes de que puedan causar mayor daño.

Nuestro software es más fácil de leer.
La estructura de los objetos, su forma de manipularlos, y el rol que cumplen en el sistema es mucho más claro.

Nuestro software es más eficiente.
Facilita optimizaciones posteriores de código, al tener un mayor conocimiento de los tipos de datos y cómo son utilizados en ese programa en particular.

Chequeo estático de tipos

Un lenguaje orientado a objetos es *estáticamente tipado* si está equipado con un conjunto de reglas de consistencia, controlada por los compiladores, cuya observancia por el texto del software garantiza que la ejecución del software no incurrirá en una violación de tipos. [Meyer]

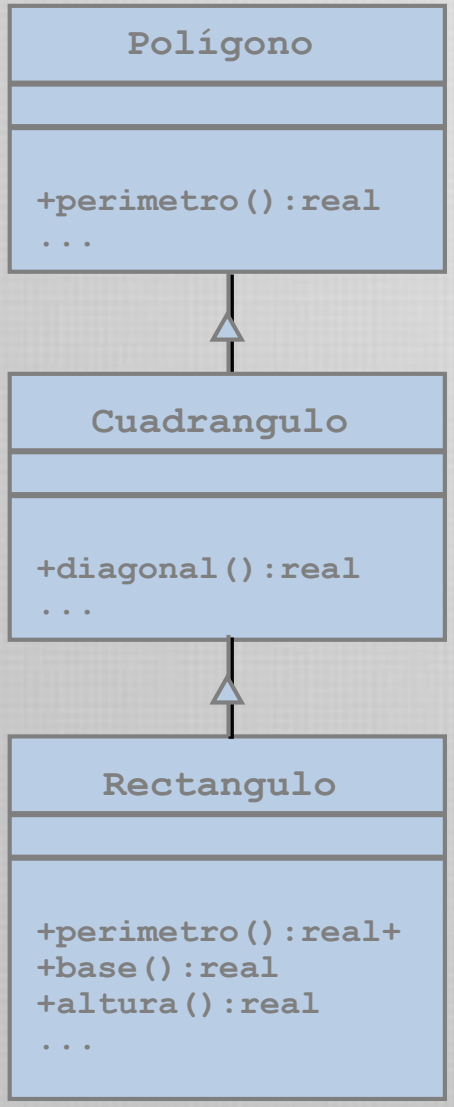
Ejemplo de una regla de control de tipos

Toda entidad debe tener un tipo declarado antes de ser utilizada

En una llamada `x.f()`, donde el *tipo base* de `x` es `C`, la operación `f()` debe estar definida en un ancestro de `C` y ser visible para la clase donde el mensaje ocurre.

*Esta regla establece la correctitud de una llamada calificada desde el punto de vista del tipo de datos del objeto **x**.*

Invocaciones y control de tipos




```
p:Poligono; r:Rectangulo; c:Cuadrangulo
```

Las siguientes operaciones son válidas:

```
p.perimetro()
r.base()
r.diagonal()
r.perimetro()
```

Las siguientes operaciones son inválidas:

```
p.base()
p.diagonal()
c.altura()
```

Conformidad de tipos

Otra regla de control de tipos

Una asociación

x ← y

de origen x y destino y es sólo válida si el tipo de y conforma el tipo de x.

Conformidad de tipos

Un tipo U *conforma* un tipo T sólo si el tipo base de U es un descendiente de la clase base de T;

Para tipos derivados genéricamente, cada parámetro actual de U debe conformar el parámetro formal de T

Ejemplos:
Rectangulo conforma a **Poligono**,
Cuadrado conforma a **Rectangulo**,
ClientePersonal conforma a **Cliente**.

Control de tipos y asignaciones

Conformidad de tipos

Un tipo U *conforma* un tipo T sólo si
el tipo base de U es un descendiente de la clase base de T;

Para tipos derivados genéricamente,
cada parámetro actual de U debe conformar el parámetro formal de T

Para clases genéricas:

B[Y] conforma a A[X], si
B es descendiente de A e
Y es descendiente de X.

Ejemplos:

ListaCircular[Rectángulo] conforma a Lista[Poligono]
Programador[Juego] conforma a Empleado[Producto]

Tipos dinámicos y estáticos de una referencia

Dado que una referencia puede estar asociada a objetos de diferente tipo, podemos hacer dos distinciones del "*tipo de una referencia*".

- El tipo estático de una entidad x es el tipo usado para declarar esa entidad.

```
p :Poligono; r: Rectangulo; c:Cuadrado a: Alumno
```

- El tipo dinámico de una entidad x en un determinado momento de ejecución es la clase de la que es instancia directa el objeto asociado a x en ese momento.

```
p ← r;
p ← c;
```