Inferencia de Tipos

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Introducción

Problema

Trabajando en un banco, hay varios tipos de cuentas:

- **Cuentas corrientes :** Tienen un tipo de interés anual (siendo el mensual el resultado de dividirlo por doce).
- **Plazos fijos :** Vienen determinados por un plazo (número de meses) y un tipo de interés a vencimiento. Antes del plazo, da interés 0 %.
- **Cuenta de nómina :** Se ingresa mensualmente una cantidad fija determinada.
- **Cuenta en divisa :** Dada otra cuenta en otra divisa y dado un factor de cambio entre divisas, calcule la cuenta con el factor de cambio aplicado.

Las cuentas están definidas mediante una jerarquía de clases, y todas disponen de los siguientes métodos:

- `float value(int t)`: valor en *t* meses
- `void actualiza()`: avanza un mes el estado de una cuenta
- `float tae()`: calcula la tasa anual equivalente.

Problema

Sobre la jerarquía de clases, generar un **interfaz mediante línea de comandos** que, dado un vector de cuentas, permita:

- Acceder a los metodos de cada cuenta (valor dentro de varios meses o su TAE) mediante su identificador (indice del vector), el nombre del metodo y los parametros que sean necesarios.
- Obtener o modificar las propiedades especificas de cada cuenta (capital, tipo de interes, plazo, cantidad mensual...), que dependen de cada tipo de cuenta.
- Volcar un vector de cuentas a fichero y luego recuperarlo.

Problema

Sobre la jerarquía de clases, generar un **interfaz mediante línea de comandos** que, dado un vector de cuentas, permita:

- Acceder a los metodos de cada cuenta (valor dentro de varios meses o su TAE) mediante su identificador (indice del vector), el nombre del metodo y los parametros que sean necesarios.
- Obtener o modificar las propiedades especificas de cada cuenta (capital, tipo de interes, plazo, cantidad mensual...), que dependen de cada tipo de cuenta.
- Volcar un vector de cuentas a fichero y luego recuperarlo.

Problema

Como requisito adicional, se pide **no modificar las clases que representan las cuentas**.
¿Cómo cambia el problema? ¿Cómo resolverlo?

Supongamos que queremos representar la topología de una red.

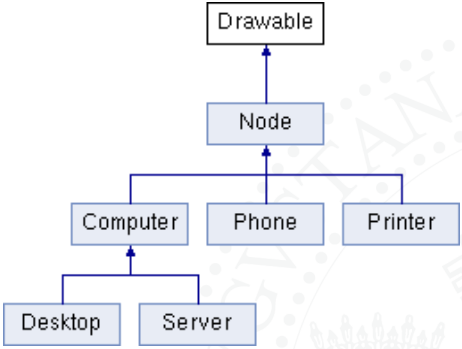
En nuestra red podemos tener diversos elementos:

- Computadores, tanto servidores como de escritorio
- Impresoras
- Teléfonos
- ...

Modelo de clases (Java):

```java
interface Drawable {
  void draw();
}

abstract class Node
  implements Drawable {
    bool ping() { ... };
}

abstract class Computer
  extends Node {
    void shutdown() { ... };
}
```

```java
class Server extends Computer { }

class Desktop extends Computer { }

class Printer extends Node { }

class Phone extends Node { }
```


Drawable
Node
Computer
Phone
Printer
Desktop
Server

Para la mayoría de las operaciones, el poliformismo es la herramienta ideal:

```
1   List<Node>  nodes = ...;
2
3
4   for (Node n : nodes)
5     n.draw();
6
7
8   for (Node n : nodes)
9     if (n.ping()) {
10        // It's alive....
11    }
```

```
1  List<Node> nodes = new ArrayList<Node>();
2
3  nodes.add(new Server("phobos"));
4  nodes.add(new Server("deimos"));
5  nodes.add(new Desktop("ws1"));
6  nodes.add(new Desktop("ws2"));
7  nodes.add(new Printer("gutenberg"));
8  nodes.add(new Phone("2354"));
9  nodes.add(new Phone("2355"));
10
11 for (Node n : nodes)
12   n.draw();
```

El polimorfismo funciona bien siempre que los objetos (clases) tengan comportamientos específicos que especializan uno genérico:
unos se comportan de forma distinta a otros, pero todos presentan esa funcionalidad.

¿ Pero qué ocurre si por ejemplo queremos . . .

- . . . resaltar en el gráfico los teléfonos ?
- . . . apagar todos los computadores ?
- . . . añadir una etiqueta con el tipo de objeto ?

- . . . resaltar en el gráfico los teléfonos ?

```
1 for (Node n : nodes) {
2   if (  n es un telefono  )
3     color(verde);
4   else
5     color(negro);
6   n.draw();
7 }
```

- . . . apagar todos los computadores ?

```
1 for (Node n : nodes)
2 {
3   if (  n es un computador  )
4     // Computer.shutdown()
5     // n es Node
6     n.shutdown();
7 }
```

Ojo: sólo los objetos `Computer` se pueden apagar . . .

- . . . añadir una etiqueta con el tipo de objeto ?

```
1 for (Node n : nodes) {
2   // Descripción textual del tipo de nodo ???
3   // "Server", "Phone", ...
4   String etiqueta = ....
5   // dibujar etiqueta
6   n.draw();
7 }
```

Conversión (*casting*) es el término empleado para definir conversiones entre diferentes tipos de datos del lenguaje.

Inferencia (*inference*) es el mecanismo mediante el cual se puede conocer de qué tipo de datos es un elemento del lenguaje.

Introspección (*introspection*) es el término empleado para describir el proceso en que un objeto puede averiguar información sobre sí mismo u otros.

Reflectividad (*reflection*) hace referencia al proceso por el que un objeto es capaz de modificarse a sí mismo o a otros.

RTTI son las siglas de *Run-Time Type Information*

Hace referencia a la capacidad de un lenguaje para obtener información de tipo de las variables en tiempo de ejecución.

Permite obtener diversa información sobre el tipo de un objeto, desde saber si es igual a otro, saber si mantienen una relación de herencia, o incluso obtener una representación textual de su nombre.

Conversiones

Conversiones

En los lenguajes orientados a objetos son frecuentes las **conversiones** entre tipos de datos.

Hay dos tipos de conversiones:

- **Implícitas:** Ocurren automáticamente en el lenguaje bajo ciertas condiciones.
- **Explícitas:** Las determina el programador.

Pregunta

¿ Qué conversiones automáticas de tipos conocemos específicas de los lenguajes orientados a objetos ?

Pregunta

¿ Qué conversiones automáticas de tipos conocemos específicas de los lenguajes orientados a objetos ?

Pista

Es uno de los mecanismos de la herencia.

Pregunta

¿ Qué conversiones automáticas de tipos conocemos específicas de los lenguajes orientados a objetos ?

Pista

Es uno de los mecanismos de la herencia.

Respuesta

Una clase puede representar a cualquiera de sus superclases.

C++

```
class Foo : public Bar { ... };
void op(Bar* bar);
op(new Foo());
```

Java

```
class Foo extends Bar { ... }
class Pet {
   static void op(Bar bar) { ... }
}
Pet.op(new Foo());
```

En general, las conversiones implícitas se aplican cuando una expresión que espera un tipo *T2* es invocada con un dato de otro tipo *T1*, y, bajo algún mecanismo, *T1* puede convertirse en *T2*.
Los mecanismos habituales de conversión son:

- Promoción numérica (entero a real)
- Relacion entre tipos (de clase hija a clase padre).
- Operadores específicos de conversión automática (constructores, operadores de tipos básicos).

- Promoción numérica (entero a real).

```
1  int x = 10;
2  char y = 'a';
3  // y convertido a numero ASCII ('a' es 97)
4  x = x + y;
5  // x se convierte en float
6  float z = x + 1.0;
```

- Operadores específicos de conversión automática: constructores.

```
1  template<typename F> class complex {
2      F re,im;
3      complex(F _re=0, F _im=0) : re(_re), im(_im) {}
4      //...
5  };
6  ...
7  complex<float> c = 1.0f;
8  ...
9  complex<float> my_fun() { return -1.0; }
```

- Operadores específicos de conversión automática: conversiones de usuario.

```
1  class Rational
2  {
3    int num, den;
4
5    operator float() const { return float(num)/float(den); }
6  }
7
8  Rational r(1,3);
9  float    f = r;
```

- Operadores específicos de conversión explícita: conversiones de usuario.

```
1  class Rational
2  {
3    int num, den;
4
5    explicit operator float() const { return float(num)/float(den); }
6  }
7
8  Rational r(1,3);
9  float    f = r;         // ERROR
10 float    f = float(r); // OK
```

C++: *type cast*

```
1 // Datos
2 T1 o1;
3
4 T2 o2 = (T2)o1;
```

```
1 // Punteros
2 T1* o1 = ...;
3
4 T2* o2 = (T2*)o1;
```

¿ Problemas ?

```
1 float  f;
2 double d = (double)f;
3 double d = double(f);
```

```
1 char*  c = ...;
2 float* f = (float*)c;
```

En lenguajes orientados a objetos tienen especial interés las conversiones entre punteros/referencias a objetos.

```cpp
// C++
T1* o1 = new T1();

T2* o2 = (T2*)o1;
```

```java
// Java
T1 o1 = new T1();

T2 o2 = (T2)o1;
```

Se definen nuevas formas de especificar las conversiones de tipo para mejorar el control de lo que ocurre realmente.

C++: reinterpret_cast

```
1 T1* o1 = new T1();
2
3 T2* o2 = reinterpret_cast<T2*>(o1);
```

Realiza una conversión, simplemente copiando la dirección, sin ninguna comprobación.
Muy arriesgado, así que el tipo de conversiones que permite el lenguaje es muy limitado.

C++: static_cast

```
1 T1* o1 = new T1();
2
3 T2* o2 = static_cast<T2*>(o1);
```

Aplica conversiones que se puedan hacer en tiempo de compilación, porque existen conversiones del lenguaje o de usuario.
Si no es posible, error de compilación.
No hace comprobaciones en tiempo de ejecución. Entre otras cosas, puede ir de clase padre a clase hija o viceversa.
Útil cuando se conocen los tipos de datos en tiempo de compilación.

C++: dynamic_cast

```
1 T1* o1 = new T1();
2
3 T2* o2 = dynamic_cast<T2*>(o1);
```

La conversión se realiza en tiempo de ejecución, verificando el tipo concreto de los datos.
Necesita RTTI: tipo polimórfico con algún método virtual.
Si la conversión no es posible (o1 no es internamente de tipo T2) devuelve `nullptr`.

C++: cast

```
1  class B { };
2  class D1 : public B { };
3  class D2 : public B { };
4
5  void fun(B* o) {
6    // Legal, pero inseguro
7    int* i1_ptr = (int*)(o);
8    // Legal, pero inseguro
9    D1*  d1_ptr = (D1*)(o);
10 }
```

C++: reinterpret_cast

```
1  class B { };
2  class D1 : public B { };
3  class D2 : public B { };
4
5  void fun(B* o) {
6    // Legal, pero inseguro
7    int* i1_ptr = reinterpret_cast<int*>(o);
8    // Legal, pero inseguro
9    D1*  d1_ptr = reinterpret_cast<D1*>(o);
10 }
```

C++: static_cast

```
1  class B { };
2  class D1 : public B { };
3  class D2 : public B { };
4
5  void fun(B* o) {
6    // Error
7    int* i1_ptr = static_cast<int*>(o);
8    // Legal, pero inseguro
9    D1*  d1_ptr = static_cast<D1*>(o);
10 }
```

C++: dynamic_cast

```
1  class B { };
2  class D1 : public B { };
3  class D2 : public B { };
4
5  void fun(B* o) {
6    // Error
7    int* i1_ptr = dynamic_cast<int*>(o);
8    // Error, no posible sin RTTI
9    D1*  d1_ptr = dynamic_cast<D1*>(o);
10   if (d1_ptr!=nullptr) // o es _realmente_ un D1
11   { ... }
12 }
```

C++: dynamic_cast

```
1  class B { public: virtual ~B() = default; };
2  class D1 : public B { };
3  class D2 : public B { };
4
5  void fun(B* o) {
6    // Error
7    int* i1_ptr = dynamic_cast<int*>(o);
8    // OK
9    D1*  d1_ptr = dynamic_cast<D1*>(o);
10   if (d1_ptr!=nullptr) // o es _realmente_ un D1
11   { ... }
12 }
```

Java *cast*

```
1 T1 o1 = new T1();
2
3 T2 o2 = (T2)o1;
```

¿ Problemas ?

Java *cast*

```
1  T1 o1 = new T1();
2
3  try {
4     T2 o2 = (T2)o1;
5     ...
6  } catch(ClassCastException e) {
7     ...
8  }
```

Inferencia

Problema

Trabajando en un banco, hay varios tipos de cuentas:

. . .

Diseña una función que devuelva el tipo de interés para una cuenta determinada, que lance una excepción si la cuenta no tiene tipo de interés:

```
double interes(const Cuenta& cuenta)
```

Haz dos versiones: una en la que puedes editar la jerarquía de clases y otra en la que no se te permite.

Inferencia

Inferencia (eng. *inference*) es el mecanismo mediante el cual podemos averiguar el tipo de datos de un elemento del lenguaje.

¿En qué casos no conocemos de qué tipo es algo?

¿En qué casos no conocemos de qué tipo es algo?

En tiempo de ejecución:

```
1 class B { ... };
2 class D1 : public B { ... };
3 class D2 : public B { ... };
4 void fun(B& o) {
5     ...
6     // o puede ser D1, D2 o B
7 }
```

¿En qué casos no conocemos de qué tipo es algo?

En tiempo de compilación:

```
template<typename T>
void fun(T& t) {
    ...
    // t puede ser de cualquier tipo
}
```

¿En qué casos no conocemos de qué tipo es algo?

En tiempo de compilación:

```
1  template<typename T>
2  T fun(...) { ... }
3
4  int main(int argc, char** argv)
5  {
6      // t puede ser de distintos tipos
7      ??? t = fun(...);
8  }
```

- **Inferencia dinámica**: Inferencia de tipos en tiempo de ejecución (relacionada con la herencia).
- **Inferencia estática**: Inferencia de tipos en tiempo de compilación (relacionada con programación genérica).

Inferencia (eng. *inference*) es el mecanismo mediante el cual podemos conocer del tipo de datos de un elemento del lenguaje.
Dinámica (eng. *dynamic*), en programación, significa que ocurre en tiempo de ejecución.

En general, está relacionada con la herencia.

La forma básica de inferencia dinámica son los *casts dinámicos*.

C++

```cpp
class B { ... };
class D1
  : public B { ... };
class D2
  : public B { ... };
```

```cpp
void fun(B* b) {
  ...
  // Si no se puede convertir, nullptr
  D1* d1 = dynamic_cast<D1*>(b);
  if (d1 != nullptr) {
    ...
  }
}
```

La forma básica de inferencia dinámica son los *casts dinámicos*.

Java

```java
class B { ... };
class D1
  extends B { ... };
class D2
  extends B { ... };
```

```java
void fun(B b) {
  ...
  // Si no se puede convertir, excepcion
  try {
    D1 d1 = (D1)b;
    ...
  } catch(ClassCastException e) {
    ...
  }
}
```

Java ofrece el operador **`instanceof`**:

Java

```java
class B { ... };
class D1
  extends B { ... };
class D2
  extends B { ... };
```

```java
void fun(B b) {
  ...
  // Si no se puede convertir, false
  if (b instanceof D1) {
    D1 d1 = (D1)b;
    ...
  }
}
```

Funciona con relaciones de herencia (igual o derivado de...).

Incluye interfaces.

C++ vs Java

```cpp
void fun(B* b) {
  ...
  // Intentamos la conversión
  D1* d1 = dynamic_cast<D1*>(b);
  // ¿ Ha sido posible ?
  if (d1 != nullptr) {
    ...
  }
}
```

```java
void fun(B b) {
  ...
  // ¿ Es posible la conversión ?
  if (b instanceof D1) {
    // Hacemos la conversión
    D1 d1 = (D1)b;
    ...
  }
}
```

Inferencia (eng. *inference*) es el mecanismo mediante el cual podemos conocer del tipo de datos de un elemento del lenguaje.
Estática (eng. *static*), en programación, significa que ocurre en tiempo de compilación.

En general, relacionada con la programación genérica.

Sobrecarga y *templates*:

```
1  template<typename T>
2  void op(T& t) {
3      ...
4      // t puede ser de cualquier tipo
5      //(menos los sobrecargados)
6  }
7  void op(int& t) {
8      ...
9      // Aqui sabemos que t es un entero
10 }
```

Especialización de *templates*

```
1  template<typename T>
2  class V {
3    ...
4    //T puede ser cualquier tipo
5    // (menos los especializados)
6  };
7  template<>
8  class V<float> {
9    ...
10   //El tipo usado es float
11 };
```

La programación genérica en general es un mecanismo habitual de inferencia estática.

Pregunta : ¿Cómo podemos saber si un tipo de dato dispone del operador `'>'`?

Pregunta : ¿Cómo podemos saber si un tipo de dato dispone del operador '>'?

Esto no compila si no está definido el operador '>' para el tipo T.

C++

```
template<typename T>
T max(const T& o1, const T& o2) {
  if (o1 > o2) return o1;
  else return o2;
}
```

Pregunta : ¿Cómo podemos saber si un tipo de dato dispone del operador '>'?

Esto no compila si T no es `Comparable`.

Java

```java
<T extends Comparable<T>>
T max(T o1, T o2) {
  if (o1.compareTo(o2) > 0) return o1;
  else return o2;
}
```

Problema

Diseña en C++ una función que sume todos los elementos de una lista, vector, array o conjunto (un contenedor de C++).

```
??? sum(const ??? & contenedor)
```

Diseña en C++ una función que sume un contenedor:
usando la información del concepto `Container`.

```
1  template<typename C>
2  typename C::value_type sum(const C& c)
3  {
4    typename C::value_type total{};
5
6    for (typename C::const_iterator it=begin(c); it!=end(c); it++)
7      total = total + (*it);
8
9    return total;
10 }
```

Diseña en C++ una función que sume un contenedor:
usando tipos `template` en el template.

```
1  template<typename T,template<typename> typename C>
2  T sum(const C<T>& c)
3  {
4    T total{};
5
6    for (typename C::const_iterator it=begin(c); it!=end(c); it++)
7      total = total + (*it);
8
9    return total;
10 }
```

C++11 incluye nuevos mecanismos de inferencia estática.
La palabra clave *auto* permite inferir automáticamente el tipo de un dato.

```
1 int  x = 4;
2
3 auto x = 4; //Se infiere que x es un entero
```

C++11 incluye nuevos mecanismos de inferencia estática.
La palabra clave *auto* permite inferir automáticamente el tipo de un dato.
Es más útil cuanto más complicada es la definición del tipo:

```
1 std::vector<int> vec;
2
3 // Sin auto
4 std::vector<int>::iterator itr = vec.begin();
5 // Con auto
6 auto                        itr = vec.begin();
```

C++11 incluye nuevos mecanismos de inferencia estática.
La palabra clave *auto* permite inferir automáticamente el tipo de un dato.
No significa que el dato definido como *auto* pueda almacenar cualquier tipo de dato.
Sólo que delegamos en el compilador para decidir su tipo en función del lado derecho de la asignación.

```
1 // Incorrecto
2 auto itr;
3 itr = 1;
4 itr = vec.begin();
5 // OK
6 auto itr = vec.begin();
```

C++11 incluye nuevos mecanismos de inferencia estática.
La palabra clave *decltype* permite inferir el tipo de dato resultante de una expresión.

```
1 int x = 4;
2
3 decltype(x) y = 1; //y es un entero, como x
```

Es más útil cuanto más complicada es la expresión:

```
1 decltype(max(a,b)) c;
```

Diseña en C++ una función que sume un contenedor:
usando tipos `template` en el template.

```
1  template<typename T,template<typename> typename C>
2  auto sum(const C<T>& c)
3  {
4    T total{};
5
6    for (const auto& e : c)
7      total = total + e;
8
9    return total;
10 }
```

En C++11, los *type traits* establecen mecanismos de inferencia estática.

```
#include <iostream>
#include <type_traits>
class A {};
int main( ) {
    std::cout << std::is_pointer<A>::value << '\n';
    std::cout << std::is_pointer<A *>::value << '\n';
    std::cout << std::is_pointer<A &>::value << '\n';
    std::cout << std::is_pointer<int>::value << '\n';
    std::cout << std::is_pointer<int *>::value << '\n';
    std::cout << std::is_pointer<int **>::value << '\n';
    std::cout << std::is_pointer<int[10]>::value << '\n';
}
```

Su implementación es sencilla especializando templates.

```
1  template<typename T> struct is_pointer {
2    static const bool value = false;
3  };
4
5  template<typename T> struct is_pointer<T*> {
6    static const bool value = true;
7  };
8
9  template<typename T> struct is_pointer<const T*> {
10   static const bool value = true;
11 };
12
13 ...
```

Se puede utilizar para aserciones estáticas (errores en tiempo de compilación)

```
1 template<typename T> struct is_pointer {...};
2 template <typename T>
3 void i_require_a_pointer (T t) {
4     static_assert(std::is_pointer<T>::value, "T must be a pointer type
5     ...
6 }
```

Se puede extender a posteriori para nuevos tipos de datos

```
1  template<typename T> struct is_pointer { ... }
2
3  template<typename T> struct is_pointer<std::shared_ptr<T>> {
4    static const bool value = true;
5  };
6  template<typename T> struct is_pointer<std::shared_ptr<const T>> {
7    static const bool value = true;
8  };
9  ...
```

¿Se puede utilizar para elegir entre dos implementaciones?

```
1 template<typename T> struct is_pointer ....
2 template<typename T> T maximum(T a, T b) {
3    if (is_pointer<T>::value)
4        return ((*a)>(*b))?a:b;
5    else
6        return (a>b)?a:b;
7 }
```

¿Se puede utilizar para elegir entre dos implementaciones?

```
1 template<typename T> struct is_pointer ....
2 template<typename T> T maximum(T a, T b) ...
3 int main() {
4     float f1 = 3, f2 = 5;
5     std::cout<<maximum(f1,f2)<<std::endl;
6     std::cout<<*maximum(&f1,&f2)<<std::endl;
7 }
```

```
1 > make
2 error: invalid type argument of unary ‘’* (have ‘’float)
3             return ((*a)>(*b))?a:b;
4                        ~^~~
```

Se puede utilizar para elegir entre dos implementaciones:

```
1  #include<type_traits>
2  template<typename T> struct is_pointer ....
3  //Overload for when T is not a pointer type
4  template <typename T>
5  typename std::enable_if<!is_pointer<T>::value,T>::type
6    maximum(T a, T b) {
7      return (a>b)?a:b;
8  }
9  //Overload for when T is a pointer type
10 template <typename T>
11 typename std::enable_if<is_pointer<T>::value,T>::type
12   maximum(T a, T b) {
13     return ((*a)>(*b))?a:b;
14 }
```

Se puede utilizar para elegir entre dos implementaciones (C++17, más sencillo)

```
1 #include<type_traits>
2 template<typename T> struct is_pointer ....
3 template<typename T>
4 T maximum(T a, T b) {
5    if constexpr (is_pointer<T>::value)
6       return ((*a)>(*b))?a:b;
7    else
8       return (a>b)?a:b;
9 }
```

Problema

Diseña en C++ una función que sume todos los elementos de una lista, vector, array o conjunto (un contenedor de C++).

```
??? sum(const ??? & contenedor)
```

Problema

Diseña en C++ una función que sume todos los elementos de una lista, vector, array o conjunto (un contenedor de C++).

```
??? sum(const ??? & contenedor)
```

Condiciones extra:

- La colección puede estar vacía (devuelve el elemento nulo).
- El tipo de dato de la colección puede no ser un tipo de dato numérico al uso (nueva clase con el operador +).

Problema

Diseña en C++ una función que sume todos los elementos de una lista, vector, array o conjunto (un contenedor de C++).

```
??? sum(const ??? & contenedor)
```

Consulta `std::accumulate(...)`.

Java 10 incluye nuevos mecanismos de inferencia estática.
La palabra clave *var* permite inferir automáticamente el tipo de un dato.

```
1 int  x = 4;
2
3 var  x = 4; //Se infiere que x es un entero
```

Sólo para variables locales.

Java 10 incluye nuevos mecanismos de inferencia estática.
La palabra clave *var* permite inferir automáticamente el tipo de un dato.
Esto tiene mayor utilidad conforme más complicada es la definición del tipo:

```
1 // Sin var
2 ArrayList<Persona> l = new ArrayList<>();
3 // Con var
4 var l = new ArrayList<Persona>();
```

- Otros mecanismos de inferencia estática en Java son las restricciones en programación genérica.
- Java no incluye más mecanismos específicos de inferencia estática, se prefiere la dinámica.

Introspección

Introspección

Introspección (eng. *introspection*) es el término empleado para describir el proceso en que un objeto puede averiguar información sobre sí mismo u otros.

Permite obtener información sobre la estructura de un objeto, como nombres y tipos de sus componentes, que métodos ofrece o qué interfaces implementa, o de qué otros tipos se deriva.

La introspección en tiempo de compilación se realiza mediante inferencia estática (templates, por ejemplo).

C++ proporciona cierta introspección (limitada) en tiempo de ejecución:

```
1  int    myint  = 50;
2  string mystr  = "string";
3  double *myptr = nullptr;
4
5  cout << "myint has type: " << typeid(myint).name() << '\n'
6       << "mystr has type: " << typeid(mystr).name() << '\n'
7       << "myptr has type: " << typeid(myptr).name() << '\n';
```

Cualquier otra ïntrospección”se deberá diseñar manualmente (ej. herencia).

C++ proporciona cierta introspección (limitada) en tiempo de ejecución:

```
1 template<typename U,typename V>
2 void fun(U u,V v)
3 {
4   const std::type_info& tu = typeid(u);
5   const std::type_info& tv = typeid(v);
6   if (tu != tv) {
7     cout << tu.name() << " " << tv.name() << endl;
8   }
9 }
```

Cualquier otra ïntrospección”se deberá diseñar manualmente (ej. herencia).

Java implementa la idea de la introspección como una parte de su biblioteca de Reflectividad (java.util.reflection).

La utilidad fundamental es la clase **Class**.

La clase Class

A toda clase o interfaz en java le corresponde un objeto que representa su información (**metaclase**), de tipo `Class`.
A través de esa clase podemos:

- explorar el espacio de clases
- averiguar que clases están cargadas
- averiguar que interfaces implementa una clase
- instanciar un objeto de un tipo especificado en tiempo de ejecución
- cargar nuevas clases

La clase Class

Podemos obtener la metaclase correspondiente a una clase dada por distintas vías:

- mediante su nombre:

```
1 Class metaClass = Class.forName("Server");
```

- mediante un literal de clase:

```
1 Class metaClass = Server.class;
```

- mediante un método de `Object`:

```
1 Class metaClass = obj.getClass();
```

En cualquier caso obtenemos un objeto de tipo `Class`.

¿Qué podemos hacer?

Crear un objeto

```
1  ArrayList<Class> nodeClasses = ....
2  try {
3    nodeClasses.add(Class.forName("Desktop"));
4    nodeClasses.add(Class.forName("Server"));
5    nodeClasses.add(Class.forName("Printer"));
6    nodeClasses.add(Class.forName("Phone"));
7  } catch(ClassNotFoundException e) {
8    System.err.println("Cannot find class");
9    throw e;
10 }
11
12 Node n = (Node)nodeClasses[1].newInstance();
```

¿Qué podemos hacer?

Averiguar su nombre e información sobre sus atributos:

```
1  interface Member {
2    String getName();
3  }
4
5  class Field implements Member {
6    Class   getType();
7    String  toString();
8  }
```

```
1  class Class {
2    String  getName();
3    Field[] getFields();
4    Field   getField(String name);
5    Field[] getDeclaredFields();
6    Field   getDeclaredField(String name);
7  }
```

Reflectividad

Reflectividad

Reflectividad (eng. *reflection*) hace referencia al proceso por el que un objeto es capaz de modificarse a sí mismo o a otros.

Permite obtener información sobre la estructura de un objeto, como nombres y tipos de sus componentes, y cambiar sus valores, llamar a métodos por su nombre textual, o en casos extremos modificar el código de esos métodos.

Reflectividad

La Reflectividad se implementa en Java permitiendo que ciertos valores inspeccionados mediante la clase Class se puedan modificar.

Ahora no nos preocupamos sólo de los tipos, sino también de los valores de los atributos.

Modificar una clase incluye llamar a un método de esa clase, o crear objetos nuevos de ese tipo.

Reflectividad

Con introspección y reflectividad podemos inspeccionar el valor asociado a un atributo en un objeto y modificarlo:

```
1  import java.lang.reflect.*;
2  class TestA {
3    double  value = 2.0;
4    public static void main(String[] args) {
5      Class mc = TestA.class;
6      try {
7        Field f = mc.getDeclaredField("value");
8        TestA o = new TestA();
9        System.out.println("before: "+f.get(o));
10       f.set(o,3.14);
11       System.out.println("after:  "+f.get(o));
12     } catch(Throwable e) {
13       System.out.println(e);
14     }
15   }
16 }
```

Inferencia de Tipos

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20