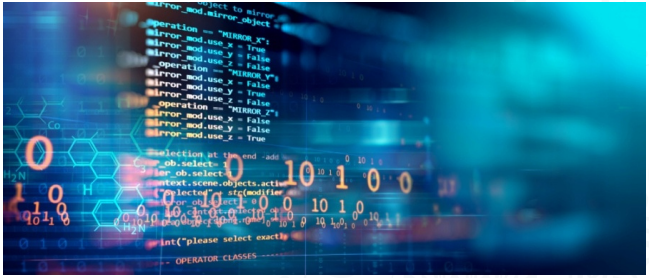
Genéricos y Herencia

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

Genéricos y Herencia

Problema

Dado un sensor de un tipo concreto (real, complejo, . . .), se desea hacer un seguimiento exhaustivo de sus diferentes estadísticas calculadas mediante diferentes acumuladores.

Diseña una clase `Logger` que implemente:

- Un conjunto de almacenes de distintos tipos, especificados por el usuario.
- Un método `push(??? v)` para actualizar todos los almacenes.
- Un método `log()` que dará información del sensor.

(. . .más detalles en el enunciado completo. . .)

Las reglas de la **herencia** son complementarias pero independientes de las reglas de la **programación genérica**.
Una clase genérica puede heredar de otra clase genérica.

```
1 template<typename T>
2 class B
3 { ... };
4
5 template<typename U>
6 class D: public B<U>
7 { ... };
```

La herencia con clases genéricas no implica que ambas deban ser genéricas.

```
1  template<typename T>
2  class B
3  { ... };
4
5
6
7  class D: public B<int>
8  { ... };
9
10 D thing;
```

```
1
2  class B
3  { ... };
4
5
6  template<typename U>
7  class D: public B
8  { ... };
9
10 D<int> thing;
```

Las reglas de la **visibilidad** cambian ligeramente cuando se implican clases genéricas.

```
1
2  class B
3  {
4  protected:
5    int x;
6  };
```

```
1
2  class D : public B
3  {
4    // 'x' es visible
5
6  public:
7    virtual void out() const
8    { cout << x << endl; }
9  };
```

Las reglas de la **visibilidad** cambian ligeramente cuando se implican clases genéricas.

```
1  template<typename T>
2  class B
3  {
4  protected:
5    T x;
6  };
```

```
1
2  class D : public B<int>
3  {
4    // 'x' es visible
5
6  public:
7    virtual void out() const
8    { cout << x << endl; }
9  };
```

Las reglas de la **visibilidad** cambian ligeramente cuando se implican clases genéricas.

```
1
2  class B
3  {
4  protected:
5    int x;
6  };
```

```
1  template<typename T>
2  class D : public B
3  {
4    // 'x' es visible
5
6  public:
7    virtual void out() const
8    { cout << x << endl; }
9  };
```

Las reglas de la **visibilidad** cambian ligeramente cuando se implican clases genéricas.

```
template<typename T>
class B
{
protected:
  T x;
};
```

```
template<typename T>
class D : public B<T>
{
  // 'x' es invisible

public:
  virtual void out() const
  { cout << x << endl; }
};
```

Las reglas de la **visibilidad** cambian ligeramente cuando se implican clases genéricas.

```
template<typename T>
class B
{
protected:
  T x;
};
```

```
template<typename T>
class D : public B<T>
{
  // 'x' se hace visible
  using B<T>::x;
public:
  virtual void out() const
  { cout << x << endl; }
};
```

Las reglas de la herencia se siguen aplicando a los datos parámetro.

```
template<typename T>
class Ptr {
  T* addr;
};

class A { ... };

class B: public A
{ ... };
```

```
void main() {
   A a; Ptr<A> pa;
   B b; Ptr<B> pb;

   pa.addr = &a; //OK
   pb.addr = &b; //OK
   pa.addr = &b; //OK
   //pb.addr = &a; <- ERROR
}
```

Clases abstractas e interfaces (Java) pueden ser genéricos:

```
1  interface Generator<T> {
2    void seed(T s);
3    T    eval(float t);
4  }
5
6  // Lanzador de dados
7  class Dice implements Generator<Integer> {
8    ...
9  }
10 // Generador de onda
11 class Wave implements Generator<Float> {
12   ...
13 }
```

Clases abstractas e interfaces (Java) pueden ser genéricos:

```
1 interface Generator<T> {
2   void seed(T s);
3   T    eval(float t);
4 }
```

Se especifica el conjunto de métodos que debe implementar el interfaz, con tipos aun no definidos pero relacionados.

Si dos tipos mantienen una relación de herencia, parecería lógico suponer que tipos derivados de ellos también estuvieran relacionados:

```
1  class String extends Object;
2
3  String s = new String("hello");
4  Object o = s;
```

Si dos tipos mantienen una relación de herencia, parecería lógico suponer que tipos derivados de ellos también estuvieran relacionados, pero:

```
1  class String extends Object;
2
3  List<String> ls = new ArrayList<String>();
4  List<Object> lo = ls;
5
6  lo.add(new Object());
7  String s = ls.get(0);
```

Ejemplo: escribir todos los elementos de una colección:

```
1  class Collection<T> {
2    ...
3  }
4
5  void print(Collection<Object> c) {
6    for (Object e : c)
7      System.out.println(e);
8  }
```

¿Problemas?

Wildcards

Mediante los denominados ***wildcards*** (sólo en Java):

```
1  class Collection<T> {
2    ...
3  }
4
5  void print(Collection<?> c) {
6    for (Object e : c)
7      System.out.println(e);
8  }
```

El tipo `Collection<?>` es la superclase de todas las posibles instancias del genérico `Collection<T>`

Los *wildcards* (Java) tienen un uso limitado, normalmente como parámetros de métodos. Por ejemplo, esto no es posible en Java:

```
1  Collection<?> c = new ArrayList<String>();
2
3  c.add(new Object()); // Compile time error
```

¿Por qué?

De nuevo, ¿cuál es el problema?

```
1  abstract class Shape {
2    abstract void draw();
3  }
4
5  class Circle extends Shape { ... }
6  class Rectangle extends Shape { ... }
7
8  void drawAll(List<Shape> shapes) {
9    for (Shape s: shapes)
10     s.draw();
11 }
```

OK, usamos *wildcards*. ¿ Es válida esta solución ?

```
1  abstract class Shape {
2    abstract void draw();
3  }
4
5  class Circle extends Shape { ... }
6  class Rectangle extends Shape { ... }
7
8  void drawAll(List<?> shapes) {
9    for (Shape s: shapes)
10     s.draw();
11 }
```

Podemos acotar también los wildcards.

```
1  abstract class Shape {
2    abstract void draw();
3  }
4
5  class Circle extends Shape { ... }
6  class Rectangle extends Shape { ... }
7
8  void drawAll(List<? extends Shape> shapes) {
9    for (Shape s: shapes)
10     s.draw();
11 }
```

Mientras tanto, en C++. . .

```
1  class Shape {
2    virtual void draw() const = 0;
3  };
4  class Circle : public Shape {
5    void draw() const override;
6  };
7  void draw_all(const list<Shape*>& shapes) {
8    for (Shape* s : shapes) s->draw();
9  }
```

¿ Que ocurre para `list<Circle*>` ?

Mientras tanto, en C++. . .

```
1  class Shape {
2    virtual void draw() const = 0;
3  };
4  class Circle : public Shape {
5    void draw() const override;
6  };
7
8  template<typename S>
9  void draw_all(const list<S*>& shapes) {
10   for (S* s : shapes) s->draw();
11 }
```

Consecuencias

C++

La implementación de los genéricos no es completamente real: consiste en replicar el código de la clase o método genéricos para cada instanciación.

Esto implica que:

- Pueden invocarse constructores.
- No tiene sentido imponer restricciones. Los errores de compilación, sin embargo, son difíciles de localizar.
- Puede usarse *metaprogramación*.

Java

La implementación de los genéricos no es completamente real: el compilador, después de verificar los tipos, sustituye las clases y métodos genéricos por versiones que trabajan con `Object` (o el tipo base en tipos acotados), e inserta las conversiones forzadas (*casts*) adecuadas.

Ese proceso se conoce como borrado de tipos o ***type erasure***.

C++

Los tipos se deducen para cada instancia, y se pueden construir variables de dicho tipo. . .

```
1 template<typename T> T average(T a,T b)
2 {
3   T res = 0.5*(a+b);
4   return res;
5 }
```

..si los operadores `'+'` y `'*'` son aplicables al tipo especificado o deducido.

Java

El proceso de *type erasure* hace que el tipo original de la instancia se olvide, con lo que no se pueden crear datos de dicho tipo.

Este código . . .

```
1  <T extends Number> T average(T a,T b)
2  {
3
4    T res = 0.5*(a.doubleValue()+b.doubleValue());
5    return res;
6  }
```

Java

El proceso de *type erasure* hace que el tipo original de la instancia se olvide, con lo que no se pueden crear datos de dicho tipo.

. . . se convierte en este, erróneo . . .

```
1  Number average(Number a,Number b)
2  {
3    // ERROR: Number es una clase abstracta
4    Number res = 0.5*(a.doubleValue()+b.doubleValue());
5    return res;
6  }
```

Java

El proceso de *type erasure* hace que el tipo original de la instancia se olvide, con lo que no se pueden crear datos de dicho tipo.

. . . que se puede .ªrreglar"parcialmente:

```
1  Number average(Number a,Number b)
2  {
3    // Devuelve un tipo probablemente distinto a los de entrada
4    Number res = new Double(0.5*(a.doubleValue()+b.doubleValue()));
5    return res;
6  }
```

C++ ¿ Cuánto vale Box::howmany ?

```
1  template<typename T>
2  class Box<T> {
3    static int howmany;
4    Box() { howmany++; }
5  }
6  template<typename T>
7  int Box<T>::howmany = 0;
```

```
1  Box<int>   bi;
2  Box<float> bf;
```

C++ ¿ Cuánto vale Box::howmany ?

```
1  template<typename T>
2  class Box<T> {
3    static int howmany;
4    Box() { howmany++; }
5  }
6  template<typename T>
7  int Box<T>::howmany = 0;
```

```
1  Box<int>   bi;
2  Box<float> bf;
```

Existe un contador para cada tipo:

```
1    Box<int>::howmany      Box<float>::howmany
```

Java ¿ Cuánto vale `Box.howmany` ?

```
1  class Box<T> {
2    static int howmany = 0;
3    Box() { howmany++; }
4  }
```

```
1  Box<Integer> bi = new Box<>();
2  Box<Float>   bf = new Box<>();
```

Java ¿ Cuánto vale `Box.howmany` ?

```
1  class Box<T> {
2    static int howmany = 0;
3    Box() { howmany++; }
4  }
```

```
1  Box<Integer> bi = new Box<>();
2  Box<Float>   bf = new Box<>();
```

Java sólo crea una implementación de la clase genérica.

Los datos estáticos son compartidos por todas las versiones instanciadas de la clase.

El contador `Box.howmany` es el mismo para ambos tipos.

C++ ¿ De qué tipo es `Box::latest` ?

```
1  template<typename T>
2  class Box {
3    static T latest;
4    T data;
5    Box(T d) : data(d) { latest = data;}
6  };
7  template<typename T>
8  T Box<T>::latest;
```

```
1  Box<int>   bi(3);
2  Box<float> bf(5.0);
```

C++ ¿ De qué tipo es `Box::latest` ?

```
1  template<typename T>
2  class Box {
3    static T latest;
4    T data;
5    Box(T d) : data(d) { latest = data;}
6  };
7  template<typename T>
8  T Box<T>::latest;
```

```
1  Box<int>   bi(3);
2  Box<float> bf(5.0);
```

Del tipo especificado en la instanciación del template:

```
1    int   Box<int>::latest;
2    float Box<float>::latest;
```

Java ¿ De qué tipo es `Box.latest` ?

```
1  class Box<T> {
2    static T latest;
3    T data;
4    Box(T d)
5    { data = d; latest = data; }
6  }
```

```
1  Box<Integer> bi = new Box<>(3);
2  Box<Float>   bf = new Box<>(5.0);
```

Java ¿ De qué tipo es `Box.latest` ?

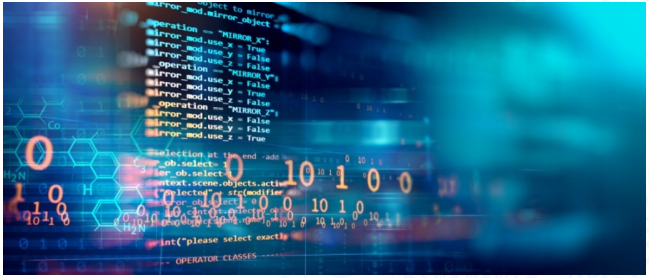
```
1  class Box<T> {
2    static T latest;
3    T data;
4    Box(T d)
5    { data = d; latest = data; }
6  }
```

```
1  Box<Integer> bi = new Box<>(3);
2  Box<Float>   bf = new Box<>(5.0);
```

Java sólo crea una implementación de la clase genérica.
No se permiten declaraciones de nuevos datos del tipo genérico, sólo parámetros.

Genéricos y Herencia

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