

Introduzione ai generics

**Polimorfismo parametrico
vs
polimorfismo per inclusione**

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - ♦ Push(element)
 - ♦ Pop()
 - ♦ Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - ♦ Implementazione con array

Una soluzione

■ Sfruttare Object

```
public class ObjectStack {  
  
    private Object[] stack;  
    private int top = -1;  
  
    public ObjectStack( int capacity ) { stack = new Object [capacity]; }  
  
    public void push( Object el ) { stack[++top] = el; }  
  
    public Object pop() { return stack[top--]; }  
}
```

**Polimorfismo per inclusione,
Polimorfismo di sottotipo**

Una soluzione

■ Esempio di uso #1

```
ObjectStack so = new ObjectStack( 10 );
```

```
so.push("a");
```

```
so.push("b");
```

```
so.push("c");
```

```
String msg = "";
```

```
for (int i = 0; i<3; i++ ) {
```

```
    String x = (String) so.pop();
```

```
    msg += x;
```

```
}
```

```
System.out.println(msg);
```

Se non faccio il downcast,
non posso assegnare a String

cba

output

Una soluzione

■ Esempio di uso #2

```
ObjectStack so = new ObjectStack( 10 );

so.push("a");
so.push("b");
so.push( Integer.valueOf(42) );
String msg = "";
for (int i = 0; i<3; i++ ) {
    String x = (String) so.pop();
    msg += x;
}
System.out.println(msg);
```

Se mi distraigo compila ancora ma...

output

java.lang.**ClassCastException**: java.lang.Integer cannot be cast to java.lang.Integer

Una soluzione migliore: *generics*

- Ovvero una **classe *parametrica***

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack( int capacity ) { stack = new ElemType[ capacity ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Parametro formale di tipo

Ci piacerebbe...
ma non si può fare!

Una soluzione migliore: *generics*

■ Esempio di uso #1

```
GenericStack< String > gs = new GenericStack< String >( 10 );
```

Parametro *attuale* di tipo

```
gs.push("a");
```

```
gs.push("b");
```

```
gs.push("c");
```

```
String msg = "";
```

```
for (int i = 0; i<3; i++ ) {
```

```
    String x = gs.pop();
```

```
    msg += x;
```

```
}
```

```
System.out.println(msg);
```

Non serve downcast:
pop() restituisce **String**

cba

output

Una soluzione migliore: *generics*

■ Esempio di uso #2

```
GenericStack< String > gs = new GenericStack< String >( 10 );
```

```
gs.push("a");
```

```
gs.push("b");
```

```
gs.push( Integer.valueOf(42) );
```

```
...
```

push(String)

1. ERROR in Test.java (at line 8)

```
gs.push( Integer.valueOf(42) );
```

```
^^^^
```

output del compilatore!

The method push(String) in the type GenericStack<String> is not applicable for the arguments (Integer)

Confronto

- Pro del polimorfismo parametrico:
 - ♦ Non richiede controlli di tipo a run time
 - ♦ Anticipa scoperta errori di tipo a tempo di compilazione
- Pro del polimorfismo per inclusione
 - ♦ Permette strutture dati eterogenee
 - ♦ Ad esempio, Stack di elementi di tipo diverso

Prendere il meglio

- In molti casi

- ♦ Servono collezioni di elementi eterogenei
- ♦ Ma vogliamo usarli allo stesso modo

- Esempio:

- ♦ Una scena è una *lista* di forme eterogenee (rettangoli, ellissi, linee, ecc.)
- ♦ Vogliamo usarla per implementare *refresh*, che deve solo inviare un messaggio *draw()* a tutte le forme della lista

- Quindi

- ♦ Identificare le modalità d'uso degli elementi
- ♦ Scegliere una superclasse comune (o fattorizzarle in una *interfaccia*)
- ♦ Usare la superclasse/interfaccia come argomento del template

Note sull'implementazione

- In Java – dopo la compilazione - diventa comunque uno stack di Object
 - ♦ I parametri di tipo vengono usati per il type checking e poi **cancellati** (*erasure*)
 - ♦ Per questo un parametro attuale di tipo non può essere primitivo come int o float, ma dobbiamo usare i wrapper
 - ♦ Per questo non si può istanziare un array di un tipo parametrico

Note sull'implementazione

- In C++, ogni uso di un template fa compilare una nuova classe
 - ♦ Sorta di macro
 - ♦ Il compilatore inserisce nel programma la definizione di classe specializzata e la ricompila per ogni parametro attuale
 - ♦ Più espressivo ma anche più “costoso”
 - ♦ Si può usare un parametro di tipo in tutti i modi in cui si potrebbe usare un tipo vero

Array parametrici in Java

- Per riuscire a compilare:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack( int capacity ) { stack = (ElemType[]) new Object[ capacity ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```


Siamo costretti a un downcast

- Otteniamo comunque un warning a tempo di compilazione

Array parametrici in Java

- Per riuscire a compilare:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    @SuppressWarnings("unchecked")  
    public GenericStack( int capacity ) { stack = (ElemType[]) new Object[ capacity ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```



Annotazione

- Ora niente warning
- Meglio ancora: usare `ArrayList<ElemType>`

2. Astrarre un'API tramite interfaccia

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - ♦ Push(element)
 - ♦ Pop()
 - ♦ Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - ♦ **Non “forzare” una specifica implementazione**
 - **Se ne occupa un altro team**
 - **Serve un modo per far lavorare i due team indipendentemente ma garantendo l'integrazione dei risultati**

Soluzione: interfacce

- Definisco un'interfaccia con i metodi richiesti

```
public interface Stack<ElemType> {  
    public void push( ElemType el );  
    public ElemType pop();  
}
```

Implementare l'interfaccia

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack ( int capacity ) { ... }  
  
    public void push( ElemType el ) { stack [++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Utilizzare l'interfaccia

■ Esempio

```
Stack< Integer > si = new GenericStack< Integer >( 10 );
```

```
si.push(1);  
si.push(2);  
si.push(3);  
for( int i = 0; i<3; i++) {  
    System.out.println( si.pop().intValue() );  
}
```

Posso cambiare implementazione
semplicemente cambiando qs tipo.
Il resto del codice resta uguale.

3
2
1

output

Confronto con classi astratte

- Se avessi definito Stack come una classe astratta...
 - ♦ ...GenericStack avrebbe dovuto estendere Stack
 - ♦ In Java, una classe può estendere una sola altra classe
 - ♦ Restringe l'insieme di classi che si possono utilizzare
- Vantaggi delle interfacce
 - ♦ Non impone lo stesso obbligo
 - ♦ Le classi che implementano l'interfaccia Stack possono essere ovunque nella gerarchia delle classi
- Vantaggi delle classi astratte
 - ♦ Posso contenere dati (aka *stato*)

3. Esempio di Stack con eccezioni

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - ♦ Push(element)
 - ♦ Pop()
 - ♦ Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - ♦ Non “forzare” una specifica implementazione
 - ♦ **Gestire in modo pulito gli errori specifici**
 - **Push su stack pieno**
 - **Pop su stack vuoto**

Soluzione: eccezioni

- Cosa succede attualmente con troppi push o troppi pop?

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack ( int capacity ) { ... }  
  
    public void push( ElemType el ) { stack [++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Soluzione: eccezioni

- Introduciamo due eccezioni custom, non verificate

```
public class EmptyStackException extends RuntimeException {  
    public EmptyStackException() {}  
    public EmptyStackException(String msg) { super(msg); }  
}
```

```
public class FullStackException extends RuntimeException {  
    public FullStackException() {}  
    public FullStackException(String msg) { super(msg); }  
}
```


Implementare l'interfaccia

- Lanciamo le eccezioni

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack ( int capacity ) { ... }  
  
    public void push( ElemType el ) {  
        if ( top < stack.length-1 ) stack[++top] = el;  
        else throw new FullStackException()  
    }  
  
    public ElemType pop() {  
        if( top >= 0 ) return stack[ top-- ];  
        else throw new EmptyStackException();  
    }  
}
```

Utilizzo

■ Esempio di svuotamento stack

```
Stack< Integer > si = new GenericStack< Integer >();
```

```
si.push(1);
```

```
si.push(2);
```

```
si.push(3);
```

```
try {
```

```
    while( true ) {
```

```
        System.out.println( si.pop().intValue() );
```

```
    }
```

```
} catch( EmptyStackException e ) {
```

```
    System.out.println( "fine" );
```

```
}
```

Non serve downcast:
pop() restituisce Integer



3

2

1

fine

output