GENERICS

::.. Generics

Java Generics consente ai tipi (classi ed interfacce) di essere **parametrizzati** quando si specificano classi, interfacce e metodi. La sintassi è del tutto similare ai parametri formali usati nella dichiarazione di metodo. I parametri di tipo rappresentano un modo di riusare lo stesso codice con differenti input. La differenza è che l'input dei parametri formali sono dei valori, mentre quelli dei **parametri di tipo sono dei tipi**.

I generics consentono, quindi, di operare delle astrazioni sui tipi di dati gestiti, e ben noti casi d'impiego sono i tipi **contenitori**, come nella gerarchia **Collections**.

Questa possibilità è stata introdotta a partire dalla JDK 5.0, e condivide delle similarità, ma anche differenze, con costrutti similari in altri linguaggi di programmazione, come i **template in C++**.

::.. Generics - Benefici

Il codice che impiega i generics fornisce dei benefici rispetto a codice che non ne fa uso:

- Un controllo di tipo più forte a tempo di compilazione
  - Il compilatore Java effettua dei **forti controlli** di tipo al codice generico e può sollevare **eccezioni** in caso in cui il codice violi la safety di tipo;
  - Risolvere errori a **tempo di compilazione** è più facile rispetto a risolverli a **tempo di esecuzione**, dove è più complesso trovare la riga di codice che ha generato l'errore.

::.. Generics - Benefici

Il codice che impiega i generics fornisce dei benefici rispetto a codice che non ne fa uso:

- Un controllo di tipo più forte a tempo di compilazione
- Eliminazione del casting

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

::.. Generics - Benefici

Il codice che impiega i generics fornisce dei benefici rispetto a codice che non ne fa uso:

- Un controllo di tipo più forte a tempo di compilazione
- Eliminazione del casting

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Il casting è necessario al fine di esplicitare il tipo degli elementi nella lista, altrimenti si solleva un errore di compilazione.

::.. Generics - Benefici

Il codice che impiega i generics fornisce dei benefici rispetto a codice che non ne fa uso:

- Un controllo di tipo più forte a tempo di compilazione
- Eliminazione del casting

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

::.. Generics - Benefici

Il codice che impiega i generics fornisce dei benefici rispetto a codice che non ne fa uso:

- Un controllo di tipo più forte a tempo di compilazione
- Eliminazione del casting

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Il casting **non è più necessario**, dato che è stato esplicitato in fase di dichiarazione il tipo degli elementi in lista.

```
List<String> list = n      ayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

::.. Generics - Benefici

Il codice che impiega i generics fornisce dei benefici rispetto a codice che non ne fa uso:

- Un controllo di tipo più forte a tempo di compilazione
- Eliminazione del casting

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

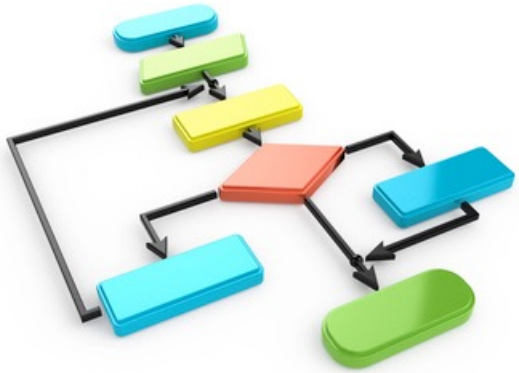
Sembra che il cast dalla riga 3 si sia spostata alla 1, ma in realtà il cambiamento è radicale visto che ora il compilatore può controllare la correttezza del codice.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

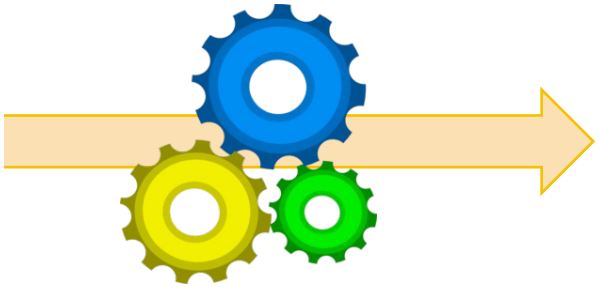
::.. Generics - Benefici

Il codice che impiega i generics fornisce dei benefici rispetto a codice che non ne fa uso:

- Un controllo di tipo più forte a tempo di compilazione
- Eliminazione del casting
- Consente ai programmatori l'implementazione di **algoritmi generici**
  - Usando i generics, i programmatori possono implementare algoritmi che lavorano su collezioni di tipi differenti, che sono type safe e facili da leggere.



Algoritmo di ordinamento



Applicato a

Numbers

Strings

Others

::.. Tipi generici

Un tipo generico è una classe o interfaccia che è stata parametrizzata rispetto ai tipi delle variabili che impiega.

Consideriamo una classe contenitore «fittizia» di nome «Box» per dimostrare concretamente questo concetto. Questa classe deve operare su oggetti di ogni tipo, e fornisce solo due metodi di inserimento e rimozione.

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

::.. Tipi generici

Un tipo generico è una classe o interfaccia che è stata parametrizzata rispetto ai tipi delle variabili che impiega.

Consideriamo una classe contenitore «fittizia» di nome «Box» per dimostrare concretamen
su oggetti di ogni tipo,
rimozione.

```
public class Box {
   private Object object;

   public void set(Object object) { this.object = object; }
   public Object get() { return object; }
}
```

Siccome i metodi accettano e ritornano istanze di Object, possono gestire qualunque tipo di dato, a parte istanze dei tipi primitivi. **Non esiste modo a tempo di compilazione di controllarne l'uso**.

È possibile che una parte del programma carichi (**set**) un dato di tipo Integer, mentre da un'altra parte ci si aspetta di estrarre (**get**) un dato di tipo String, con conseguente errore.

::.. Tipi generici

Una classe generica è definita come:

class name<T1, T2, ..., Tn> { /* ... */ }

::.. Tipi generici

Una classe generica è definita come:

class name<T1, T2, ..., Tn> { /* ... */ }

La sezione di parametrizzazione di tipo è delimitata da parentesi uncinate, e segue il nome della classe. Definisce una serie di parametri di tipo o anche variabili di tipo T1, T2, …, Tn, che possono essere usate in ogni punto all'interno della specifica della classe.

::.. Tipi generici

Una classe generica è definita come:

class name<T1, T2, ..., Tn> { /* ... */ }

Ad esempio, una classe «wrapper» Box può essere resa generica con l'introduzione della variabile di tipo T:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

::.. Tipi generici

Una classe generica è definita come segue:

class name<T1, T2, ..., Tn> { /* ... */ }

Ad esempio, una classe «wrapper» Box può essere resa generica con l'introduzione della variabile di tipo T:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

La variabile di tipo è collocata dove precedentemente avevamo Object, e rappresenta **ogni possibile tipo, non primitivo**: ogni possibile classe, interfaccia, array o anche un altra variabile di tipo.

::.. Tipi generici

Una classe generica è definita come segue:

class name<T1, T2, ..., Tn> { /* ... */ }

Lo stesso può essere fatto per la specifica di **interfacce**.

Ad esempio, una classe «wrapper» Box può essere resa generica con l'introduzione della variabile di tipo T:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

::.. Generics - Convenzioni sui Nomi

Secondo convenzione, i nomi dei parametri di tipo sono singole lettere in maiuscolo, in contrasto con la convenzione sui nomi di variabili. Questo per poter distinguere la differenza tra una variabile di tipo e una classe/interfaccia ordinaria.

I nomi di parametri di tipo più usati sono:

- E – Elemento
- K – Chiave
- N – Numero
- T – Tipo
- V – Valore
- S,U,V etc. – secondo, terzo, quarto tipo

::.. Generics vs Templates

Anche se molto simili nella sintassi e nell'utilizzo, i Java generics non sono la stessa cosa dei template C++:

•I template C++ si riconducono a **macro del preprocessore**, che **producono il codice sorgente** di una nuova classe con i tipi "fissati".

•Generics opera a livello di **compilatore** e non "sporca" il codice della classe che si sta utilizzando.

::.. Cancellazione del Tipo

Per **implementare i generics**, il compilatore Java effettua un processo detto “cancellazione del tipo” (**type erasure**) :

- Sostituisce tutti i parametri di tipo nei tipi generici con Object (se i parametri non sono vincolati), oppure con i vincoli del tipo. Il **bytecode generato**, quindi, contiene solo classi, interfacce e metodi ordinari;
- Inserisce **cast di tipo** per preservare la safety di tipo;
- Inserisce dei metodi ponte per preservare il **polimorfismo** quando si definiscono tipi generici con l’ereditarietà.

La cancellazione del tipo assicura che **nessuna nuova classe venga generata per i tipi parametrizzati**, quindi non sussiste un overhead a tempo di esecuzione.

::.. Cancellazione del Tipo

Per comprendere la cancellazione di tipi generici consideriamo la seguente classe generica:

```
public class Node<T> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) }
        this.data = data;
        this.next = next;    }
    public T getData() { return data; }
    // ...
}
```

::.. Cancellazione del Tipo

Per comprendere la cancellazione di tipi generici consideriamo la seguente classe generica:

```
public class Node<T> {
   private T data;
   private Node<T> next;
   public Node(T data, Node<T> next) }
      this.data = data;
      this.next = next;    }
   public T getData() { return d
   // …
}
```

```
public class Node {
   private Object data;
   private Node next;
   public Node(Object data, Node next) {
      this.data = data;
      this.next = next;    }
   public Object getData() { return data; }
   // …
}
```

Siccome il parametro T non è vincolato, questo viene sostituito con Object:

::.. Restrizioni Generics

Esistono 7 restrizioni per l'uso dei generics:

1. **Non è possibile istanziare tipi generici con tipi primitivi**
2. **Non è possibile istanziare (con new) i parametri di tipo**
3. **Non è possibile dichiarare dei campi statici i cui tipi sono parametri di tipo**
4. **Non è possibile usare cast o instanceof con tipi parametrizzati**
5. **Non è possibile creare array di tipi parametrizzati**
6. **Non è possibile intercettare o sollevare eccezioni con oggetti di tipi parametrizzati**
7. **Non è possibile sovraccaricare un metodo dove i tipi dei parametri formali di ogni sovraccarico cancellano lo stesso tipo raw**

::.. Tipi Raw (1/3)

Un **tipo raw** è una classe o interfaccia generica in cui non è indicato alcun parametro di tipo (ciò non vale per classi o interfacce non generiche).
Nel caso del precedente esempio della classe generica Box:

```
public class Box<T> {    public void set(T t) { /* ... */ }    // ...}
```

Per creare un tipo parametrizzato di Box<T>:

```
Box<Integer> intBox = new Box<>();
```

Se l'argomento di **tipo** viene **omesso**, si ha un **tipo raw** di Box<T>:

```
Box rawBox = new Box();
```

Box è il tipo raw del tipo generico **Box<T>**.

I tipi raw sono spesso impiegati per compatibilità in **codice legacy**, perché molte delle classi di API (come le classi di Collections) non erano generiche prima di JDK 5.0.

::.. Tipi Raw (2/3)

Quando si usano i tipi raw, si ottiene un comportamento come prima dell'avvento di generics, in cui Box **restituiva istanze di Object**. Per la compatibilità all'indietro, assegnare un tipo parametrizzato a un tipo raw è consentito:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;            // OK
```

Se si assegna un tipo raw ad uno parametrizzato si ottiene un **warning** in compilazione:

```
Box rawBox = new Box();          // rawBox is a raw type Box<Integer>
intBox = rawBox;                 // warning: unchecked conversion
```

Si ha un warning se si usa un tipo raw per invocare metodi generici definiti nel corrispondente tipo generico:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);           // warning: unchecked invocation to set(T)
```

::.. Tipi Raw (3/3)

Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

Questi warning mostrano che i tipo raw **bypassano i controlli dei tipi generici**, rinviando il catch di codice non sicuro a **tempo di esecuzione**. Questo perché il compilatore non dispone di sufficienti informazioni per effettuare i necessari controlli di tipo per garantire type safety.
Pertanto, è buona pratica **evitare l'uso di tipi raw**.

È possibile vedere tutti gli "unchecked" warnings ricompilando il codice Java con –Xlint:unchecked, che restituisce il riferimento alla riga che ha sollevato il warning.

È possibile disabilitare tali warnings usando il flag –Xlint:-unchecked.

::.. Generics ed Ereditarietà (1/4)

È possibile assegnare un oggetto di un tipo T1 a un oggetto di un altro tipo T2, a condizione che i tipi siano **compatibili** (ad esempio assegnare un Integer a un Object), ovvero che **T1 sia un super-tipo per T2**

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

Dal momento che un Intero è un tipo di Object, l'assegnazione è consentita. Ma Integer è anche un tipo di Number, quindi questo codice non presenta errori:

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));      // OK
someMethod(new Double(10.1));   // OK
```

Questo è vero anche con i generics.

::.. Generics ed Ereditarietà (2/4)

È possibile effettuare un'invocazione di un tipo generico, passando Number come suo argomento di tipo, e ogni conseguente invocazione del metodo 'add' sarà consentita se l'argomento è compatibile con Number:

```
Box<Number> box = new Box<Number>();

box.add(new Integer(10));    // OK
box.add(new Double(10.1));   // OK
```

Consideriamo adesso questo metodo:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

Che tipo di argomento accetta?

Vedendo la sua firma, accetta Box<Number>, andrà bene anche Box<Integer> o Box<Double>?

::.. Generics ed Ereditarietà (3/4)

È possibile effettuare un'invocazione di un tipo generico, passando Number come suo argomento di tipo, e ogni conseguente invocazione del metodo 'add' sarà consentita se l'argomento è compatibile con Number:

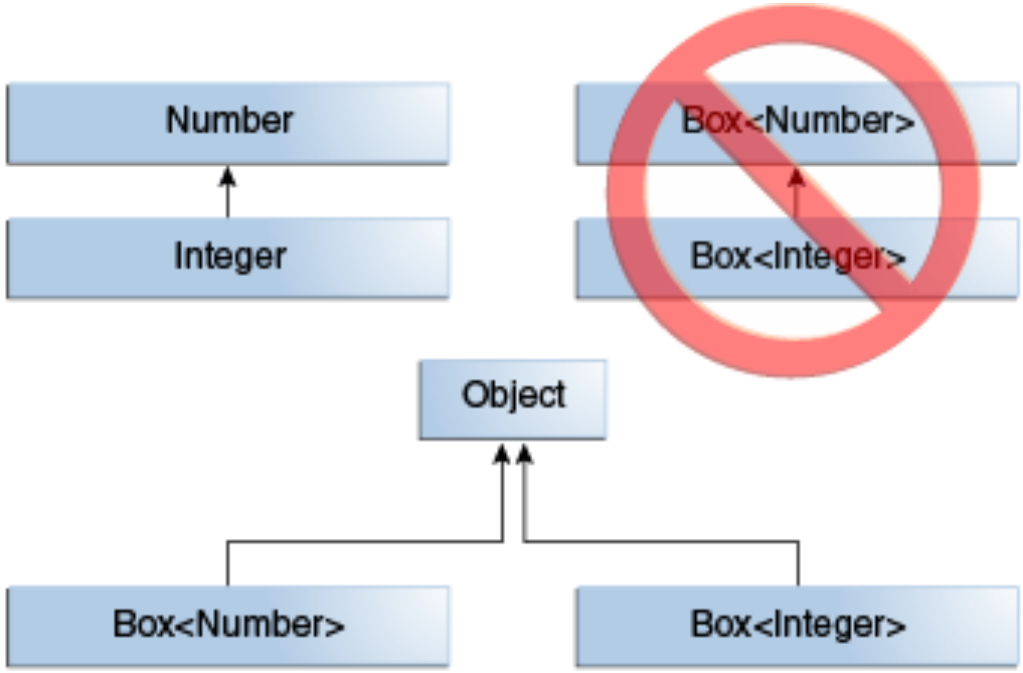
```
Box<Number> box = new Box<Number>();

box.add(new Integer(10));    // OK
box.add(new Double(10.1));   // OK
```

Consideriamo adesso questo metodo:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

La risposta è no! Questi due tipi non sono sotto-tipi di Box<Number>, sebbene parametrizzati con tipi che hanno come super-tipo Number.

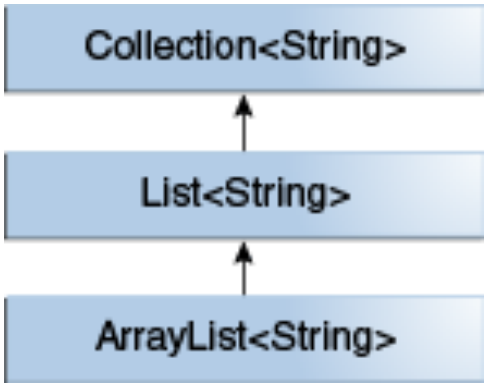
::.. Generics ed Ereditarietà (3/4)



Dati due tipi concreti A e B, MyClass<A> non ha nessun legame con MyClass<B>, la classe parente in comune tra questi due tipi parametrizzati è solo Object.

::.. Generics ed Ereditarietà (4/4)

È possibile **derivare classi generiche** e **implementare interfacce generiche**, allo stesso modo delle classi e interfacce tradizionali. Fintanto che **non si modificano gli argomenti di tipo**, la relazione di ereditarietà è preservata tra i tipi.



Consideriamo che ArrayList<E> implementa List<E> che deriva da Collection<E>.

Per cui, ArrayList<String> è un sotto-tipo di List<String>, che lo è di Collection<String>.

::.. Wildcards

In classi con tipi generici, la sintassi del punto di domanda («**?**»), chiamato **wildcard**, rappresenta un tipo sconosciuto.

È possibile usare una wildcard per **rilassare le restrizioni su una variabile**. Ipotizziamo che si vuole scrivere un metodo che può lavorare su **List<Integer>**, **List<Double>** e **List<Number>**. Ciò è possibile usando una wildcard limitata superiormente.

Per dichiarare questo tipo di wildcard, basta usare il carattere '?' seguita da '**extends**' e il limite superiore ('extends' è inteso sia come estensione di classe che implementazione di un'interfaccia).

::.. Wildcards

Una wildcard può essere usata in vari modi: come tipo per un parametro, campo, o variabile locale, certe volte come tipo di ritorno (sebbene è buona pratica essere più specifici).

Una wildcard non è mai impiegata come argomento di tipo nell'invocazione di un metodo generico, nella creazione di un'istanza di una classe generica o come super-tipo.

::.. Wildcards

Come esprimere il metodo che può lavorare su List<Integer>, List<Double> e List<Number>?

List<? extends Number>

Oppure

List<Number>

Quale è il più adatto?

::.. Wildcards

Come esprimere il metodo che può lavorare su List<Integer>, List<Double> e List<Number>?

List<? extends Number>

Oppure

List<Number>

Indica una lista di tipo Number o di ognuno dei suoi sottotipi.

::.. Wildcards

Come esprimere il metodo che può lavorare su List<Integer>, List<Double> e List<Number>?

List<? extends Number>

Oppure

List<Number>

Indica **solo** una lista di tipo Number.

::.. Wildcards

Come esprimere il metodo che può lavorare su List<Integer>, List<Double> e List<Number>?

List<? extends Number>

Consideriamo il seguente metodo e la sua invocazione:

```
public static void process(List<? extends Number> list) {
   for (Number elem : list) {
      // ...
   }
}
List<Integer> list = new ArrayList<>();
list.add(10);
process(list)
```

Nella clausola foreach, la variabile elem itera su ogni elemento nella lista. Ogni metodo definito nella classe Number può essere invocato da elem.

::.. Wildcards

Come esprimere il metodo che può lavorare su List<Integer>, List<Double> e List<Number>?

List<Number>

Consideriamo il seguente metodo e la sua invocazione:

```
public static void process(List<Number> list) {
    for (Number elem : list) {
        // ...
    }
}
List<Integer> list = new ArrayList<>();
list.add(10);
process(list)
```

Errore di compilazione: *The method process(List<Number>) is not applicable for the arguments (List<Integer>)*

::.. Wildcards

Ecco un diagramma delle relazioni tra varie liste con wildcard limitati superiormente ed inferiormente: