



UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Università degli Studi di Salerno

Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2023-2024

Programmazione Generica

Docente: Prof. Massimo Ficco

E-mail: mficco@unisa.it

Introduzione

Se un algoritmo o struttura dati può essere rappresentato in maniera indipendente dai dettagli relativi alla rappresentazione dei dati, allora può essere rappresentato come un concetto generale parametrizzato sul tipo dati. Questo paradigma prende il nome di **Programmazione generica**.

Nel linguaggio C e durante il corso di PSD questo è stato ottenuto con void* e il linking di implementazioni di un interfaccia specificata in un header file.

```
#include "item.h"

typedef struct list *List;

List newList();
int isEmpty(List);
Item removeHead(List);
int sizeList(List);
void printList(List);
int addTail(List, Item);
```

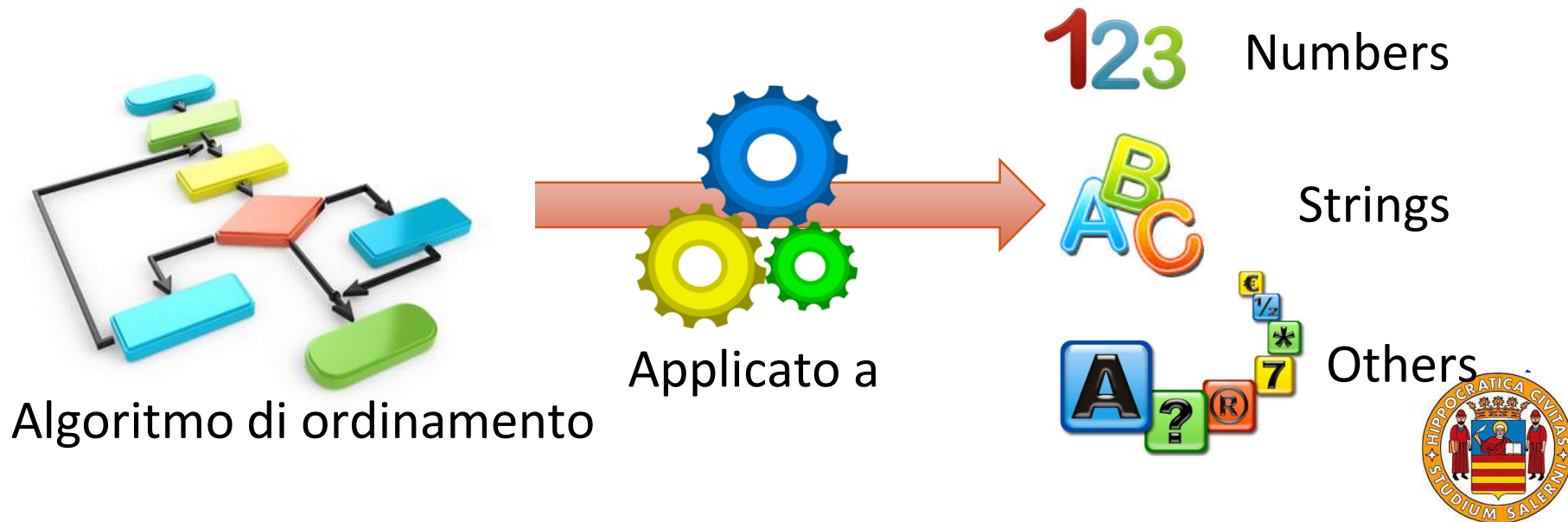
```
typedef void *Item;

Item inputItem();
void outputItem(Item);
int cmplItem(Item, Item);
Item cloneItem(Item);
```



Introduzione

- ▶ Il principale problema di questa soluzione è che richiede molto effort al programmatore e non ha nessun controllo da parte del compilatore sulla type safety della soluzione, causando possibili errori in esecuzione.
- ▶ Molti algoritmi sono per loro natura generici, visto che possono essere applicati a qualsiasi tipo di dati, eseguendo sempre gli stessi passi. Per poterli realizzare genericamente basterebbe parametrizzare il tipo su cui lavorano.



Problema

- Immaginiamo di voler implementare una funzione di massimo:

```
void main() {  
    cout << " max(10, 15) = " << max(10, 15) << endl;  
    cout << " max('k', 's') = " << max('k', 's') << endl;  
    cout << " max(10.1, 15.2) = " << max(10.1, 15.2) << endl;  
}
```



Problema

- Immaginiamo di voler implementare una funzione di massimo:

```
void main() {  
    cout << " max(10, 15) = " << max(10, 15) << endl;  
    cout << " max('k', 's') = " << max('k', 's') << endl;  
    cout << " max(10.1, 15.2) = " << max(10.1, 15.2) << endl;  
}
```

Per poter eseguire questo codice è necessario implementare tre differenti funzioni.

```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```

```
char max(char a, char b) {  
    return a > b ? a : b;  
}
```

```
float max(float a, float b) {  
    return a > b ? a : b;  
}
```



Problema

- Immaginiamo di voler implementare una funzione di massimo:

```
void main() {  
    cout << " max(10, 15) = " << max(10, 15) << endl;  
    cout << " max('k', 's') = " << max('k', 's') << endl;  
    cout << " max(10 1 15 2) = " << max(10 1 15 2) << endl;  
}
```

È uno spreco, perchè in molti casi cambia solo il tipo dei dati di lavoro mentre le istruzioni sono sempre le stesse.



```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```

```
char max(char a, char b) {  
    return a > b ? a : b;  
}
```

```
float max(float a, float b) {  
    return a > b ? a : b;  
}
```



Problema

- Un'alternativa, utilizzando un approccio C, è definire una macro e lasciare che il preprocessore la espanda.

```
#define MAX(a,b) (((a)>(b))?(a):(b))
```

- Questo è generalmente corretto per i tipi di base, ma cosa succede se gli argomenti macro sono più complessi o possiamo ottenere effetti collaterali indesiderati?



Problema

- ▶ Un'alternativa, utilizzando un approccio C, è definire una macro e lasciare che il preprocessore la espanda.

```
#define MAX(a,b) (((a)>(b))?(a):(b))
```

- ▶ Questo è generalmente corretto per i tipi di base, ma cosa succede se gli argomenti macro sono più complessi o possiamo ottenere effetti collaterali indesiderati?
- ▶ Poiché la macro viene valutata dal preprocessore (prima della compilazione), funziona esclusivamente sul testo del codice sorgente, il che significa:
 - ▶ Il tipo dei parametri non sono controllati;
 - ▶ I parametri non vengono valutati.



Problema

- Consideriamo il seguente codice:

```
#define MAX(a,b) (((a)>(b))?(a):(b))

int main() {
    int i = 10;
    int j = 5;
    int m = MAX(++i, ++j);
}
```



Che valore assume m?



Problema

- Consideriamo il seguente codice:

```
#define MAX(a,b) (((a)>(b))?(a):(b))  
  
int main() {  
    int i = 10;  
    int j = 5;  
    int m = MAX(++i, ++j);  
}
```

Che valore assume m?

Eseguendo il codice il valore non è 11, ma 12. Perché?



Problema

- Consideriamo il seguente codice:

```
#define MAX(a,b) (((a)>(b))?(a):(b))
```

```
int main() {  
    int i = 10;  
    int j = 5;  
    int m = MAX(++i, ++j);  
}
```

Compilazione

```
int main() {  
    int i = 10;  
    int j = 5;  
    int m = (++i) > (++j) ? (++i) : (++j);  
}
```



Che valore assume m?

La variabile viene valutata due volte: una durante il test e una durante il ramo selezionato dell'istruzione di scelta.



Introduzione a Java Generics

- ▶ **Java Generics** consente ai tipi di essere parametrizzati quando si specificano classi, interfacce e metodi. La sintassi è del tutto simile 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 perché prima si pensava sufficiente il ricorso a Object, e condivide delle similarità, ma anche differenze, con costrutti simili in altri linguaggi di programmazione, come i template in C++.



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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 degli errori 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.



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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);
```



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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);
```

Si assume una lista di istanze di Object.



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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.



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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
```



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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 = new ArrayList<String>();
```


```
list.add("hello");
```

```
String s = list.get(0); // no cast
```



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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
```



Introduzione a Java Generics

- ▶ Il codice che impiega i Generics ottengono 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.



Tipi Generici in Java

- ▶ Un tipo generico è una classe o interfaccia che è stata parametrizzata rispetto ai tipi delle variabili che impiega.
- ▶ Consideriamo una classe Box per dimostrare concretamente questo concetto. Questa classe deve operare su oggetti di ogni tipo (non primitivo), 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; }  
}
```



Classi Generici in Java

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

- Consideriamo un'istanza di Box carichi nell'oggetto un dato di tipo Integer, mentre da un'altra parte ci si aspetta di estrarre un dato di tipo String, con conseguente errore.

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



Classi Generici in Java

- Modifichiamo la classe Box così da renderla generica con l'introduzione della variabile di tipo T:

```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```



Classi Generici in Java

- Modifichiamo la classe Box così da renderla generica con l'introduzione della variabile di tipo T:

```
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.



Classi Generici in Java

- Modifichiamo la classe Box così da renderla generica con l'introdu

```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

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



Classi Generici in Java

- ▶ Per riferirsi alla classe generica Box, bisogna realizzare un'invocazione di un tipo generico, che consiste nel sostituire al parametro T un valore concreto come Integer:

Box<Integer> integerBox;

- ▶ Tale invocazione può essere pensata come ad un'invocazione ordinaria di un metodo, ma invece di passare un argomento al metodo, viene passato un argomento di tipo alla classe stessa.
- ▶ Come ogni istanziazione, non viene creato un nuovo oggetto Box, ma semplicemente dichiara che integerBox è un riferimento a un'istanza di Box con un elemento Integer. Per l'istanziazione di usa il costrutto new:

Box<Integer> integerBox = new Box<Integer>();

- ▶ Un'invocazione di un tipo generico è generalmente nota come tipo parametrizzato.



Tipi Generici in Java

- ▶ 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



Classi Generici in Java

- Una classe generica su n tipi generici è definita come segue:

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



Classi Generici in Java

- Una classe generica su n tipi generici è definita come segue:

```
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.



Esempi di Tipi Generics - ArrayList

- La classe **ArrayList** è una classe parametrica (programmazione generica)
- In genere un ArrayList ha lo stesso scopo di un array
- A differenza degli array, che una volta creati hanno dimensione fissa, un ArrayList può cambiare la propria dimensione durante l'esecuzione del programma
- Risulta meno efficiente degli array



Esempi di Tipi Generics - ArrayList

Gli elementi contenuti sono di un solo tipo: Object (utilizzo di wrapper per i tipi semplici).

Questo vincolo è meno restrittivo di quanto sembrerebbe: in virtù del subtyping possiamo infatti mettere in un ArrayList istanze qualunque discendente di Object, ovvero qualunque oggetto Java.

Possiamo memorizzare oggetti di classi completamente scorrelate (come String, Rectangle, Persona) nella stessa istanza di ArrayList.

Quando li estraiamo dobbiamo però usare un downcast per passare dal tipo Object al tipo voluto.

```
ArrayList aList = new ArrayList();
```



Esempi di Tipi Generics - ArrayList

- E' contenuto nel package `java.util.*`

`ArrayList` di tipo `T`

```
ArrayList<T> aList = new ArrayList<T>();
```

```
ArrayList<BankAccount> accounts =  
    new ArrayList<BankAccount>();  
accounts.add(new BankAccount(1001));  
accounts.add(new BankAccount(1015));  
accounts.add(new BankAccount(1022));
```

- Il tipo base <T> può essere solo un tipo strutturato (per i tipi semplici bisogna usare delle apposite classi *wrapper*)



Esempi di Tipi Generics - ArrayList

Come per gli array

- gli indici iniziano da 0
- errore se l'indice è fuori range
- posizioni accessibili: 0 .. size() – 1.
- **size()**: il metodo che calcola il numero di elementi nella struttura

Accesso tramite il metodo **get()**

```
BankAccount anAccount = accounts.get(2);  
    // il terzo elemento dalla arraylist
```



Altri metodi

set() sovrascrive un valore esistente

```
BankAccount anAccount = new BankAccount(1729);  
accounts.set(2, anAccount);
```

add() aggiunge un nuovo valore nella posizione i

```
accounts.add(i, a)
```

oppure all'ultima posizione

```
accounts.add(a)
```

remove() rimuove l'elemento all'indice i

```
accounts.remove(i)
```



La classe **Vector<T>**

- La classe **Vector<T>** è simile alla classe **ArrayList<T>**
- E' possibile fare le stesse cose della classe ArrayList, ma ha qualche metodo in più
- E' più vecchia come classe e meno efficiente in termini di prestazione



Classi/Interfacce Generici in Java

- Una classe generica può avere parametri di tipo multipli, come l'esempio di una classe generica `OrderedPair`, che implementa l'interfaccia generica `Pair`:

```
public interface Pair<K, V> {  
    public K getKey();    public V getValue();  
}  
  
public class OrderedPair<K, V> implements Pair<K, V> {  
    private K key;    private V value;  
    public OrderedPair(K key, V value) {  
        this.key = key;    this.value = value;  
    }  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```



Classi/Interfacce Generici in Java

- Una classe generica può avere parametri di tipo multipli, come l'esempio di una classe generica `OrderedPair`, che implementa l'interfaccia generica `Pair`:

```
public interface Pair<K, V> {  
    public K getKey();    public V getValue();  
}
```

```
public class OrderedPair<K, V> implements Pair<K, V> {
```

```
    private K key;    private V value;
```

```
    public OrderedPair(K key, V value) {
```

```
        this.key = key;    this.value = value;
```

```
    }
```

```
    public K getKey() { return key; }
```

```
    public V getValue() { return value; }
```

```
}
```

I parametri di tipo dell'interfaccia
sono gli stessi della classe generica



Classi/Interfacce Generici in Java

- ▶ Le seguenti righe di codice creano due istanze della classe `OrderedPair`:
 - ▶ `Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);`
 - ▶ `Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");`



Tipi Generici in Java

- ▶ Le seguenti righe di codice creano due istanze della classe OrderedPair:
 - ▶ `Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);`
 - ▶ `Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");`

K è risolto come String e V come Integer, per questo i parametri del costruttore sono una stringa ed un intero.



Tipi Generici in Java

- ▶ Le seguenti righe di codice creano due istanze della classe OrderedPair:
 - ▶ `Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);`
 - ▶ `Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");`

K è risolto come String e V come String, per questo i parametri del costruttore sono due stringhe.



Tipi Generici in Java

- ▶ Le seguenti righe di codice creano due istanze della classe OrderedPair:
 - ▶ `Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);`
 - ▶ `Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");`



Come mai nella prima riga di codice al costruttore si passa un intero e non un'istanza della classe Integer? Il compilatore solleva errore? Il codice è type safe?

Il codice non solleva errore grazie al fenomeno dell'Autoboxing, ovvero la conversione automatica che il compilatore Java opera tra i tipi primitivi e le loro corrispondenti classi wrapper.



Metodi Generici in Java

- ▶ Si dicono generici quei metodi che introducono dei propri parametri di tipo, in maniera simile ai tipi generici, ma lo scopo del parametro di tipo è limitato al metodo dove è stato dichiarato. Metodi statici e non-statici sono consentiti, come i costruttori di classi generici.
- ▶ La sintassi per un metodo generico include un parametro di tipo, all'interno delle parentesi uncinate, prima del suo tipo di ritorno. Tali metodi possono essere definiti in classi ordinarie o generiche, nel secondo caso non devono esplicitamente dichiarare i parametri di tipo se sono quelli della classe generica.

```
public class Util {  
    // Generic static method  
    public static <K, V> boolean compare(Pair<K, V> p1,  
        Pair<K, V> p2) {  
        return p1.getKey().equals(p2.getKey()) &&  
            p1.getValue().equals(p2.getValue());    }  
}
```



Metodi Generici in Java

```
public class Pair<K, V> {  
    private K key;  
    private V value;
```

// Generic constructor

```
public Pair(K key, V value) {  
    this.key = key;  
    this.value = value; } }
```

// Generic methods

```
public void setKey(K key) { this.key = key; }  
public void setValue(V value) { this.value = value; }  
public K getKey() { return key; }  
public V getValue() { return value; }  
}
```



Metodi Generici in Java

- La sintassi completa per invocare questi metodi è la seguente:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
```

```
Pair<Integer, String> p2 = new Pair<>(2, "pear");
```

```
boolean same = Object.<Integer, String>compare(p1, p2);
```

Import java.util.Objects;
Objects.compare() utile quando
gli oggetti non sono comparabili



Metodi Generici in Java

- La sintassi completa per invocare questi metodi è la seguente:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
```

```
Pair<Integer, String> p2 = new Pair<>(2, "pear");
```

```
boolean same = Object.<Integer, String>compare(p1, p2);
```

Il tipo è stato fornito esplicitamente. Generalmente, però, questo non è necessario, e si può omettere.

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
```

```
Pair<Integer, String> p2 = new Pair<>(2, "pear");
```

```
boolean same = Object.compare(p1, p2);
```



Parametri Limitati

- ▶ È possibile che sussista l'esigenza di limitare i tipi che possono essere usati come argomenti in un tipo parametrizzato. Ad esempio, una determinata operazione su numeri implementata come metodo generico potrebbe dover essere applicata solo su numeri, quindi si vorrebbe che il metodo accettasse solo istanze di Number o sue sottoclassi. A questo scopo esistono i parametri di tipo limitati.
- ▶ Per dichiarare un parametro limitante come in esempio basta far seguire al nome del parametro la parola chiave 'extends' seguita dal limite superiore, che nell'esempio è Number. In questo senso, 'extends' assume sia il significato di estensione tra classi che di implementazione di interfacce.
- ▶ Consideriamo il caso della classe generica Box, introducendo un metodo generico con un parametro limitante su Number.



Parametri Limitati

```
public class Box<T> {  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
  
    // Metodo generico con parametri limitati  
    public <U extends Number> void inspect(U u){  
        System.out.println("T: " + t.toString());  
        System.out.println("U: " + u.toString()); }  
  
    public static void main(String[] args) {  
        Box<String> StringBox = new Box<String>();  
        StringBox.set("some text1");  
        StringBox.inspect("some text2");  
    }  
}
```



Parametri Limitati

```
public class Box<T> {  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```



Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
be applied to (java.lang.String)

```
    StringBox.inspect("some text2");  
                ^
```

1 error

```
public static void main(String[] args) {  
    Box<String> StringBox = new Box<String>();  
    StringBox.set("some text1");  
    StringBox.inspect("some text2");  
}
```



Parametri Limitati

- ▶ In precedenza abbiamo visto l'uso di un vincolo singolo, ma è possibile definire più di un vincolo sul parametro di tipo:

```
<T extends B1 & B2 & B3>
```

- ▶ Il tipo accettabile per il parametro di tipo T deve essere un sotto-tipo di tutti quelli presenti nel vincolo. Se uno di essi è una classe, deve essere specificata per prima:

```
class A { /* ... */ }
```

```
interface B { /* ... */ }
```

```
interface C { /* ... */ }
```

```
class D <T extends A & B & C> { /* ... */ }
```

- ▶ Se il vincolo su A non è specificata per prima si ha un errore di compilazione:

```
class D <T extends B & A & C> { /* ... */ } // compile-time error
```

