# Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

**Capitolo 10: Tipi generici e collezioni**

**Autore: Tommaso C.**

---

Appunti tratti dalle lezioni in aula e dal materiale del corso

---

Argomenti: Tipi generici · Generici e sottoclassi · Caratteri jolly · Cancellazione
Autoboxing · Java Collections Framework · Iteratori
Mappe · Collezioni ordinate · Implementazioni

# Indice

**1 Perché i tipi generici** **1**

**2 Definire e usare un tipo generico** **2**

**3 Generici e sottoclassi** **3**

**4 Il carattere jolly** **3**

**5 La cancellazione dei tipi** **4**

**6 Tipi primitivi e autoboxing** **5**

**7 Il Java Collections Framework** **5**

**8 `Collection`, `Set` e `List`** **6**

**9 Iteratori** **8**

**10 Mappe** **10**

**11 Collezioni ordinate** **10**

**12 Le implementazioni** **12**

**13 Collezioni e schemi realizzativi** **13**

**14 Esercizi** **14**

**15 Soluzioni** **16**

# 1 Perché i tipi generici

**Definizione – Tipo generico**

Un **tipo generico** è una classe, un'interfaccia o un metodo **parametrico** rispetto ai tipi dei dati su cui opera. Il parametro si scrive fra parentesi angolari: `Coppia<T>` è una coppia di elementi di un tipo `T` non ancora fissato.

Due classi come `CoppiaDiInteri` e `CoppiaDiStringhe` hanno la stessa logica: due campi, un costruttore, `getPrimo` e `getSecondo`. Cambia solo il tipo. Prima di Java 5 si scriveva una sola classe con campi di tipo `Object`, che può contenere qualsiasi oggetto:

```
public class Coppia {
    private Object primo;
    private Object secondo;
    public Coppia(Object a, Object b) { primo = a; secondo = b; }
    public Object getPrimo() { return primo; }
    public Object getSecondo() { return secondo; }
}
...
Coppia c = new Coppia("ciao", "mondo");
String primo = (String) c.getPrimo();      // cast obbligatorio
String secondo = (String) c.getSecondo();
```

Questa soluzione ha due difetti:

1. ogni lettura richiede un cast esplicito;
2. il compilatore non controlla che gli elementi siano dello stesso tipo: `new Coppia(1, "mondo")` compila, e il cast a `String` del primo elemento fallisce solo in esecuzione. Per evitarlo servono controlli con `instanceof`, scritti a mano.

Con il tipo generico la classe si scrive una volta, e il tipo si fissa quando la si usa:

```
public class Coppia<T> {
    private T primo;
    private T secondo;
    public Coppia(T a, T b) { primo = a; secondo = b; }
    public T getPrimo() { return primo; }
    public T getSecondo() { return secondo; }
}
...
Coppia<String> c = new Coppia<String>("ciao", "mondo");
String primo = c.getPrimo();               // nessun cast
```

Da qui in poi il compilatore tratta ogni `T` di `c` come `String`. Non servono cast, e la creazione di una coppia con `1` e `"mondo"` è un errore di compilazione. Si ha la generalità di `Object` con il controllo dei tipi di Java.

## 2 Definire e usare un tipo generico

**Definizione – Parametri di tipo formali e attuali**

- Nella dichiarazione `class C<T>`, `T` è il **parametro di tipo formale**, come il parametro formale di un metodo.
- Nell'uso `C<String>`, `String` è il **parametro di tipo attuale**: sostituisce `T` in tutta la classe.

Dentro la classe, `T` si usa dove si userebbe un altro tipo: campi, parametri di costruttori e metodi, tipi restituiti.

```
class C<T> {
    private T info;
    public C(T info) { this.info = info; }
    public T getInfo() { return info; }
    public void setInfo(T info) { this.info = info; }
}
```

Per usarla si istanzia `T` con una classe. Il resto del codice è codice tipato come sempre:

```
C<String> c1 = new C<String>("ciao");
C<Persona> c2 = new C<Persona>(new Persona(...));
String s = c1.getInfo();
Persona p = c2.getInfo();
```

Il nome del parametro formale è un identificatore qualsiasi. Le librerie Java seguono però queste convenzioni, che aiutano a leggere il codice:

| Lettera | Significato |
|---|---|
| `E` | elemento (usato in tutte le collezioni) |
| `K` | chiave |
| `V` | valore |
| `N` | numero |
| `T` | tipo |
| `S`, `U`, `V` | secondo, terzo e quarto parametro |

## 3 Generici e sottoclassi

Gli esempi usano una pila generica e una sua derivata:

```
class Pila<T> {
    public void inserisci(T elem) {...}
    public T rimuovi() {...}
}
class PilaCollegata<T> extends Pila<T> {...}
```

**Proposizione – Ereditarietà e tipi generici**

1. Se `D<T>` deriva da `C<T>`, a un riferimento di tipo `C<T>` si può assegnare un oggetto `D<T>`, con lo *stesso* parametro attuale.
2. In un `C<T>` si può mettere un oggetto di qualsiasi sottoclasse di `T`.
3. Fra `C<A>` e `C<B>` non c'è ereditarietà, anche se `A` deriva da `B`: `Pila<Integer>` non è una `Pila<Number>`.

```
Pila<Number> pn = new PilaCollegata<Number>();   // OK, punto 1
pn.inserisci(Integer.valueOf(1));                 // OK, punto 2
pn.inserisci(Double.valueOf(0.5));                // OK, punto 2

Pila<Integer> pi = new Pila<Integer>();
Pila<Number> pn2 = pi;                            // ERRORE di compilazione,
    punto 3
```

Il punto 3 evita un errore che altrimenti si vedrebbe solo più tardi. Se l'assegnazione fosse lecita, si potrebbe scrivere:

```
Pila<Number> po = pi;                 // supponiamo che compili
po.inserisci(Double.valueOf(0.5));    // un Double entra nella pila di Integer
Integer i = pi.rimuovi();             // un Double assegnato a un Integer
```

`po` e `pi` sono la stessa pila. Attraverso `po` si inserisce un `Double`, che poi esce da `pi` come se fosse un `Integer`. Per questo il compilatore rifiuta già la prima riga (Esercizio 1).

## 4 Il carattere jolly

**Definizione – Carattere jolly**

Dato un tipo generico `C<T>`, il tipo `C<?>` (**carattere jolly**, *wildcard*) comprende tutte le istanziazioni di `C`: un `C<?>` è un `C` di elementi di tipo sconosciuto.

```
Pila<Integer> pi = new Pila<Integer>();
pi.inserisci(Integer.valueOf(1));
Pila<?> pu = pi;                    // OK: pila di elementi sconosciuti
Object o = pu.rimuovi();            // OK, ma il tipo restituito è Object
pu.inserisci(Integer.valueOf(2));   // ERRORE di compilazione
```

Attraverso `pu` si può leggere, ma ciò che esce si conosce solo come `Object`. Non si può invece passare un argomento a un metodo il cui parametro ha tipo `T`: il compilatore non sa quale sia `T`, quindi non può controllare l'argomento.

Il carattere jolly serve quindi soprattutto a scrivere codice che *legge* collezioni create da altri, senza sapere che cosa contengono.

**Carattere jolly limitato.** La forma `<? extends E>` indica un tipo sconosciuto che è `E` o una sua sottoclasse. Si trova nelle collezioni (Sezione 8): `addAll(Collection<? extends E> c)` accetta, per una collezione di `Number`, anche una collezione di `Integer`, perché ogni suo elemento è un `Number`.

## 5 La cancellazione dei tipi

**Definizione – Cancellazione**

I tipi generici esistono solo fino alla compilazione. Il compilatore traduce un tipo generico con la **cancellazione** (*erasure*):

1. sostituisce ogni occorrenza del parametro di tipo con `Object`;
2. inserisce i cast espliciti dove servono;
3. produce nel bytecode una sola copia della classe o del metodo.

Dopo la cancellazione, `Coppia<T>` della Sezione 1 è proprio la `Coppia` con `Object` scritta prima di Java 5, e l'uso diventa:

```
// scritto dal programmatore
Coppia<String> c = new Coppia<String>("ciao", "mondo");
String primo = c.getPrimo();
// tradotto dal compilatore
Coppia c = new Coppia("ciao", "mondo");
String primo = (String) c.getPrimo();
```

I cast che prima si scrivevano a mano li scrive il compilatore, che però li ha già controllati. Il controllo dei tipi avviene in compilazione, non in esecuzione.

## 6 Tipi primitivi e autoboxing

Il parametro attuale di un tipo generico deve essere una classe, perché dopo la cancellazione diventa `Object`. Un tipo primitivo come `int` o `boolean` non è ammesso: si usa la sua classe wrapper (Capitolo 2, Sezione 14).

```
Coppia<int> c1 = new Coppia<int>(1, 2);              // ERRORE di compilazione
Coppia<Integer> c1 = new Coppia<Integer>(1, 2);      // corretto
```

**Definizione – Autoboxing e unboxing**

- L'**autoboxing** è la conversione automatica di un valore primitivo nell'oggetto della sua classe wrapper, dove serve un oggetto.
- L'**unboxing** è la conversione inversa, dove serve un valore primitivo.

Nella seconda riga, `1` e `2` sono `int`, e Java li converte da solo in `Integer`. Nella direzione opposta:

```
int s = c1.getPrimo() + c1.getSecondo();   // unboxing: s vale 3
```

## 7 Il Java Collections Framework

**Definizione – Java Collections Framework**

Il **Java Collections Framework** (JCF) è la libreria del pacchetto `java.util` per lavorare con collezioni di oggetti. È formato da:

- **interfacce**, che rappresentano i tipi di collezione di uso comune: lista, insieme, coda, mappa;
- **implementazioni**, classi concrete che realizzano le interfacce con strutture dati efficienti;
- **algoritmi**, metodi che realizzano operazioni comuni, come la ricerca e l'ordinamento, sulle collezioni del JCF.

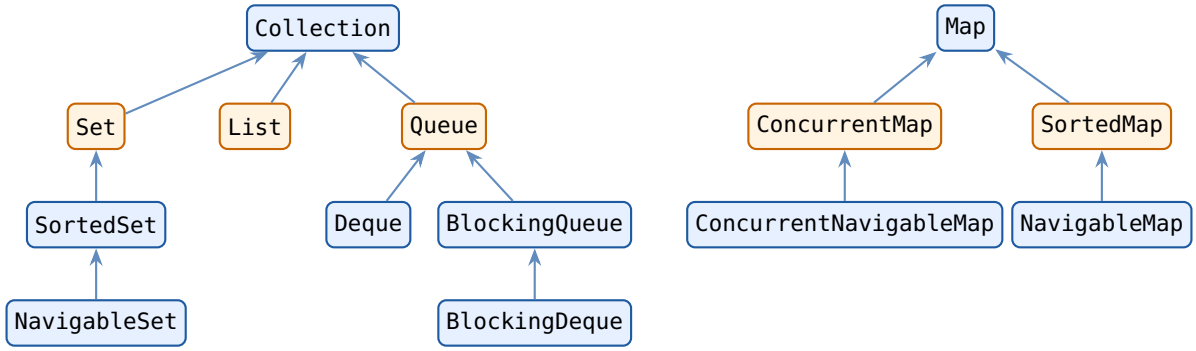
Un'interfaccia è un tipo di dato astratto (Capitolo 6): una lista si può realizzare con un array o con una struttura collegata, e le due classi offrono la stessa interfaccia. I vantaggi del JCF sono tre:

- **generalità**: si cambia l'implementazione di una collezione senza cambiare i clienti, che usano l'interfaccia;
- **interoperabilità**: codice scritto da persone diverse si scambia collezioni dello stesso tipo;

- **efficienza**: le implementazioni sono ottimizzate.

Con i tipi generici, il compilatore controlla anche il contenuto di una collezione. In una `List<Persona>` entrano solo oggetti di classe `Persona` o di una sua sottoclasse.

**Le interfacce.** Le interfacce del JCF hanno due radici, `Collection` e `Map`. La figura mostra la gerarchia: ogni freccia va da un'interfaccia a quella che essa estende.




*Le interfacce del JCF. `Collection` e `Map` sono le radici; ogni freccia va verso l'interfaccia estesa.*

Le interfacce da conoscere sono queste:

| Interfaccia | Rappresenta |
|---|---|
| `Set<E>` | insieme: ogni elemento compare al più una volta |
| `SortedSet<E>` | insieme ordinato |
| `List<E>` | lista o sequenza: elementi in ordine di posizione, anche ripetuti |
| `Queue<E>` | coda: si inserisce in fondo e si estrae dalla testa |
| `Deque<E>` | coda a doppia uscita: si inserisce e si estrae da entrambe le estremità |
| `Map<K,V>` | mappa: coppie (chiave, valore) con chiavi distinte |
| `SortedMap<K,V>` | mappa con chiavi ordinate |

## 8 `Collection`, `Set` e `List`

`Collection<E>` dichiara le operazioni di ogni collezione, in tre gruppi:

```
public interface Collection<E> extends Iterable<E> {
    // operazioni di base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                    // facoltativa
    boolean remove(Object element);            // facoltativa
```

```
    Iterator<E> iterator();
    boolean equals(Object o);
    // operazioni sull'intera collezione (bulk)
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);    // facoltativa
    boolean removeAll(Collection<?> c);           // facoltativa
    boolean retainAll(Collection<?> c);           // facoltativa
    void clear();                                 // facoltativa
    // conversione in array
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

- Un'operazione **facoltativa** (*optional*) può non essere supportata da un'implementazione, che allora lancia `UnsupportedOperationException`.
- `containsAll`, `removeAll` e `retainAll` sono inclusione, differenza e intersezione fra insiemi.
- `toArray()` restituisce un `Object[]`; `toArray(T[] a)` un array del tipo dato, e quindi conserva il tipo degli elementi.

**Osservazione – `Set`**

`Set<E> extends Collection<E>` non aggiunge metodi. Cambia il significato: un insieme non contiene elementi ripetuti, e quindi `add` di un elemento già presente non lo aggiunge e restituisce `false`.

`List<E>` aggiunge l'accesso per posizione, con indici da 0:

```
public interface List<E> extends Collection<E> {
    void add(int index, E element);       // facoltativa
    E get(int index);
    E set(int index, E element);          // facoltativa
    int indexOf(Object o);
    int lastIndexOf(Object o);
    E remove(int index);                  // facoltativa
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);
    boolean addAll(int index, Collection<? extends E> c);   // facoltativa
}
```

Una lista, a differenza di un insieme, può contenere lo stesso elemento più volte. Per questo esistono sia `indexOf`, che dà la prima posizione di un elemento, sia `lastIndexOf`, che dà l'ultima.

# 9 Iteratori

**Definizione – Iteratore**

Un **iteratore** è un oggetto che implementa l'interfaccia `Iterator<E>` e fa da **cursore** su una collezione: la scandisce un elemento alla volta. Il metodo `iterator()` di una collezione restituisce un iteratore posto all'inizio.

```
public interface Iterator<E> {
    boolean hasNext();   // c'è un elemento successivo?
    E next();            // restituisce l'elemento successivo e avanza
    void remove();       // toglie l'ultimo elemento restituito (facoltativa)
}
```

La realizzazione dell'iteratore dipende dalla classe della collezione, ma il programmatore usa solo l'interfaccia. Così scandisce allo stesso modo un array dinamico e una lista collegata, senza conoscerne la struttura. Lo schema tipico è:

```
Collection<E> c = ...;
Iterator<E> it = c.iterator();
while (it.hasNext()) {
    E e = it.next();     // elemento corrente
    ...                  // lo si elabora
}
```

**Proposizione – Regole degli iteratori**

1. Un iteratore va in una sola direzione, in avanti.
2. È "usa e getta": a scansione finita non serve più, e per scandire di nuovo si chiede un iteratore nuovo con `iterator()`.
3. Durante la scansione non si modifica la collezione con i suoi metodi. Altrimenti, alla chiamata successiva di `next()`, l'iteratore lancia l'eccezione `ConcurrentModificationException`. Si può invece usare `remove` dello stesso iteratore.

```
LinkedList<String> a = new LinkedList<String>();
a.addAll(Arrays.asList("A", "B"));
Iterator<String> it = a.iterator();
while (it.hasNext()) {
    it.next();
    it.remove();          // OK: remove dell'iteratore
}
System.out.println(a);    // []

a.addAll(Arrays.asList("A", "B"));
```

```
it = a.iterator();
while (it.hasNext()) {
    a.add("K");           // modifica la lista durante la scansione
    it.next();            // ConcurrentModificationException
}
```

**Iteratore di lista.** `listIterator()` di una lista restituisce un `ListIterator<E>`, che estende `Iterator<E>` e va anche all'indietro:

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void set(E e);     // sostituisce l'ultimo restituito (facoltativa)
    void add(E e);     // inserisce dove sta il cursore (facoltativa)
}
```

`add` non aggiunge in fondo alla lista: inserisce nel punto in cui si trova il cursore. Per scandire una lista dall'ultimo elemento al primo:

```
ListIterator<E> it = c.listIterator(c.size());   // cursore dopo l'ultimo
while (it.hasPrevious()) {
    E e = it.previous();
    ...
}
```

**Definizione – `Iterable` e ciclo `for` migliorato**

Una classe è **iterabile** se implementa `Iterable<E>`, cioè ha il metodo `Iterator<E> iterator()`. Tutte le collezioni del JCF sono iterabili, perché `Collection` estende `Iterable`. Su ogni oggetto iterabile si può usare il ciclo `for` migliorato, come sugli array (Capitolo 3, Sezione 6):

`for (E e : collezione) blocco`

```
List<Integer> l = new ArrayList<Integer>(Arrays.asList(1, 2, 3));
for (Integer i : l)
    System.out.println(i);
```

Il ciclo equivale allo schema con `hasNext` e `next`, che il compilatore genera da solo, ed è altrettanto efficiente. Come sugli array, serve a leggere: modificare la lista dentro il ciclo lancia `ConcurrentModificationException`.

# 10 Mappe

**Definizione – Mappa**

Una **mappa** memorizza coppie (**chiave**, **valore**):

- la chiave serve ad accedere alla coppia ed è unica nella mappa;
- il valore contiene le informazioni associate alla chiave.

Esempi di mappe sono un dizionario, con coppie (parola, significato), una rubrica, con coppie (nominativo, indirizzo e telefono), e una funzione discreta, con coppie $(x, y)$.

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    void clear();                          // facoltativa
    boolean equals(Object o);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);                     // valore della chiave, o null
    V put(K key, V value);                 // facoltativa
    V remove(Object key);                  // facoltativa
    Set<K> keySet();
    Collection<V> values();
}
```

- `put` inserisce la coppia. Se la chiave c'era già, sostituisce il valore e restituisce quello vecchio.
- In una mappa si entra per chiave. La posizione di una coppia non è garantita, né l'ordine di inserimento: se serve un ordine, si usa una `SortedMap`.

**Osservazione – `keySet` e `values`**

`keySet()` restituisce un `Set` perché le chiavi sono distinte. `values()` restituisce una `Collection`, non un `Set`, perché valori uguali possono essere associati a chiavi diverse: in una rubrica, più persone della stessa famiglia hanno lo stesso indirizzo.

# 11 Collezioni ordinate

Il JCF offre due collezioni ordinate:

- `SortedSet<E>`, un insieme ordinato, senza elementi ripetuti;
- `SortedMap<K,V>`, una mappa ordinata per chiave.

Tenere l'ordine rende più costoso l'inserimento, ma più efficienti le ricerche. Conviene quando le letture sono molte più delle scritture.

Il criterio di ordinamento si fissa in uno di due modi.

**Definizione – `Comparable` e `Comparator`**

- La classe degli elementi implementa `Comparable<T>`, con il metodo `int compareTo(T o)`, che confronta `this` con `o`.
- Una classe a parte implementa `Comparator<T>`, con il metodo `int compare(T o1, T o2)`, che confronta `o1` con `o2`.

Il risultato è negativo se il primo oggetto precede il secondo, zero se sono uguali per l'ordinamento, positivo se lo segue.

L'ordinamento lo decide chi progetta la classe: due persone si possono ordinare per cognome, per età o per reddito, secondo ciò che serve all'applicazione.

**Proposizione – Costruttori delle collezioni ordinate**

Le classi che implementano `SortedSet` o `SortedMap` hanno due costruttori:

- senza argomenti: ordinano con `compareTo`, quindi gli elementi devono implementare `Comparable`;
- con un argomento `Comparator`: ordinano con il suo `compare`.

**Esempio – Persone per nome e per età**

`Persona` si ordina per cognome e nome; un comparatore a parte la ordina per età.

```
class Persona implements Comparable<Persona> {
    private String cognome, nome;
    private int eta;
    ...
    public int compareTo(Persona o) {        // per cognome e nome
        int c = cognome.compareTo(o.cognome);
        if (c != 0) return c;
        return nome.compareTo(o.nome);
    }
}
class PersonaComparatorEta implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2) {   // per età
        return Integer.compare(p1.getEta(), p2.getEta());
    }
}
...
SortedSet<Persona> s1 = new TreeSet<Persona>();
SortedSet<Persona> s2 = new TreeSet<Persona>(new PersonaComparatorEta());
```

Con Anna Bianchi di 22 anni, Luca Rossi di 31 e Mario Rossi di 20, scandire `s1` dà Anna Bianchi, Luca Rossi, Mario Rossi; scandire `s2` dà Mario Rossi, Anna Bianchi, Luca Rossi.

**Osservazione – Uguali per l'ordinamento**

Un `TreeSet` considera uguali due elementi per cui il confronto dà zero. In `s1`, una seconda persona chiamata Mario Rossi, anche con un'altra età, non viene aggiunta.

## 12 Le implementazioni

Per ogni interfaccia il JCF offre classi basate su strutture dati diverse:

| Interfaccia | Tavola hash | Array dinamico | Albero bilanciato | Lista collegata |
|---|---|---|---|---|
| `Set` | `HashSet` | | `TreeSet` | `LinkedHashSet` |
| `List` | | `ArrayList` | | `LinkedList` |
| `Deque` | | `ArrayDeque` | | `LinkedList` |
| `Map` | `HashMap` | | `TreeMap` | `LinkedHashMap` |

- `HashSet` e `HashMap` usano una tavola hash; `LinkedHashSet` e `LinkedHashMap` vi aggiungono una lista collegata.

- `TreeSet` e `TreeMap` usano un albero di ricerca bilanciato, che tiene gli elementi in ordine: realizzano `SortedSet` e `SortedMap`.
- `ArrayList` usa un array che si allarga e si restringe da solo; `LinkedList` una lista collegata doppia.
- `Collection` non ha un'implementazione diretta.

Il codice cliente dichiara le variabili con il tipo dell'interfaccia e nomina la classe solo nella `new`. Così, per cambiare implementazione, basta cambiare una riga:

```
List<String> l = new ArrayList<String>();    // oppure new LinkedList<String>()
```

## 13 Collezioni e schemi realizzativi

Con il JCF non si scrivono più a mano insiemi e liste, come `InsiemeFC` del Capitolo 6. Quando però si realizza una propria collezione, per esempio una classe `MioSet<E>` che implementa `Set<E>`, si sceglie uno schema realizzativo (Capitolo 6, Sezione 3).

**Proposizione – Schema per le collezioni del JCF**

Una collezione che implementa un'interfaccia del JCF segue lo schema **con side effect, senza condivisione di memoria**: i metodi come `add` e `remove` modificano l'oggetto di invocazione.

Lo schema segue dalle interfacce: `add` e `remove` restituiscono un `boolean`, non una collezione nuova, quindi devono modificare l'oggetto. Senza condivisione, due collezioni non hanno mai parti della struttura in comune, e la modifica di una non tocca l'altra. Lo schema funzionale con condivisione resta adatto alle strutture collegate scritte a mano fuori dal JCF, come `InsiemeFC`.

**Riepilogo**

- **Tipo generico**: `class C<T>`, con `T` parametro formale; nell'uso `C<String>` il parametro attuale sostituisce `T`, senza cast.
- **Sottoclassi**: `D<T>` va bene dove serve `C<T>`, ma `C<Integer>` non è un `C<Number>`.
- **Carattere jolly**: `C<?>` legge tutto come `Object`, ma non accetta argomenti di tipo `T`; `<? extends E>` accetta sottoclassi di `E`.
- **Cancellazione**: in compilazione `T` diventa `Object` e si aggiungono i cast.
- **Autoboxing**: i parametri attuali sono classi; `int` diventa `Integer` da solo, e viceversa.
- **JCF**: interfacce (`Set`, `List`, `Queue`, `Map`), implementazioni e algoritmi in `java.util`.
- **Iteratore**: `hasNext` e `next`, in avanti, usa e getta; niente modifiche alla collezione durante la scansione; `for` migliorato su ogni `Iterable`.
- **Ordinamento**: `Comparable.compareTo` o `Comparator.compare`; `TreeSet` e `TreeMap` hanno un costruttore per ciascuno.

## 14 Esercizi

**Esercizio 1 – Pile di numeri**

Con la classe `Pila<T>` della Sezione 3, quali righe compilano?

```
Pila<Number> pn = new Pila<Number>();      // (1)
pn.inserisci(Integer.valueOf(3));          // (2)
Pila<Integer> pi = new Pila<Integer>();    // (3)
pn = pi;                                   // (4)
Pila<?> pu = pi;                           // (5)
Number n = pu.rimuovi();                   // (6)
```

Per le righe che non compilano, spiega il motivo.

*Soluzione a pagina 16 →*

**Esercizio 2 – Togliere i numeri pari**

`l` è una `List<Integer>` con i valori 4, 7, 10, 3. Il codice

```
for (Integer x : l)
    if (x % 2 == 0)
        l.remove(x);
```

lancia un'eccezione. Quale, e perché? Riscrivi il ciclo con un iteratore in modo che tolga i numeri pari, lasciando 7 e 3.

*Soluzione a pagina 16 →*

### Esercizio 3 – Una rubrica

Una rubrica associa a ogni nominativo un indirizzo.

1. Dichiara e crea la rubrica come `Map`, scegliendo un'implementazione che tenga i nominativi in ordine alfabetico.
2. Inserisci Rossi, Via Roma 1, e Bianchi, Via Roma 1. Quanti elementi hanno `keySet()` e `values()`?
3. Inserisci poi Rossi, Via Po 2. Che cosa restituisce questa `put`, e quanto vale `get("Rossi")`?
4. Stampa i nominativi con un ciclo `for` migliorato.

*Soluzione a pagina 16 →*

# 15 Soluzioni

### Soluzione dell'Esercizio 1 – Pile di numeri

(1) Compila: stesso parametro attuale a sinistra e a destra.

(2) Compila: `Integer` è una sottoclasse di `Number`, e in una pila di `Number` si può mettere un oggetto di una sottoclasse.

(3) Compila.

(4) Non compila. `Pila<Integer>` non è una `Pila<Number>`: se lo fosse, attraverso `pn` si potrebbe inserire un `Double` nella pila di `Integer pi`, e poi estrarlo da `pi` come se fosse un `Integer`.

(5) Compila: `Pila<?>` accetta ogni istanziazione di `Pila`.

(6) Non compila. Attraverso `pu`, `rimuovi` restituisce un `Object`, perché il tipo degli elementi è sconosciuto, e un `Object` non si assegna a un `Number` senza cast. Si può scrivere `Object n = pu.rimuovi();`.

← *Torna all'Esercizio 1, pagina 14*

### Soluzione dell'Esercizio 2 – Togliere i numeri pari

Il ciclo `for` migliorato usa un iteratore sulla lista. Al primo elemento, 4, `l.remove(x)` modifica la lista con un suo metodo durante la scansione. Alla chiamata successiva di `next()` l'iteratore lancia `ConcurrentModificationException`. Si usa invece `remove` dell'iteratore, che toglie l'ultimo elemento restituito da `next`:

```
Iterator<Integer> it = l.iterator();
while (it.hasNext())
    if (it.next() % 2 == 0)
        it.remove();
// l = [7, 3]
```

`it.next() % 2` funziona per unboxing: `next()` restituisce un `Integer`, che diventa `int`.

← *Torna all'Esercizio 2, pagina 14*

### Soluzione dell'Esercizio 3 – Una rubrica

1. Le chiavi in ordine richiedono una `SortedMap`, realizzata da `TreeMap`. `String` implementa già `Comparable`, quindi basta il costruttore senza argomenti:

   ```
   Map<String, String> rubrica = new TreeMap<String, String>();
   ```

2. `keySet()` ha 2 elementi, Bianchi e Rossi. Anche `values()` ne ha 2, ma sono

uguali: `"Via Roma 1"` due volte. Per questo `values()` è una `Collection` e non un `Set`.

3. La chiave `"Rossi"` c'è già, quindi `put` ne sostituisce il valore e restituisce quello vecchio, `"Via Roma 1"`. Ora `get("Rossi")` vale `"Via Po 2"`, e la mappa ha ancora 2 coppie.
4. 
```
for (String nome : rubrica.keySet())
    System.out.println(nome);   // Bianchi, poi Rossi
```

← *Torna all'Esercizio 3, pagina 15*