



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Tipi di dati generici (o parametrici) e collezioni



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

Tipi generici: motivazioni - I



La specifica di tipi precisi nei metodi può portare a moltiplicare il codice (overloading di metodi quasi uguali) per trattare tutti i casi. Es: **per ogni tipo di dato deve definire un metodo printArray(). E' ridondante.**

```
public class SenzaGenerici {  
    public static void printArray(Integer[] ar) {  
        for (Integer element : ar) {System.out.print(element + ", "); }  
        System.out.println();  
    }  
    public static void printArray(Double[] ar) {  
        for (Double element : ar) {System.out.print(element + ", "); }  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        Integer[] elenco1 = {1,2,3,4,5,6};  
        Double[] elenco2 = {1.1, 2.2, 3.3, 4.4};  
        printArray(elenco1);  
        printArray(elenco2);  
    }  
}
```

Tipi generici: motivazioni - II



I tipi generici sono stati introdotti per scrivere codice generico, applicabile a più tipi di dati (*reuse* di codice): **Il tipo E fa match con qualunque tipo non primitivo** → basta un metodo per trattare ogni tipo di oggetto

```
public static <E> void printArray(E[] ar) {  
    for (E element : ar)  
        System.out.print(element + ", ");  
    System.out.println();  
}  
  
public static void main(String[] args) {  
    Integer[] elenco1 = {1,2,3,4,5,6};  
    Double[] elenco2 = {1.1, 2.2, 3.3, 4.4};  
    String[] elenco3 = {"aa", "bb", "cc"};  
  
    printArray(elenco1);  
    printArray(elenco2);  
    printArray(elenco3);  
}
```

Tipi generici: motivazioni - III



Io potrei usare Object come tipo dei metodi. Tuttavia in quel caso dovrei fare esplicitamente il cast al tipo di oggetto atteso nell'invocazione del metodo:

```
public class SenzaGenericiObject {  
    public static void printArray(Object[] ar) {  
        for (Object element : ar)  
            System.out.print(element + ", ");  
        System.out.println();  
    }  
    public static Object getElement(Object[] ar, int index) {  
        Object ris = ar[index]; return ris; }  
    public static void main(String[] args) {  
        Integer[] elenco1 = {1,2,3,4,5,6};  
        Integer intero = (Integer)getElement(elenco1, 0); // senza il  
        // downcast non compila perché non si può assegnare un  
        // Object a una variabile di tipo Integer  
    }  
}
```

Tipi generici con Object – NB:



```
public static void main(String[] args) {  
    Integer[] elenco1 = {1,2,3,4,5,6};  
    Double[] elenco2 = {1.1, 2.2, 3.3, 4.4};  
    String[] elenco3 = {"aa", "bb", "cc"};  
    printArray(elenco1);  
    printArray(elenco2);  
    printArray(elenco3);  
    Integer intero = (Integer)getElement(elenco1, 0);  
    // intero = getElement(elenco1, 0); // errore, manca il downcast  
    System.out.println(intero);  
    System.out.println(getElement(elenco3, 0));  
}
```

```
Object[] elencoMisto = {1, 2.2, "dd"};  
printArray(elencoMisto); // senza  
// controlli posso inserire elementi  
// eterogenei → fondamentale controllare  
// il tipo dei dati
```

```
C:\WINDOWS\system32\cmd.exe  
1, 2, 3, 4, 5, 6,  
1.1, 2.2, 3.3, 4.4,  
aa, bb, cc,  
1  
aa  
1, 2.2, dd,  
Premere un tasto per continuare . . .
```

Tipi generici: motivazioni - IV



Con i generici il compilatore **inferisce il tipo degli oggetti in fase di type checking statico**. Il compilatore verifica la compatibilità tra il tipo attuale e il tipo generico E, e sostituisce il tipo attuale a **E, inserendo i cast (questa operazione si chiama erasure)**:

```
public class ConMetodiGenerici {  
    public static <E> void printArray(E[] ar) {  
        for (E element : ar)  
            System.out.print(element + ", ");  
        System.out.println(); }  
  
    public static <E> E getElement(E[] ar, int index) {  
        E ris = ar[index]; return ris; }  
  
    public static void main(String[] args) {  
        Integer[] elenco1 = {1,2,3,4,5,6};  
        Integer intero = getElement(elenco1, 0);  
        System.out.println(intero);  
        System.out.println(getElement(elenco3, 0)); }  
}
```

Il compilatore sa che verrà restituito un Integer e inserisce il cast nel bytecode
→ non devo scriverlo io

Tipi generici (o tipi parametrici) - I



I tipi di dati *generici* (o *parametrici*) sono classi e metodi che hanno un *parametro di tipo*.

Consideriamo una classe Point. Senza tipi generici si può solo definire:

```
ArrayList miaLista = new ArrayList();
```

```
miaLista.add(new Point(...));
```

```
Point e = (Point)miaLista.get(0);
```

Gli oggetti ArrayList possono contenere oggetti di qualunque tipo. Quindi, quando si estrae un oggetto da miaLista, è necessario fare il downcast. Tuttavia, di fronte a errori di programmazione non si può essere certi del tipo dell'oggetto che si estrae (il programmatore potrebbe inserire oggetti di tipo diverso).

Tipi generici (o tipi parametrici) - II



Invece, utilizzando i tipi generici si può scrivere:

```
ArrayList<Point> miaLista = new ArrayList<Point>();  
miaLista.add(new Point(...));  
Point e = miaLista.get(0);
```

ArrayList<Point> è un **tipo parametrico**, in cui è stato specificato il tipo argomento da applicare alla classe generica.

miaLista può contenere solo oggetti **Point** (**controllo a tempo di compilazione**) → verifica statica delle add()
→ si accettano solo Point.

Non è più necessario il cast nell'ultima istruzione: il compilatore sa che l'oggetto estratto è un **Point**.

Variabili di tipo - I



Nelle definizioni di metodi e classi generici, i tipi generici vengono chiamati **variabili di tipo** in quanto rappresentano tipi "formali", da abbinare ai tipi "attuali" specificati in sede di creazione degli oggetti, o di invocazione dei metodi:

```
public static <E> E getElement(E[] ar, int index)
```

→ **E** è la variabile di tipo, associata al tipo attuale nelle invocazioni del metodo. Es, in:

```
Integer[] elenco1 = {1,2,3,4,5,6}; \\ autoboxing  
Integer intero = getElement(elenco1, 0);
```

E prende valore "Integer"

Definizione di classi generiche - I



Oltre a usare classi generiche predefinite, come `ArrayList<E>`, è possibile definire delle **nuove classi generiche**. Vediamo un esempio di definizione di una *classe generica* (o *parametrica*) che realizza una coppia di oggetti di tipo **T**.

```
public class Pair<T> {  
    public Pair() {  
        first = null;  
        second = null; }  
  
    public Pair(T first, T second) {  
        this.first = first;  
        this.second = second; }  
  
    public T getFirst() { return first; }  
    public T getSecond() { return second; }  
  
    private T first;  
    private T second;  
}
```

Definizione di classi generiche - II



Per usare la classe `Pair<T>`:

```
Pair<String> coppia = new Pair<String>("AAA","BBB");
```

// oppure, usando la “diamond notation”:

```
// Pair<String> coppia = new Pair<>("AAA","BBB");
```

```
String s = coppia.getFirst();
```

NB: Non è possibile istanziare un parametro di tipo con un tipo primitivo. Le variabili di tipo sono solo riferimenti

`Pair<double>` non è accettato

`Pair<Double>` è corretto

Definizione di classi generiche – III



Altro esempio: Pila<T> usa una LinkedList<T> per mantenere la lista di elementi, ma la gestisce come uno stack con push() e pop()

```
public class Pila<T>
{
    private List<T> list = new LinkedList<T>();
    public boolean isEmpty() {return list.isEmpty();}
    public void push(T v) {list.addFirst(v);}
    public T pop() {return list.removeFirst();}
}
```

//... altra classe: applicazione che usa Pila<T>

```
public static void main(String[] args)
{
    Pila<String> stack = new Pila<String>();
    stack.push("a");
    stack.push("bcd");
    String s1 = stack.pop();
    String s2 = stack.pop();
    System.out.println(s1);
    System.out.println(s2);
}
```

Codice generico e macchina virtuale



La macchina virtuale non ha classi con parametri di tipo: tutti gli oggetti appartengono a classi ordinarie (senza parametri).

Il compilatore traduce le classi generiche, e le istruzioni che utilizzano oggetti di queste classi, in classi e istruzioni accettate dalla macchina virtuale.

Ogni volta che si definisce un *tipo generico*, il compilatore lo trasforma con una operazione in un *tipo grezzo (raw type)*. Il nome del tipo grezzo coincide con quello del tipo generico senza i parametri di tipo. Tutte le variabili di tipo che compaiono nella definizione della classe generica sono sostituite con **Object**.

Tipo grezzo (Raw type) - I



Ad esempio il tipo grezzo di **Pair<T>** è:

```
public class Pair {  
  
    public Pair()  
        { first = null; second = null; }  
    public Pair(Object first, Object second)  
        { this.first = first; this.second = second; }  
  
    public Object getFirst()  
        { return first; }  
    public Object getSecond()  
        { return second; }  
  
    private Object first;  
    private Object second;  
}
```

Tipo grezzo (Raw type) - II



Tutte le classi parametriche diventano a runtime delle classi senza parametri. Infatti gli oggetti appartengono ad una sola classe. Es., consideriamo **Pair<T1, T2>** in cui usiamo 2 tipi parametrici per permettere di gestire coppie di elementi eterogenei:

```
public class PairApp {  
    public static void main(String[] args) {  
  
        Pair<String, Integer> p1= new Pair<>("Mario Rossi", 30);  
        Pair<Integer, Integer> p2= new Pair<>(25, 48);  
  
        System.out.println("Le due coppie sono istanze della stessa classe? " +  
            (p1.getClass() == p2.getClass()));           // stampa true  
  
        System.out.println("A quale classe appartengono? " +  
            p1.getClass() + "; " + p2.getClass());       // stampa Pair; Pair  
    }  
}
```

Tipo grezzo (Raw type) - III



Tuttavia questa è solo una questione implementativa. Dal punto di vista del programmatore è possibile usare le classi generiche.

È compito del compilatore tradurle in classi senza parametri, facendo tutti i controlli necessari per assicurarsi che non si verifichino errori di tipo.

È però da notare che la scelta implementativa ha degli effetti sulle operazioni consentite sulle classi generiche, come vedremo più avanti.

*Per il momento, per es., si noti che una classe parametrica con parametro di tipo **T** non può utilizzare **T** nella dichiarazione di variabili statiche, o all'interno di un metodo statico o di codice di inizializzazione statico (solo gli oggetti hanno il tipo istanziato)*

Tipo grezzo (Raw type) - esempio



```
public class PairWrong {  
    public static void main(String[] args) {  
    }  
}  
class Pair<T> {  
    public static T getPrimo() { return 1; }  
    public static T getSecondo() {return 2; }  
    public String toString() { return "1, 2"; }  
}
```

Errori di compilazione:

error: non-static type variable T cannot be referenced from a static context

```
public static T getPrimo() {  
    ^
```

error: non-static type variable T cannot be referenced from a static context

```
public static T getSecondo() {  
    ^
```

2 errors

Ereditarietà tra generici



Come con le normali classi Java, **anche con i generici possiamo definire relazioni di sottoclasse**. Es., estendo `Pair<T>` per avere coppie di elementi non ordinate (`<2,3>` è uguale a `<3,2>`)

```
class NonOrderedPair<T> extends Pair<T> {  
    public NonOrderedPair(T uno, T due) {  
        super(uno, due); }  
    public boolean equals(Object o) {  
        if (o==null) return false;  
        if (this.getClass()!=o.getClass()) return false;  
        return ((this.getPrimo().equals(coppia.getPrimo()) &&  
            this.getSecondo().equals(coppia.getSecondo())) ||  
            (this.getSecondo().equals(coppia.getPrimo()) &&  
            this.getPrimo().equals(coppia.getSecondo())));  
    }  
}
```

Overloading di metodi generici



Un metodo generico può essere **overloaded** da altri metodi generici o anche da metodi non generici. In caso di overloading il compilatore seleziona il metodo più specifico che fa match con l'invocazione. Es, dati: `Integer[] ar = {1,2,3}; C.printArray(ar);` viene eseguito il secondo metodo `printArray()`.

```
public class C {  
    public static <E> void printArray(E[] ar) {  
        for (E element : ar) {  
            System.out.print(element + ", ");  
        }  
        System.out.println();  
  
    public static void printArray(Integer[] ar) { ← ←  
        for (Integer element : ar) {  
            System.out.print("Speciale: " + element + ", ");  
        }  
        System.out.println();  
    }  
}
```

Overloading vs. overriding di metodi - I

Attenzione ai metodi ereditati! Se faccio **overloading** di equals() gli oggetti

Pair NON eseguono questo metodo quando utilizzati in upcasting come

Object (es. Arrays.sort())



```
class Pair<T1, T2> {
```

```
    private T1 primo; private T2 secondo;
```

```
    public Pair(T1 uno, T2 due) { primo = uno; secondo = due; }
```

```
    public T1 getPrimo() { return primo; }
```

```
    public T2 getSecondo() { return secondo; }
```

```
    public String toString() {    // overrides toString() di Object
```

```
        return "primo: " + primo.toString()+"\n"+"secondo: "+secondo.toString(); }
```

```
    public boolean equals(Pair<T1,T2> coppia) { // overloads equals()
```

```
        // di Object
```

```
        if (coppia==null) return false;
```

```
        return (primo.equals(coppia.primo) &&
```

```
            secondo.equals(coppia.secondo));
```

```
}}
```



Overloading vs. overriding di metodi - II

```
class Pair<T1, T2> {  
    private T1 primo; private T2 secondo;  
    public Pair(T1 uno, T2 due) {  
        primo = uno; secondo = due; }  
    ...  
    public String toString() {                // overrides toString() di Object  
        return "primo: " + primo.toString() + "\n" +  
            "secondo: " + secondo.toString();  
    }  
    public boolean equals(Object o) { // OK!! overrides equals() di Object  
        if (o==null)    return false;  
        if (this.getClass() != o.getClass()) return false;  
        Pair<T1, T2> coppia = (Pair<T1, T2>)o;  
        return (primo.equals(coppia.primo) &&  
            secondo.equals(coppia.secondo));  
    }  
}
```

Vincoli sui tipi parametrici - I



A volte non ha senso sostituire un tipo di riferimento qualunque ad un tipo parametrico in una classe generica. Es. se volessi aggiungere a `Pair<T>` il metodo `getMassimo()`, che restituisce il valore più alto in una coppia, dovrei imporre che il tipo sostituito a `T` implementi `Comparable`. Per farlo aggiungo a definizione di `Pair` la **restrizione di tipo su T: dò upperbound al tipo parametrico di una classe**

```
class Pair <T extends Comparable<T>> {  
    private T primo; private T secondo;  
public T getMassimo() {  
    if (primo.compareTo(secondo) >=0) return primo;  
    else return secondo; }  
  
    // ... altri metodi di Pair  
}
```

Vincoli sui tipi parametrici - II



class Pair <T extends Comparable<T>>

⇒ Data la restrizione, il **raw type** con cui sostituire T è Comparable.

⇒ Il compilatore farà il **controllo di tipo** sui parametri attuali di costruttori e di invocazioni dei metodi, impedendo di creare Pair di oggetti che non **implementino Comparable**.

La restrizione di tipo può specificare al più una classe e n interfacce: **class C <T extends Classe0 & Interf₁ &...& Interf_n>**

NB: in questo caso:

- **il tipo grezzo (raw type) di T è Classe0**
- C può essere istanziata solo con elementi in cui T è Classe0 o un sottotipo di Classe0

Vincoli sui tipi parametrici – esempio – I



Consideriamo due classi: MioTipo e SottoTipo:

```
class MioTipo {  
    private String x;  
    public MioTipo(String x) { this.x = x; }  
    public String toString() { return x; }  
}  
  
class SottoTipo extends MioTipo {  
    public SottoTipo(String x) {  
        super(x);  
        //... altro codice }  
}
```



Vincoli sui tipi parametrici – esempio - II



Se impongo MioTipo come upper bound di T potrò creare oggetti Pair che contengono elementi MioTipo o SottoTipo, non altro:

```
class Pair <T extends MioTipo> {  
    // ... La definizione di Pair già data ...  
}  
  
public static void main(String[] args) {  
    Pair<MioTipo> coppia1 =    // OK  
        new Pair<MioTipo>(new MioTipo("Ciao "), new MioTipo("mondo"));  
  
    Pair<SottoTipo> coppia2 =  // OK  
        new Pair<SottoTipo>(new SottoTipo("Ciao "),  
                             new SottoTipo("mondo2"));  
  
    Pair<String> coppiaWrong =  
        new Pair<String>("Ciao ", "mondo"); // NO!!  
}
```

Vincoli sui tipi parametrici – esempio - III

Per garantire la robustezza del software può essere necessario stringere i tipi generici utilizzati nei metodi di una classe/interface parametrica. Esempio:

```
public class Studente implements Comparable<Studente> {  
    private int matricola;  
    private String nome;  
    private String cognome;  
  
    public int compareTo(Studente s) { // siamo certi che s sia Studente  
        // perché il compilatore richiede compareTo(Studente), non  
        // compareTo(Comparable)  
        if (this.cognome.compareTo(s.cognome)<0 ||  
            (this.cognome.equals(s.cognome) &&  
             this.nome.compareTo(s.nome)<0))  
            return -1;  
        else if (this.equals(s)) return 0;  
        else return 1;  
    }  
}
```

Collezioni - I



Java fornisce un insieme di classi che realizzano le strutture dati più utili (collezioni), come liste o insiemi.

Il package `java.util` che contiene le collezioni distingue fra *interfacce* e *implementazioni*.

Ad esempio **List<E>** è una **interface** che specifica le operazioni principali sulle liste.

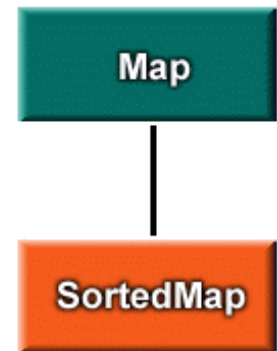
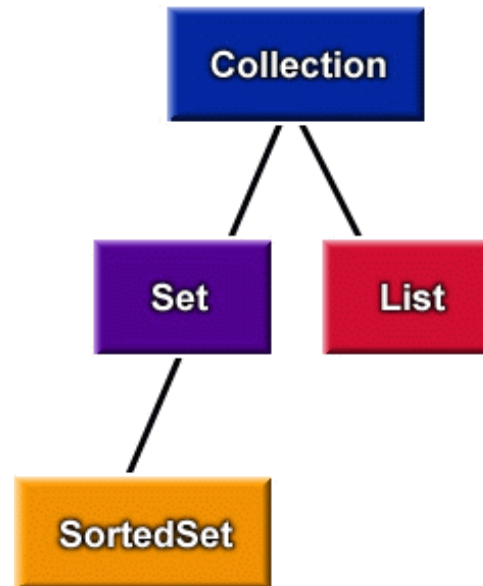
ArrayList<E> e **LinkedList<E>** sono due classi concrete che **implementano** l'interfaccia **List<E>** in modi diversi.



Strutture dati: Interfacce

Sono tutte Interface:

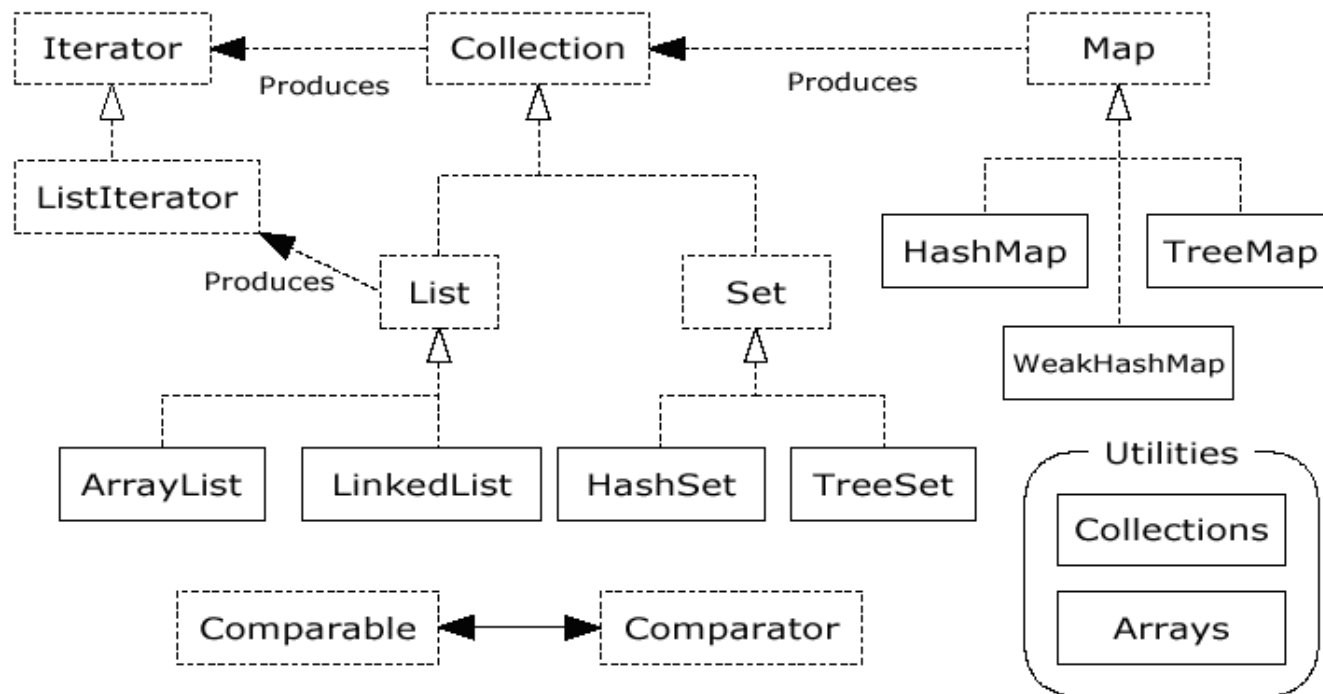
- **Collection**: un arbitrario gruppo di oggetti
- **List**: un gruppo di oggetti memorizzati in una data sequenza
- **Set**: un gruppo di oggetti senza duplicati
- **Map**: un gruppo di coppie (oggetti) chiave-valore



Strutture dati: implementazioni



Ogni interfaccia ha più implementazioni che possono essere scelte in modo indifferente a seconda delle esigenze (anche di efficienza)





L'interfaccia **Set<E>** specifica un insieme:

```
interface Set<E> {  
    boolean add(E o);  
    boolean contains(Object o);  
    boolean remove(Object o);  
    .....  
}
```

La classe **HashSet<E>** implementa l'interfaccia **Set<E>** mediante una tabella hash.

La classe **TreeSet<E>** implementa l'interfaccia **Set<E>** garantendo che gli elementi dell'insieme siano ordinati. Gli elementi dell'insieme devono implementare l'interfaccia **Comparable<T>**. Oltre ai metodi definiti in **Set<E>**, vengono forniti altri metodi come **first()** o **last()**.



Quando in un programma si utilizza un insieme, non è necessario sapere quale implementazione dell'insieme verrà utilizzata.

Conviene utilizzare la *classe concreta* solo quando si costruisce l'insieme, mentre si può utilizzare il *tipo dell'interfaccia* per specificare il riferimento alla collezione (upcasting):

```
Set<Point> s = new HashSet<Point>;  
Point e = ...;  
s.add(e);  
...
```

In questo modo, se si decide di usare l'implementazione **TreeSet**, è sufficiente modificare la prima istruzione in cui si crea l'insieme, senza modificare il resto del programma.

Iteratori - I



L'interfaccia **Collection** prevede vari metodi:

```
public interface Collection<E> {  
    boolean add(E element);  
    Iterator<E> iterator();  
    ...  
}
```

Il metodo `iterator` restituisce un oggetto che implementa l'interface `Iterator`:

```
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
    void remove();  
}
```


Iteratori - I



La chiamata ripetuta del metodo `next()` permette di scorrere gli elementi della collezione uno alla volta.

```
void printCollection(Collection<Object> c) {  
    Iterator<Object> iter = c.iterator();  
    while (iter.hasNext()) {  
        Object ob = iter.next();  
        System.out.println(ob);  
    }  
}
```

Per iterare sugli elementi della collezione si può usare anche una istruzione di tipo **for each**:

```
void printCollection(Collection<Object> c) {  
    for(Object ob : c)  
        System.out.println(ob);  
}
```

L'istruzione può essere usata con qualunque collezione che implementi l'interfaccia ***Iterable***.



Auto-boxing

Le strutture come **Collection** possono contenere solo oggetti. Quindi per inserire un tipo primitivo (es. `int`) occorre convertirlo (**boxing**) nel corrispondente oggetto (es. `Integer`). Viceversa, quando si estrae un oggetto dalla collezione occorre riconvertirlo nel tipo primitivo (**unboxing**). Il compilatore fa automaticamente boxing e unboxing.

```
ArrayList<Integer> al = new ArrayList<Integer>();
```

```
al.add(0, new Integer(25));  
int x = (al.get(0)).intValue();
```

```
// oppure, grazie all'autoboxing:  
al.add(0, 25);  
int x = al.get(0);
```

Array e collezioni



È possibile specificare il tipo degli elementi di una collezione, es. `ArrayList<Point>`, esattamente come si fa con gli array: `Point[]`.

Tuttavia ci sono differenze:

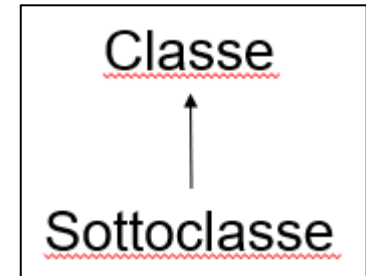
1. non è possibile istanziare il parametro di tipo di una collezione con un tipo primitivo (i tipi parametrici sono riferimenti);
2. **gli oggetti array mantengono a runtime l'informazione sul tipo degli elementi, mentre gli oggetti collezione perdono l'informazione a runtime.**

Vediamo le conseguenze del secondo punto.



Con gli array sono ammissibili:

1. `Classe[] arrayClasse;`
2. `Sottoclasse[] arraySottoclasse;`
3. `arraySottoclasse = new Sottoclasse[2];`
4. `arrayClasse = arraySottoclasse;`
5. `arrayClasse[0] = new Classe(...); // compila correttamente`



Il compilatore accetta queste istruzioni, in particolare l'assegnazione nella linea 4 perché un array di **Sottoclasse** è una sottoclasse di array di **Classe**.

L'interprete dà errore a *runtime* eseguendo la linea 5, perché **arrayClasse** fa riferimento ad un array di oggetti di **Sottoclasse**, creato alla linea 3, a cui non è possibile assegnare oggetti **Classe**.

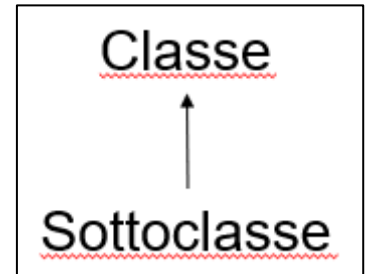
L'interprete può rilevare l'errore perché, quando viene creato un array, gli viene associata l'informazione sul tipo degli elementi.

Generici e sottotipi - II



Vediamo lo stesso programma con ArrayList al posto degli array:

1. ArrayList<Classe> alClasse;
2. ArrayList<Sottoclasse> alSottoclasse;
3. alSottoclasse = new ArrayList<Sottoclasse>();
4. **alClasse = alSottoclasse;**
5. alClasse.add(new Classe(...));



In questo caso **il compilatore dà errore** alla linea 4, dicendo che i tipi **ArrayList<Classe>** e **ArrayList<Sottoclasse>** sono incompatibili.

Se il compilatore accettasse questo programma l'interprete non sarebbe in grado di scoprire a runtime l'errore alla linea 5. Questo perché le collezioni non conservano a runtime l'informazione sul tipo degli elementi.

Generici e sottotipi - III



Data una classe parametrica **C<T>**, e due tipi **Tipo1** e **Tipo2**, tali che **Tipo2** è sottotipo di **Tipo1**, non esiste nessuna relazione tra i tipi **C<Tipo1>** e **C<Tipo2>**.

Es., anche se **SottoTipo** extends **MioTipo**, non è possibile assegnare un **Pair<SottoTipo>** a un **Pair<MioTipo>**, né viceversa (mentre sarebbe possibile assegnare un **SottoTipo[]** a un **MioTipo[]**).

```
public static void main(String[] args) {
```

```
    Pair<MioTipo> coppia3;
```

```
    Pair<SottoTipo> coppia4;
```

```
    coppia3 = coppia4; // error: incompatible types:
```

```
        // Pair<SottoTipo> cannot be converted to Pair<MioTipo>
```

```
    coppia4 = coppia3; // error: incompatible types:
```

```
        // Pair<MioTipo> cannot be converted to Pair<SottoTipo>
```

```
}
```

MioTipo
↑
SottoTipo

Generici e sottotipi - III



Il comportamento descritto prima è molto restrittivo. Ad esempio, il seguente metodo per stampare gli elementi di una collezione:

```
void printCollection(Collection<Object> c) {  
    for(Object ob : c)  
        System.out.println(ob);  
}
```

può essere usato solo per stampare gli elementi di una **Collection<Object>**. Il metodo non può avere come parametro, ad esempio, una **Collection<Integer>**, perchè i due tipi sono incompatibili.



Il tipo jolly (wildcard) - I

Per definire una *Collection* di qualunque cosa si può usare la notazione **Collection<?>**, dove ? è la wildcard (jolly).

```
void printCollection(Collection<?> c) {  
    for(Object ob : c)  
        System.out.println(ob);  
}
```

È possibile chiamare **printCollection()** con un parametro **Collection<Integer>**, **Collection<Point>** o collezione di qualsiasi altro tipo: **l'upper bound di default di ? è Object.**



Il tipo jolly (wildcard) - II

Anche con il tipo jolly ci sono delle limitazioni.
Ad esempio in

```
Collection<?> c = new ArrayList<String>();  
c.add(new Object());
```

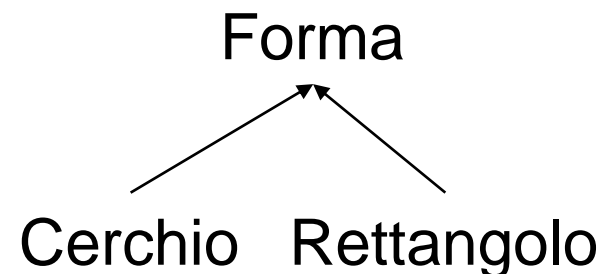
il compilatore accetta la prima istruzione, ma non la seconda. Questo perché non può garantire la correttezza del tipo dell'elemento che viene inserito nella collezione (di cui non conosce il tipo).

Limitazioni sul tipo jolly - I



Consideriamo la classe astratta:

```
public abstract class Forma {  
    public abstract void disegna();  
}
```



e varie sue sottoclassi come **Cerchio**, **Rettangolo**, ... che implementano il metodo **disegna**. Un metodo per disegnare le forme contenute in una lista potrebbe essere:

```
public void disegnaTutto(List<Forma> forme) {  
    for (Forma f: forme)  
        f.disegna();  
}
```

Limitazioni sul tipo jolly - II



La definizione

```
public void disegnaTutto(List<Forma> forme) {
```

è restrittiva perché il metodo **disegnaTutto** può essere applicato solo a una **List<Forma>** e non, per esempio, a una **List<Rettangolo>**.

D'altra parte

```
public void disegnaTutto(List<?> forme) {
```

sarebbe troppo generale perché il **?** può essere legato a qualunque tipo, in particolare ad **Object** che non ha il metodo **disegna**.

Limitazioni sul tipo jolly - III



Per risolvere questo problema, il ? può essere limitato con la notazione

```
public void disegnaTutto(List<? extends Forma> forme) {  
    for (Forma f: forme)  
        f.disegna();  
}
```

in cui il tipo degli elementi della lista può essere una qualunque sottoclasse di **Forma**. Quindi certamente avrà il metodo `disegna`.

In questo caso il raw type che viene assegnato a `List` è `Forma` (il più generico tipo che gli elementi della collezione possono avere – **Forma è l'upper bound della wildcard**).

Uso scorretto delle wildcards



Voi non potete usare le wildcard nelle dichiarazioni delle variabili: il tipo dei dati deve essere definito in modo preciso.

```
public class WildCardTest {  
  
    public static void main(String[] args) {  
        ArrayList<?> lista1;  
        //lista1.add("Ciao"); // error: no suitable method found for add(String)  
        //lista1.add(new Object());      // idem, per Object  
  
        ArrayList<? extends Number> lista2;  
        //lista2.add(new Integer(2));  
        // error: no suitable method found for add(Integer)  
        Number num = 54;  
        // lista2.add(num); //error: no suitable method found for add(Number)  
    }  
}
```

Esempi di uso corretto delle wildcards



```
public class WildCardTest {
```

```
    public static void main(String[] args) {
```

```
        ArrayList<Integer> numeri = new ArrayList<Integer>();
```

```
        numeri.add(54); numeri.add(23); // OK autoboxing
```

```
        System.out.println(sum(numeri)); // OK
```

```
        ArrayList<Number> numeri2 = new ArrayList<Number>();
```

```
        Number num = 54;
```

```
        numeri2.add(num); numeri2.add(num); // con upcasting OK!
```

```
        System.out.println(sum(numeri2));
```

```
    }
```

```
    public static double sum(ArrayList< ? extends Number > lista) {
```

```
        double tot = 0;
```

```
        for (Number el : lista) {
```

```
            tot = tot+el.doubleValue(); }
```

```
        return tot;
```

```
}}
```

Upcasting e tipi generici



```
public class PairApp {  
  
    public static void main(String[] args) {  
        Pair<MioTipo> coppia5 =  
            new Pair<>(new SottoTipo("Prova "),  
                    new SottoTipo("upcasting"));  
        Pair<MioTipo> coppia1 = coppia5;  
    }  
}
```

Usando upcasting non ci sono problemi di compilazione né in fase di esecuzione perché sia coppia1 che coppia5 sono viste come variabili di tipo Pair<MioTipo>