# Metodi Generici

# Metodi Generici

- Metodi che dipendono da una variabile di tipo
- Possono essere definiti all'interno di qualunque classe, generica o meno

```
/* trasforma un array in una lista, copiando
 * tutti gli elementi di a in l
 */
static void array2List(Object[] a, List<?> l){ . . . }
```

- N.B. Evitiamo `List<Object>` perché renderebbe il metodo non utilizzabie su liste arbitrarie

*Continua*

# Metodi Generici

- **Al solito però . . .**

```
/* trasforma un array in una lista, copiando
 * tutti gli elementi di a in l
 */
static void array2List(Object[] a, List<?> l)
{
  for (Object o : a) l.add(o) // compiler error
}
```

- **. . . non possiamo aggiungere elementi ad una struttura (o modificare) con elementi di tipo wildcard**

*Continua*

# Metodi Generici

- **Soluzione: rendiamo il metodo parametrico**

```
/* trasforma un array in una lista, copiando
 * tutti gli elementi di a in l
 */
static <T> void array2List(T[] a, List<T> l)
{
  for (T o : a) l.add(o)
}
```

- **possiamo invocare questo metodo con una qualunque lista il cui tipo sia supertipo del tipo base dell'array**
  - purché sia un tipo riferimento

# Invocazione di metodi generici

- **Nell'invocazione di un metodo generico non è necessario  passare l'argomento di tipo**
- **il compilatore inferisce il tipo, se esiste, dai tipi degli argomenti del metodo**

# Invocazione di metodi generici

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co); // T = Object (inferito)
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs); // T = String (inferito)
fromArrayToCollection(sa, co); // T = Object (inferito)
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn); // T = Number (inferito)
fromArrayToCollection(fa, cn); // T = Number (inferito)
fromArrayToCollection(na, cn); // T = Number (inferito)
fromArrayToCollection(na, co); // T = Object (inferito)
fromArrayToCollection(na, cs); // compiler error
```

*Continua*

# Wildarcds vs variabili di tipo

- Ci sono situazioni in cui è possibili usare equivalentemente wildcards e variabili di tipo.
- Nella libreria `Collection` troviamo

```
interface Collection<E>
{
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
    . . .
}
```

*Continua*

# Wildarcds vs variabili di tipo

- **Queste specifiche possono essere espresse equivalentemente con metodi parametrici**

```
interface Collection<E>
{
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
    . . .
}
```

- **Il secondo metodo è parametrico in qualunque sottotipo di E**
  - i bounds si possono utilizzare anche con variabili, non solo con wildcards

*Continua*

# Wildarcds vs variabili di tipo

- **Wildcards e variabili di tipo possono coesistere**

```
interface Collection<E>
{
  public static <T>
        void copy(List<T> dest, List<? extends T> src)
  . . .
}
```

- **Notiamo la dipendenza tra i tipi dei due parametri:**
  - il tipo della sorgente deve essere un sottotipo del tipo della destinazione

***Continua***

# Wildarcds vs variabili di tipo

- **Potremmo analogamente riformulare in modo da evitare le wildcards**

```
interface Collection<E>
{
  public static <T, S extends T>
        void copy(<List<T> dest, List<S> src)
  . . .
}
```

- **Come scegliere tra le due soluzioni?**

*Continua*

# Wildarcds vs variabili di tipo

- **In generale, preferiamo le wildcards quando entrambe le soluzioni sono possibili**
- **Possiamo darci la seguente *"rule of thumb"***
  - se una variabile di tipo ha una unica occorrenza nella specifica di un metodo
  - e il tipo non è il target di un operazione di modifica
  - utilizziamo una wildcard al posto della variabile

# Variabili di Tipo e Bounds

- Abbiamo visto che possiamo definire bounds anche per variabili di tipo (non solo wildcards)
- Un caso paradigmatico

```
public static
   <T extends Comparable<T>> max(Collection<T> coll)
{
  T candidate = coll.iterator().next();
  for (T e : coll)
     if candidate.compareTo(e) < 0) candidate = e;
  return candidate;
}
```

# Variabili di Tipo e Bounds

- **Il bound su una variabile impone vincoli sulla variabile, determinando quali metodi possono essere utilizzati su valori del tipo variabile**

```
public static
   <T extends Comparable<T>> T max(List <T> coll)
```

- **Qui il bound è ricorsivo:**
  - informa che i valori con cui operiamo forniscono un metodo **`compareTo()`**
  - che gli argomenti del metodo devono essere dello stesso tipo dei valori

# Generics e *"erasure"*

- **I tipi generici sono significativi a compile-time**
- **La JVM opera invece con tipi *"raw"***
- **Il tipo raw è ottenuto da un tipo generico mediante un processo detto *erasure* che rimuove le variabili di tipo**
  - il bycode generato da un tipo generico è lo stesso che viene generato dal corrispondente tipo raw.

# Generics e *"erasure"*

- **Generano lo stesso bytecode**

```
List<String> words = new ArrayList<String>();
words.add("hi");
words.add("there");
String welcome = words.get(0) + words.get(1);
```

```
List words = new ArrayList();
words.add("hi");
words.add("there");
String welcome =
        (String)words.get(0) + (String)words.get(1);
```

# Generics e *“erasure”*

- ***Cast-iron guarantee***
  - i cast che vengono aggiunti dalla compilazione di codice generico non falliscono mai.

# Generics e Arrays

- **In generale:**

```
A ≤ B NON implica C<A> ≤ C<B>
```

- **MA:**

```
A ≤ B implica A[] ≤ B[]
```

- **Quali conseguenze?**

# Generics e Arrays

- **Conseguenze / 1: ArrayStoreException**

```
Integer[] ai = new Integer[10]
Number[] an = ai;          // type OK
an[0] = 3.14;              // ArrayStoreException
```

- **Meglio così che continuare**

```
Integer i = ai[0];         // uh oh ...
```

# Generics e Arrays

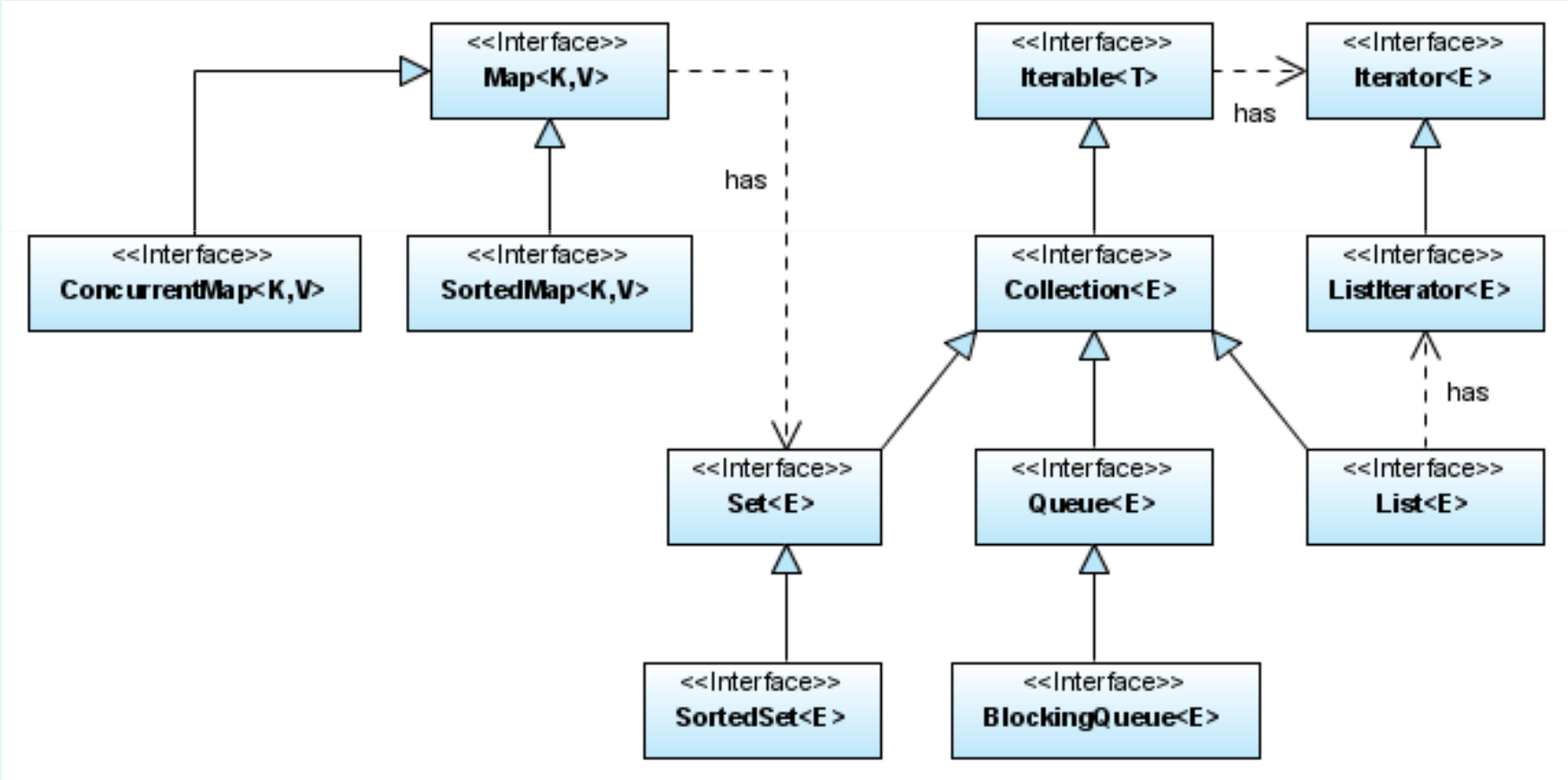
- **Conseguenze / 2: no new per array generici**

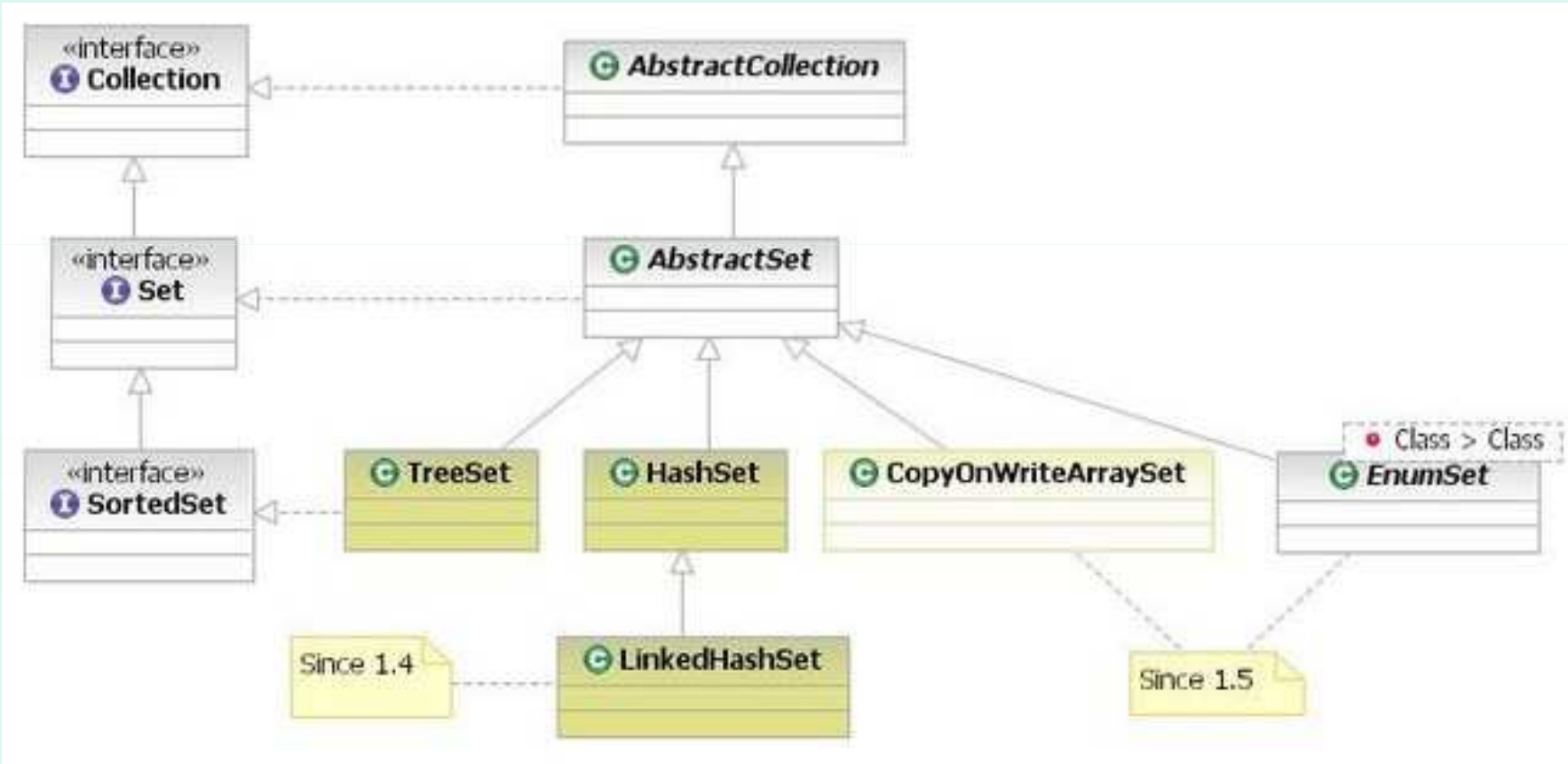
```
class MyClass<T>
{
   T[] contents = new T[100]; // Non compila

   // ecco perchè

   public void showTheProblem()
   {
       Object[] objs = contents;
       objs[0] = new String(); // no ArrayStoreException
       T bump = contents[0];   // ClassSclassException
                               // per T != String
   }
}
```

# Collections


<<Interface>>
Map<K,V>
<<Interface>>
Iterable<T>
<<Interface>>
Iterator<E>
has
has
<<Interface>>
ConcurrentMap<K,V>
<<Interface>>
SortedMap<K,V>
<<Interface>>
Collection<E>
<<Interface>>
ListIterator<E>
has
<<Interface>>
Set<E>
<<Interface>>
Queue<E>
<<Interface>>
List<E>
<<Interface>>
SortedSet<E>
<<Interface>>
BlockingQueue<E>


«interface»
Collection
AbstractCollection
«interface»
Set
AbstractSet
«interface»
SortedSet
TreeSet
HashSet
CopyOnWriteArraySet
Class > Class
EnumSet
Since 1.4
LinkedHashSet
Since 1.5