UNIVERSITÀ PARTHENOPE
DiST
Dipartimento di Scienze e Tecnologie

Programmazione 3
e
Laboratorio di Programmazione 3

Tipi Generici

Angelo Ciaramella

Array

- array
  - sequenza di lunghezza prefissata di valori dello stesso tipo
    - tipo primitivo o una classe

```
//dichiarazioni di array

char alfabeto []; // oppure
char [] alfabeto;

Button bottoni []; //oppure (Button in java.awt)
Button [] bottoni;
```


Creazione e inizializzazione di Array

```
// creazione
alfabeto = new char[21];
bottoni = new Button[3];

// Inizializzazione
alfabeto [0] = 'a';
alfabeto [1] = 'b';
alfabeto [2] = 'c';
alfabeto [3] = 'd';
. . . . . .
alfabeto [20] = 'z';
bottoni [0] = new Button();
bottoni [1] = new Button();
bottoni [2] = new Button();
```


Array 2D

```
// un array bidimensionale non deve per forza essere
// rettangolare

int arrayNonRettangolare [][] = new int[4][];
arrayNonRettangolare [0] = new int[2];
arrayNonRettangolare [1] = new int[4];
arrayNonRettangolare [2] = new int[6];
arrayNonRettangolare [3] = new int[8];
arrayNonRettangolare [0][0] = 1;
arrayNonRettangolare [0][1] = 2;
arrayNonRettangolare [1][0] = 1;
. . . . . .
arrayNonRettangolare [3][7] = 10;
```

Gli array sono utilizzati relativamente poco in Java, sono preferiti gli oggetti della libreria Collections

Creazione e inizializzazione di Array

```
//creazione e inizializzazione
char alfabeto [] = {'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
'u', 'v', 'z'};
Button bottoni [] = {new Button(), new Button(), new
Button()} ;


// lunghezza array

alfabeto.length
```


Array 2D

```
// un array bidimensionale non deve per forza essere
// rettangolare

int arrayNonRettangolare [][] = new int[4][];
arrayNonRettangolare [0] = new int[2];
arrayNonRettangolare [1] = new int[4];
arrayNonRettangolare [2] = new int[6];
arrayNonRettangolare [3] = new int[8];
arrayNonRettangolare [0][0] = 1;
arrayNonRettangolare [0][1] = 2;
arrayNonRettangolare [1][0] = 1;
. . . . . .
arrayNonRettangolare [3][7] = 10;
```

Gli array sono utilizzati relativamente poco in Java, sono preferiti gli oggetti della libreria Collections

Vettori

- Un vettore (`ArrayList`) rappresenta un insieme di oggetti disposti in sequenza
  - si può accedere indipendentemente a ciascun elemento della sequenza

```
import java.util.ArrayList;

ArrayList coins = new ArrayList();
coins.add(new Coin(0.1, "dime"));
coins.add(new Coin(0.25, "quarter"));
```

- Per accedere agli oggetti nel vettore usiamo il metodo `get` specificando la posizione nel vettore (`coins.get(4)`)

ArrayList

- Un oggetto di tipo `ArrayList` memorizza riferimenti a `Object`

```
Coin aCoin = (Coin)coins.get(0);
```


ArrayList

- Il metodo `size` restituisce il numero di elementi di un vettore

- Per impostare un elemento ad un nuovo valore
  - metodo `set`

- Per aggiungere un nuovo oggetto al termine del vettore
  - metodo `add`

Esercizio

- Creare una classe `Purse` (borsellino contenente monete) contenente i seguenti metodi
  - `add()`
  - `getTotal()`
  - `count()`
  - `find(`*`Coin`*`)`
  - `count (`*`Coin`*`)`
  - `getMaximum()`

Codice di riferimento

`Purse.java - Creare PurseTest.java`

Tipo generico

- Quando dichiariamo una classe o un'interfaccia possiamo "renderla generica" (generic type)
  - aggiungiamo alla definizione uno o più tipi parametro

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


Classe contenitore

```
public class Contenitore {
private Object object;
public void setObject(Object object) {
this.object = object;
}
public Object getObject() {
return object;
}
}

// Esempio di utilizzo
Contenitore contenitore = new Contenitore();
contenitore.setObject("Stringa");

// contenitore.setObject(new Integer(1));

String object = (String)contenitore.getObject();
System.out.println(object);
```

Se avessimo impostato un oggetto `Integer` nel contenitore avremmo ottenuto una `ClassCastException`

Classe contenitore

```
public class ContenitoreGenerics<T> {
private T object;
public void setObject(T object) {
this.object = object;
}
public T getObject() {
return object;
}
}

// Esempio di utilizzo
ContenitoreGenerics<String> contenitore = new
ContenitoreGenerics<String>();
contenitore.setObject("Stringa");
String object = contenitore.getObject();
System.out.println(object);
```

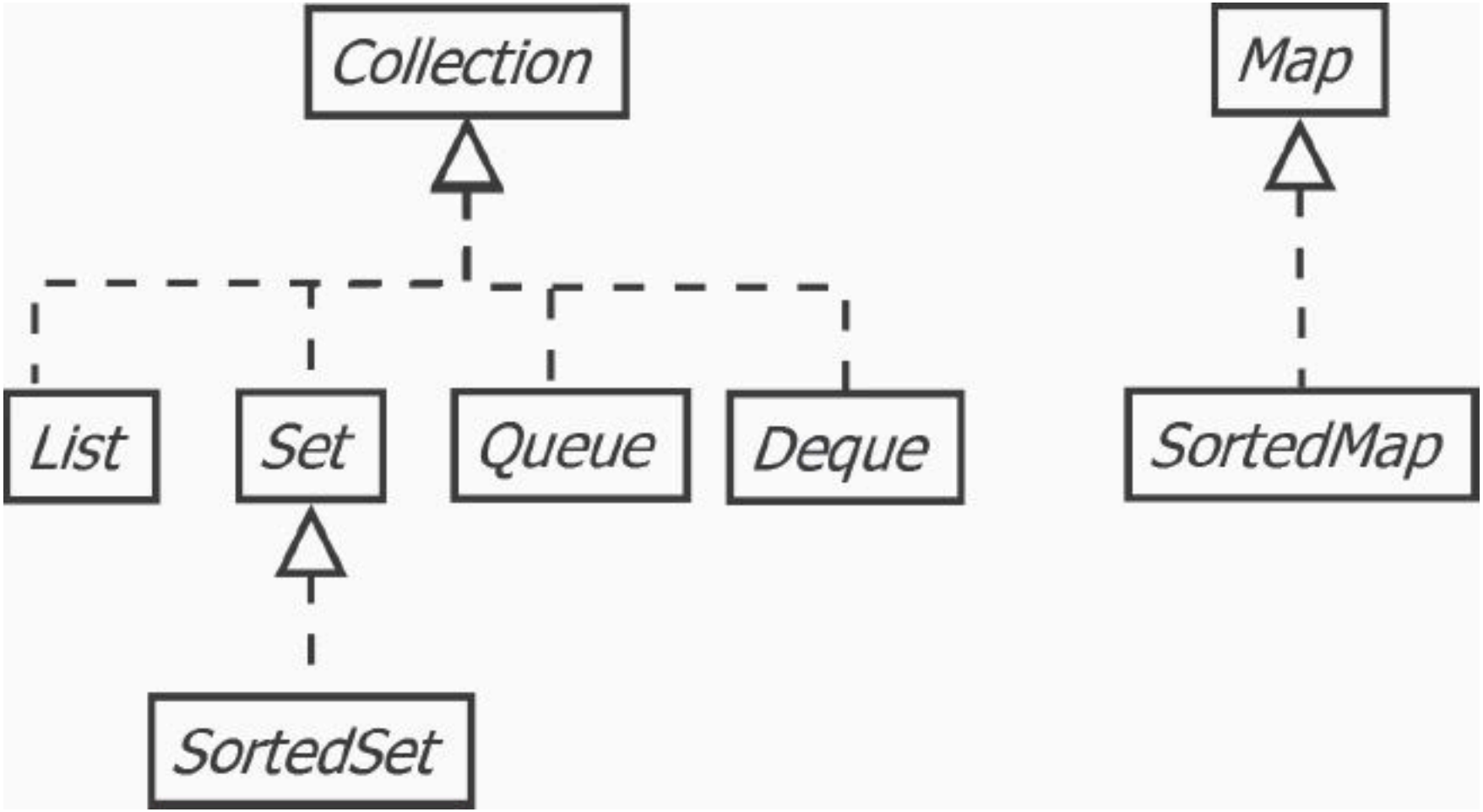

Tipo generico

- Per convenzione quando si dichiara un tipo parametro si usa un identificatore costituito da una sola lettera maiuscola
  - la libreria standard usa spesso
    - E per "Element"
    - K per "Key"
    - N per "Number"
    - T per "Type"
    - V per "Value"
    - S, U, V per il secondo, terzo e quarto tipo

Framework Collections

- collezione
  - oggetto che raggruppa più elementi in una singola unità
- framework
  - insieme di classi e di interfacce riutilizzabili ed estendibili

Framework Collections



La gerarchie tra le interfacce fondamentali del framework Collections

Generics e collection

- L'interfaccia `List` estende `Collection` dichiarata anch'essa generica

```
public interface List<E> extends Collection<E>

public boolean add(E o) //metodo add di List
```

- La sottoclasse `ArrayList` implementa `List`

```
List<Auto> lista = new ArrayList<Auto>();
```

- Anche se una classe è definita generica, l'utilizzo dei tipi parametri non è obbligatorio

Classi wrapper

- Classi wrapper
  - fanno da contenitore ad un tipo di dato primitivo
    - `Byte`
    - `Short`
    - `Integer`
    - `Long`
    - `Float`
    - `Double`
    - `Boolean` (non è una sottoclasse di `Number`)
    - `Character`
  - Non hanno metodi del tipo "set"
    - Un oggetto istanziato non può cambiare il suo stato
  - Un metodo molto utile è quello per il parsing

```
int i = Integer.parseInt("1");
```


Classi wrapper

- In Java 8
  - Arricchiti con metodi statici `sum()`, `max()` e `in()`
    - `Byte`
    - `Short`
    - `Integer`
    - `Long`
    - `Float`
    - `Double`
    - `Character`
  - Anche la classe `Boolean` è stata arricchita
    - `logicalAnd()`, `logicalOr()` e `logicalXor()`

Autoboxing-Autounboxing

- Le collezioni accettano come elementi solo oggetti e non tipi di dati primitivi
  - grazie all'autoboxing-autounboxing è possibile aggiungere tipi primitivi direttamente alle collezioni

```
List list = new ArrayList();
list.add(5.3F);
```

autoboxing

equivalenti

```
List list = new ArrayList();
list.add(new Float(5.3F));
```


Classe contenitore

```
ArrayList list = new ArrayList();
list.add(1);
list.add(false);
list.add('c');
int intero = (Integer) list.elementAt(0);
boolean booleano = (Boolean) list.elementAt(1);
character c = (Character) list.elementAt(2);
```

Esempi di recupero dati

Generics and collections

```
List<String> strings = new ArrayList<String>();

//oppure

List<String> strings = new ArrayList<>();
```

Lista che accetta solo stringhe e nessun altro tipo di oggetto

```
List<String> strings = new ArrayList<String>();
List<Integer> ints = new ArrayList<Integer>();
List<Date> dates = new ArrayList<Date>();
```

Liste con differenti parametri

```
List<int> ints = new ArrayList<int>();
```

Non è ammesso per tipi primitivi – errore di compilazione

```
List<Integer> ints = new ArrayList<Integer>();
```

E' possibile aggiungere anche tipi primitivi

Iterator

- Iterator
  - interfaccia del framework Collections che permette di iterare su una collezione

```
List<String> strings = new ArrayList<String>();
strings.add("Autoboxing & Auto-Unboxing");
strings.add("Generics");
strings.add("Static imports");
strings.add("Enhanced for loop");
//. . .
Iterator<String> i = strings.iterator();
while (i.hasNext()) {
String string = i.next();
System.out.println(string);
}
```

Deve essere generico

wildcard

```
public void print(ArrayList<Object> al) {
Iterator<Object> i = al.iterator();
while (i.hasNext()) {
Object o = i.next();
System.out.println(o);
}
}
```

metodo che accetta in input solo tipi generici con parametro Object

```
public void print(ArrayList<?> al) {
Iterator<?> i = al.iterator();
while (i.hasNext()) {
Object o = i.next();
System.out.println(o);
}
}
```

wildcard (caratteri jolly) accetta qualsiasi tipo

Propri tipi generici

```
public class OwnGeneric<E> {
private List<E> list;
public OwnGeneric () {
list = new ArrayList<E>();
}
public void add(E e) {
list.add(e);
}
public void remove(int i) {
list.remove(i);
}
public E get(int i) {
return list.get(i);
}
public int size() {
return list.size();
}
public boolean isEmpty() {
return list.size() == 0;
}
```


Esempio di creazione tipi generici

Propri tipi generici

```
public String toString() {
StringBuilder sb = new StringBuilder();
int size = size();
for (int i = 0; i < size; i++) {
sb.append(get(i) + (i != size - 1 ? "-" : ""));
}
return sb.toString();
}
}
```

Esempio di creazione di tipi generici

```
public class OwnGeneric <E extends Number> {...

public void print(List <? extends Number> list) {
for (Iterator<? extends Number> i = list.iterator();
i.hasNext( ); ) {
System.out.println(i.next());
}
}
```

upper bounded wildcard (solo sottoclassi di Number)

Propri tipi generici

```
public static void riempiLista(List<? super Integer>
list) {
int size = list.size();
for (int i = 1; i <= size; i++) {
list.add(i);
}
System.out.println(list);
}
```

lower bounded wildcard - permette di riempire e stampare una lista riempita di `Integer`, `Number` o `Object`

Metodi generici

- Metodi generici
  - metodi che definiscono i propri tipi parametro

```
public class GenericMethod {
public static <N extends Number> String getValue(N
number) {
String value = number.toString();
return value;
}
public static void main(String args[]) {
String value = GenericMethod.getValue(new
Integer(25));
System.out.println(value);
}
}
```


Type inference

- Type inference for generic instance creation

```
ArrayList<String> arrayList;

// Prima del Java 7
ArrayList<String> arrayList = new ArrayList<String>();

// Dopo Java 7
ArrayList<String> arrayList = new ArrayList<>();
```


Parametri covarianti

- tipi di ritorno covarianti
  - possibilità di creare override di metodi il cui tipo di restituzione è una sottoclasse del tipo di restituzione del metodo originale

```
interface Cibo {
String getColore();
}
interface Animale {
void mangia(Cibo cibo);
}
```

```
public class Erba implements Cibo {
public String getColore() {
return "verde";
}
}
```


Parametri covarianti

```
public class Carnivoro implements Animale {
public void mangia(Cibo cibo) {
//un carnivoro potrebbe mangiare erbivori
}
}
```

```
public class Erbivoro implements Cibo, Animale {
public void mangia(Cibo cibo) {
//un erbivoro mangia erba
}
public String getColore() {
. . .
}
}
```

In questo modo sia un carnivoro sia un erbivoro potrebbero mangiare qualsiasi cosa

Parametri covarianti

```
interface Animale<C extends Cibo> {
void mangia(C cibo);
}
```

```
public class Carnivoro implements Animale<Erbivoro> {
public void mangia(Erbivoro erbivoro) {
//un carnivoro potrebbe mangiare erbivori
}
}
public class Erbivoro<E extends Erba> implements Cibo,
Animale<E> {
public void mangia(E erba) {
//un erbivoro mangia erba
}
public String getColore() {
. . .
}
}
```


Parametri covarianti

```
public class TestAnimali {
public static void main(String[] args) {
Animale<Erbivoro> tigre = new Carnivoro<Erbivoro>();
Erbivoro<Erba> erbivoro = new Erbivoro<Erba>();
tigre.mangia(erbivoro);
}
}
```