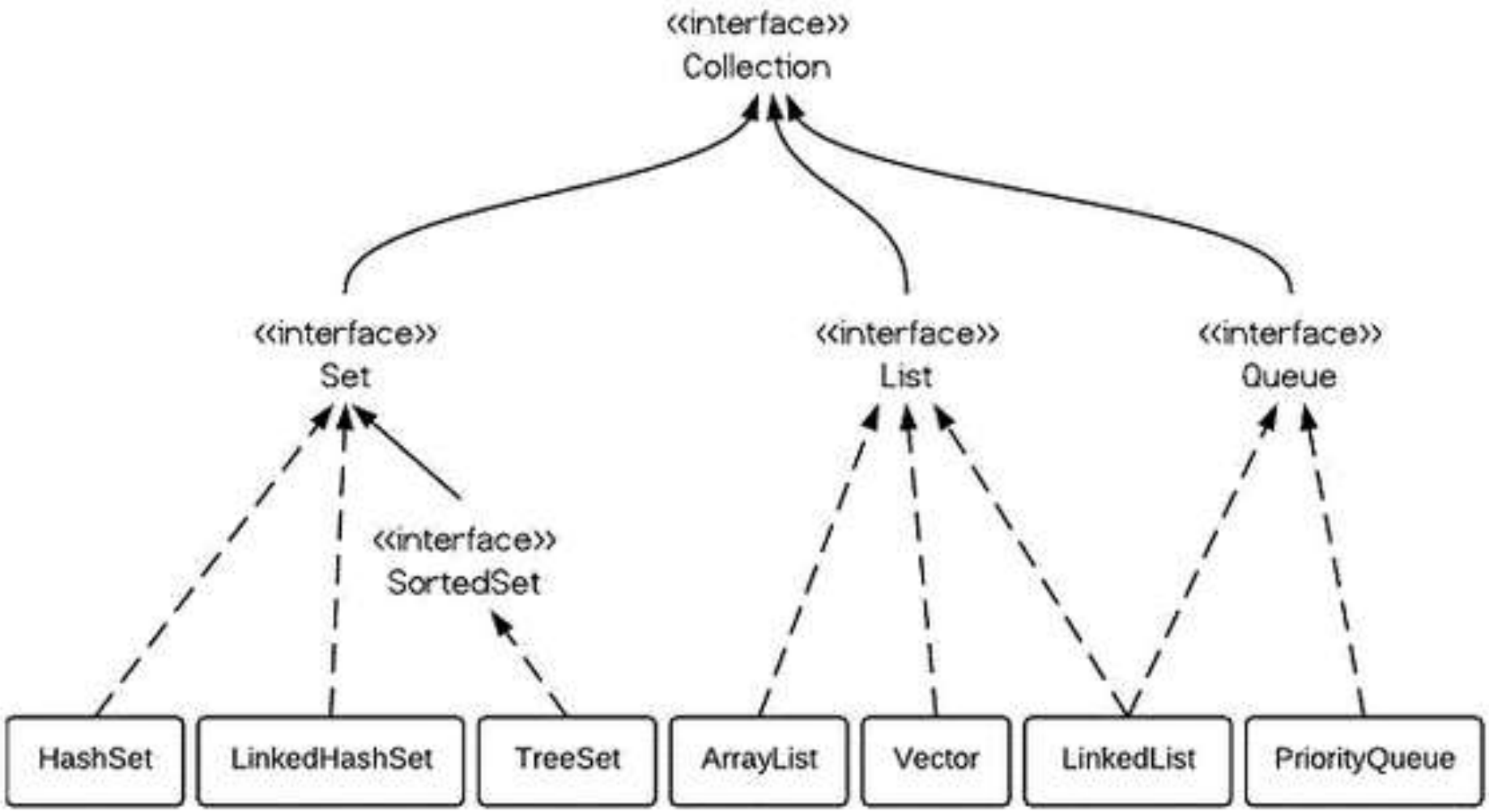
Scandire .....

….. Collezioni


«interface»
Collection
«interface»
Set
«interface»
List
«interface»
Queue
«interface»
SortedSet
HashSet
LinkedHashSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue

for generalizzato

Dalla versione Tiger (java 1.5)

Il ciclo for generalizzato

- Il ciclo for generalizzato scandisce tutti gli elementi di una raccolta:

```
double[] data = . . .;
double sum = 0;

for (double e : data)   // Si può leggere questo loop co
                        // "for each e in data" "per ogn

{
   sum = sum + e;
}
```

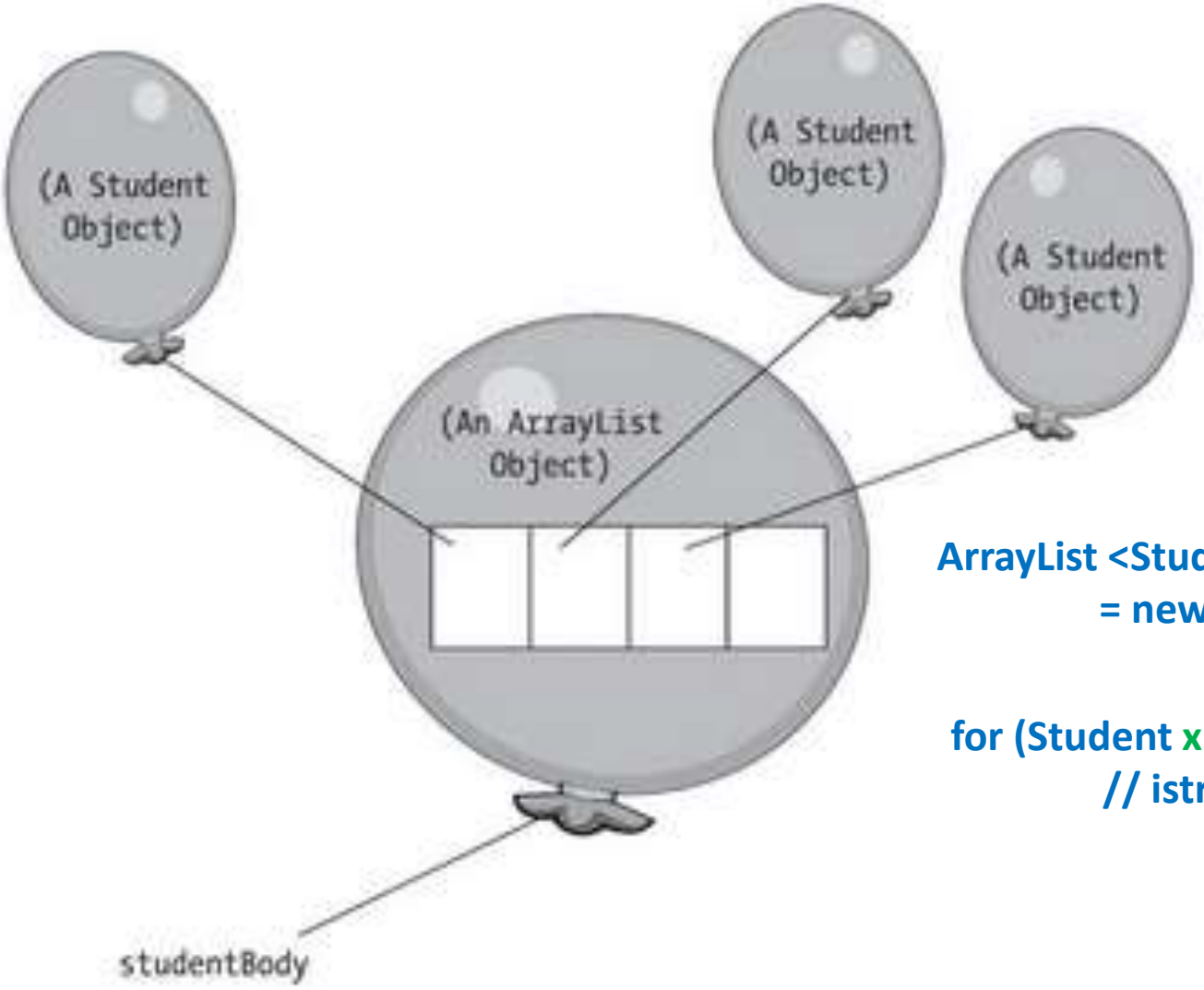
utilizzabile anche per array statici

Sintassi del ciclo for generalizzato: **for (*Tipo variabile* : *raccolta*)**
enunciato // **ciclo** for-each usato per scandire gli elementi

Obiettivo:

Eseguire un ciclo avente un'iterazione per ogni elemento

Scandire collezioni




```
ArrayList <Student> studentBody
        = new ArrayList <Student> ();

for (Student x: studentBody)
        // istruzione che usa x
```

Carrello della spesa:
iteratore vs for generalizzato

Carrello della spesa: Classe astratta

Supponiamo di voler realizzare un classico carrello della spesa partendo dalla sua **definizione astratta** attraverso, ad esempio, la seguente classe:

```
public abstract class ShoppingCart<T> {

    protected List<T> products; // la collezione come attributo

    public abstract void add(T product);
    public abstract void delete(T product);
    public abstract int getNumberOfProducts();
    public abstract List<T> getProducts();
}
```

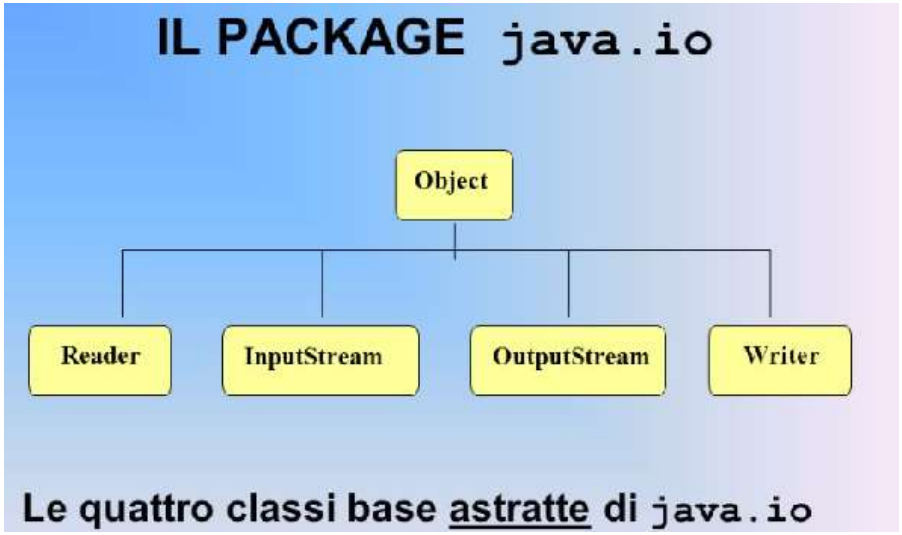
nel caso delle classi astratte, l'implementazione dei metodi "lasciati in bianco" dovrà essere fornita *da una sottoclasse*

CLASSI ASTRATTE

Una classe che può definire **anche** metodi astratti (cioè senza corpo).

Usata per poter dichiarare caratteristiche comuni fra classi di una **determinata gerarchia**.

La classe astratta non può essere instanziata.



Carrello della spesa: un prodotto

Realizziamo quindi una classe che rappresenti un semplice **prodotto** da inserire nel carrello:

```
public class Product {
        protected String description;
        protected double price;
        public Product(String description, double price){
                this.description = description;
                this.price=price;
        }
    //.... get e set
public String getDescription(){   //almeno
        return description;
   }
   public double getPrice(){      //almeno
        return price;
   }
}
```

Carrello della spesa: implementazione

Proseguiamo con l'implementazione del carrello:

```
public class ShoppingCartImpl extends ShoppingCart <Product> {
    public ShoppingCartImpl() {
        products = new ArrayList<Product>();
    }
    @Override
    public void add(Product product){
        products.add(product);
    }
    @Override
    public void delete(Product product){
        products.remove(product);
    }
    @Override
    public int getNumberOfProducts(){
        return products.size();
    }
    @Override
    public List<Product> getProducts() {
        return products;
    }
}
```

Carrello della spesa: applicazione per test (1)

Ed infine, all'interno di una applicazione, nel metodo **main**, testiamo il tutto:

```
ShoppingCart<Product> shoppingCart= new ShoppingCartImpl();
Product productA = new Product("A",10.20);
Product productB = new Product("B",2.39);
shoppingCart.add(productA);
shoppingCart.add(productB);

List<Product> products = shoppingCart.getProducts();

Iterator<Product> iterator = products.iterator();          // uso Iterator

while(iterator.hasNext()) {
        Product product = iterator.next();
        System.out.println(product.getDescription());
        System.out.println(product.getPrice());
}
```

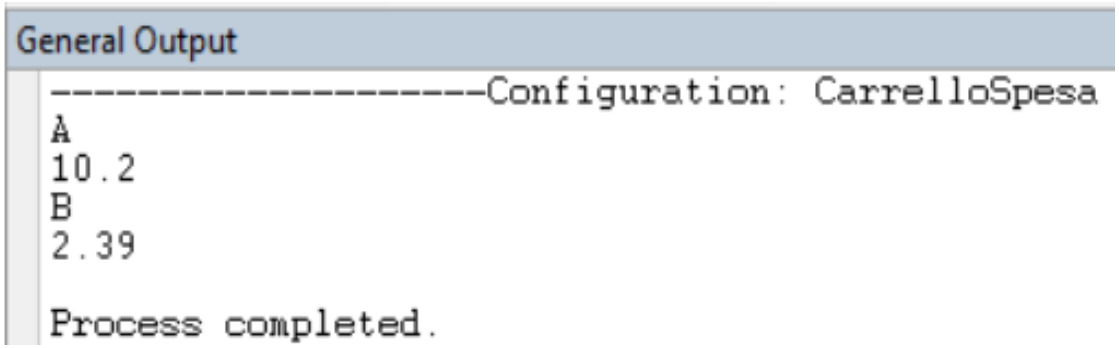
Carrello della spesa: applicazione per test (2)

Ed infine, all'interno di una applicazione, nel metodo main, testiamo il tutto:

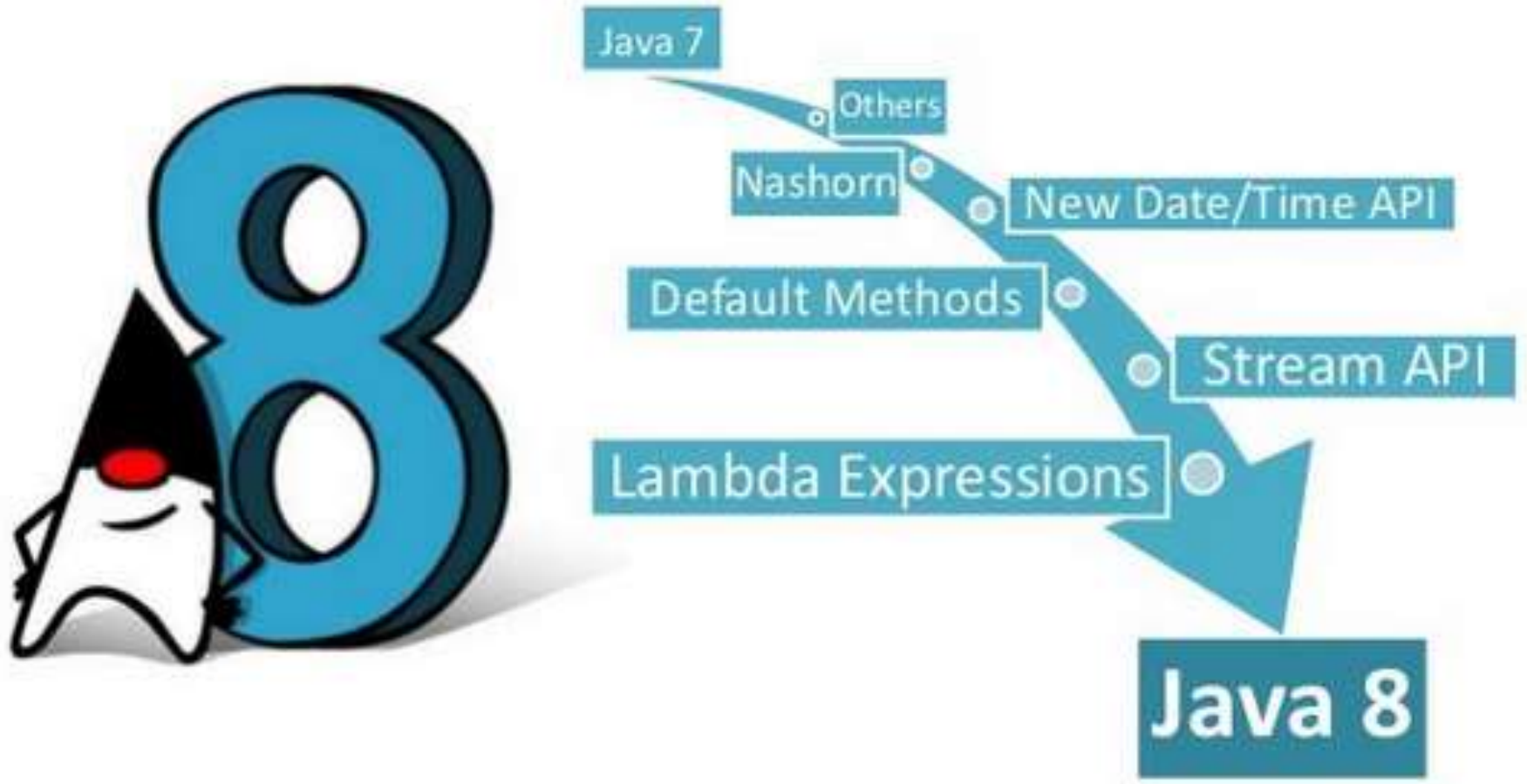
```
ShoppingCart<Product> shoppingCart= new ShoppingCartImpl();
Product productA = new Product("A",10.20);
Product productB = new Product("B",2.39);
shoppingCart.add(productA);
shoppingCart.add(productB);

List<Product> products = shoppingCart.getProducts();
                                        // senza iterator - semplice for generalizzato
System.out.println("Stampa con for-each");
for ( Product x : products){
                        System.out.println(x.getDescription());
                        System.out.println(x.getPrice());
}
```

OUTPUT


General Output
--------------------Configuration: CarrelloSpesa
A
10.2
B
2.39
Process completed.

Scandire collezioni: Java 8



Java 8: stile di programmazione funzionale

Codice (1) …. array dinamico

```
import java.util.*;

public class Prova8 {

    ArrayList<String> elenco = new ArrayList<String>();

    /**
    * Metodo per inizializzare
    * gli elementi dell'elenco
    */
    public void creaElenco(){
            elenco.add("Rossi");
            elenco.add("Verdi");
            elenco.add("Gialli");
    }
```

… proseguo

Codice(2) …. uso ForEach

```
/**
 * Metodo per visualizzare a terminale
 * il numero e gli elementi dell'elenco
 */
 public void vediNew(){

        // Java 8 è stato esteso con una API di tipo pipe-filter specifica per le
 Collection
        // Un ciclo può essere espresso con la Stream API del JDK 8
        // in un modo molto più leggibile, facendo ampio uso di espressioni lambda:

        elenco.forEach(x -> {
                                        System.out.println(x);});

         // oppure senza fare uso di espressioni lambda:
         System.out.println("\nUso ForEach senza fare uso di espressioni lambda");
       elenco.forEach(System.out::println);
 }
```

Codice (3) … main

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

        Prova8 o = new Prova8();

        o.creaElenco();

        System.out.println("\nUso ForEach");

        o.vediNew();
    }
} // fine applicazione
```