

Identità, uguaglianza, confronto

Marco Patrignani

[mailto: marco.patrignani@unitn.it](mailto:marco.patrignani@unitn.it)

(basato sulle slides di Picco, Ronchetti, Marchese)

Uguaglianza

```
public class Test {  
    public static void main(String[] a){ new Test(); }  
    Test() {  
        int k1 = 1;  
        int k2 = 1;  
        System.out.println(k1==k2);  
    }  
}
```

true

Uguaglianza

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
    }  
}
```

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y);  
    }  
}
```

p1 e p2 sono uguali?

Uguaglianza

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p1==p2);  
    }  
}
```

false

Uguaglianza

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        int k1 = 1;  
        int k2 = k1;  
        System.out.println(k1==k2);  
    }  
}
```

true

Uguaglianza

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=p1;  
        System.out.println(p1==p2);  
    }  
}
```

true

Uguaglianza

```
int k1=1;  
int k2=1;
```

k1==k2 ? **TRUE**

```
int k1=1;  
int k2=k1;
```

k1==k2 ? **TRUE**

```
P p1=new P();  
p1.x=1; p1.y=2;  
P p2=new P();  
p2.x=1; p2.y=2;
```

p1==p2 ? **FALSE**

```
P p1=new P();  
p1.x=1; p1.y=2;  
P p2= p1;
```

p1==p2 ? **TRUE**

Perché? (ricordiamoci dell'allocazione di memoria...)

Oggetti diversi

```
public class Test {  
    public static void main(String a[]) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        System.out.println(p1);  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p2);  
        p1.x=3;  
        System.out.println(p1);  
        System.out.println(p2);  
    }  
}
```

x=1 ; y=2

x=1 ; y=2

x=3 ; y=2

x=1 ; y=2

Oggetti diversi?

```
public class Test {  
    public static void main(String []a){new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1;  
        p1.y=2;  
        System.out.println(p1);  
        P p2=p1;  
        p2.x=3;  
        System.out.println(p1);  
    }  
}
```

x=1 ; y=2

x=3 ; y=2

p1 e p2 si riferiscono allo **stesso** oggetto!

Come testare l'egualità?

```
public class Test {  
    public static void main(String a[]){new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        // cosa scriviamo qui?  
    }  
}
```

Operatore ==



```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p1==p2);  
    }  
}
```

== testa l'**identità**
(due riferimenti puntano
allo stesso oggetto)

false

java.lang

Class Object

java.lang.Object

```
public class Object
```

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

See Also:

Class

Constructor Summary

Constructors

Constructor and Description

[Object\(\)](#)

Method Summary

Methods

Modifier and Type

`protected Object`

`boolean`

Method and Description

`clone()`

Creates and returns a copy of this object.

`equals(Object obj)`

Indicates whether some other object is "equal to" this one.

equals () = testa l'uguaglianza
(due riferimenti puntano a oggetti «uguali»)

Metodo equals ()

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p1.equals(p2));  
    }  
}
```

false

Metodo `equals()`

*The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object
(`x==y` has the value `true`)*

Ma allora a cosa serve?!?

...

Un possibile (errato) `equals()`

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y);  
    }  
    public boolean equals(P var) {  
        return (x==var.x && y==var.y)  
    }  
}
```

Funziona?

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p1.equals(p2));  
        System.out.println(p1==p2);  
    }  
}
```

true
false

Un possibile (errato) `equals()`

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+"; y="+y);  
    }  
    public boolean equals(P var) {  
        return (x==var.x && y==var.y)  
    }  
}
```

Overloading o
overriding?!?

`equals()` deve essere ridefinito
(override) da `Object`!

`boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

Attenzione: `equals()` va ridefinito

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y);  
    }  
    public boolean equals(Object var) {  
        return (x==var.x && y==var.y)  
    }  
}
```

`equals()` in `Object`
è la base per ridefinire la
nozione di uguaglianza
come necessario per
l'applicazione

`equals()` viene invocato
automaticamente da molte librerie
Java (es. Collection); se si effettua
overloading invece di overriding,
viene invocato il metodo sbagliato!

Funziona?

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p1.equals(p2));  
        System.out.println(p1==p2);  
    }  
}
```

equals () deve
confrontare due Object!

```
error: cannot find symbol  
    return (x==var.x && y==var.y);  
          ^
```

Altro problema ...

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        Integer p2=new Integer(3);  
        System.out.println(p1.equals(p2));  
        System.out.println(p1==p2);  
    }  
}
```

Sono classi diverse:
deve ritornare
false a priori!

Soluzione

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y);  
    }  
    public boolean equals(Object var) {  
        if (!(var instanceof P)) return false;  
        return (x==((P)var).x && y==((P)var).y);  
    }  
}
```

Funziona?

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=new P();  
        p2.x=1; p2.y=2;  
        System.out.println(p1.equals(p2));  
        System.out.println(p1==p2);  
    }  
}
```

true
false

Altro problema...

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        P p2=null;  
        System.out.println(p1.equals(p2));  
        System.out.println(p1==p2);  
    }  
}
```

p2 non punta a nessun oggetto:
deve ritornare **false** a priori!

Soluzione

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y) ;  
    }  
    public boolean equals(Object var)  
    {  
        if (var == null) return false;  
        if (!(var instanceof P)) return false;  
        return (x==((P)var).x && y==((P)var).y);  
    }  
}
```

In realtà non serve
se si usa con **instanceof**
ma meglio inserirlo (vedi dopo)

E le sottoclassi?

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        Q p2 = new Q();  
        p2.x=1; p2.y=2;  
        System.out.println(p1.equals(p2));  
        System.out.println(p1==p2);  
    }  
}
```

```
class Q extends P {  
    int z;  
}
```

true
false

E le sottoclassi?

```
class P {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+"; y="+y);  
    }  
    public boolean equals(Object var) {  
        if(var==null) return false;  
        if (var.getClass() != this.getClass())  
            return false;  
        return (x==((P)var).x && y==((P)var).y)  
    }  
}
```

getClass() è definito su Object e dunque presente (ereditato) in ogni classe

E le sottoclassi?

```
public class Test {  
    public static void main(String[] a) {new Test();}  
    Test() {  
        P p1=new P();  
        p1.x=1; p1.y=2;  
        Q p2 = new Q();  
        p2.x=1; p2.y=2;  
        System.out.println(p1.equals(p2));  
        System.out.println(p1==p2);  
    }  
}
```

```
class Q extends P {  
    int z;  
}
```

false
false

E le sottoclassi?

- Per verificare la compatibilità del tipo dell'oggetto `o` passato come parametro a `equals()` abbiamo due possibilità

```
if (o.getClass() != this.getClass()) return false;
```

```
if (!(o instanceof P)) return false;
```

- La prima vincola il tipo del parametro `o` a **coincidere** con quello dell'oggetto su cui `equals()` è invocato
- La seconda consente il confronto anche con oggetti appartenenti a una sua **sottoclasse**
- **Quale usare? Dipende dall'applicazione!**

Proprietà di `equals()`

- Il metodo `equals()` implementa una relazione di equivalenza fra elementi non nulli e deve soddisfare le seguenti proprietà:
 1. **riflessiva**: per ogni riferimento non nullo `x`, `x.equals(x)` ritorna `true`
 2. **simmetrica**: per ogni riferimento non nullo `x` e `y`, `x.equals(y)` ritorna `true` se e solo se `y.equals(x)` ritorna `true`
 3. **transitiva**: per ogni riferimento non nullo `x`, `y`, `z`, se `x.equals(y)` e `y.equals(z)` ritornano `true` allora `x.equals(z)` ritorna `true`
 4. **consistente**: per ogni riferimento non nullo `x` e `y`, invocazioni diverse di `x.equals(y)` ritornano lo stesso valore, se nessuna delle informazioni usate da `equals()` sono state modificate
 5. per ogni riferimento non nullo `x`, `x.equals(null)` ritorna `false`

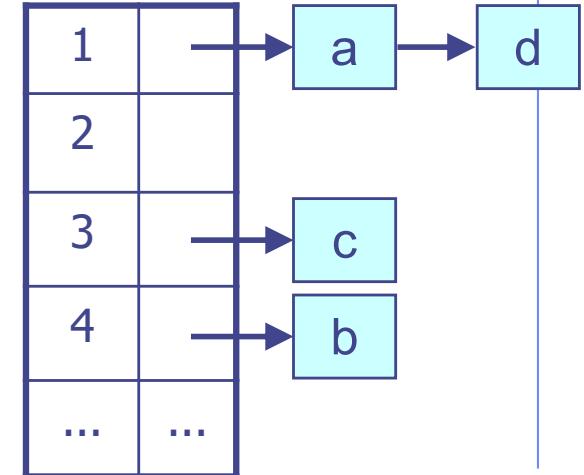
Se il metodo viene ridefinito, garantire queste proprietà
è compito del programmatore

Ridefinire `equals()` non basta...

- La classe `Object` fornisce anche un metodo `hashCode()`
- Rappresenta una «funzione *hash*» non iniettiva (e quindi non invertibile)
che mappa un oggetto su un intero
 - Sono possibili «collisioni» cioè oggetti diversi mappati sullo stesso intero
- Viene utilizzata dal Java runtime per gestire in maniera efficiente strutture dati di uso comune
- Il comportamento di `hashCode()` è legato al metodo `equals()`

A che serve hashCode () ?

- Ricercare un elemento all'interno di una struttura dati è costoso (ricerca lineare)
- La funzione hash consente di velocizzare il processo, sfruttando l'associazione fra hash code e oggetto:
 - Quando si inserisce un nuovo elemento, viene posizionato all'indice corrispondente all'hash code
 - Elementi diversi con lo stesso hash code sono organizzati in una lista (*bucket*)
 - Per verificare se un elemento si trova nella struttura dati, basta calcolare il suo hash code e cercarlo nella (corta) lista corrispondente



Proprietà di hashCode ()

1. Se invocato più di una volta sullo stesso oggetto deve ritornare lo stesso intero
 - questo può essere diverso in esecuzioni diverse dell'applicazione; l'importante è che rimanga identico all'interno di una singola esecuzione
2. se due oggetti sono uguali secondo il metodo `equals ()` allora `hashCode ()` deve ritornare lo stesso intero
3. non è richiesto che a due oggetti diversi (secondo `equals ()`) siano associati due `hashCode ()` diversi
 - Tuttavia, questo in generale migliora le performance in alcune strutture dati («hash-based»)

Se il metodo viene ridefinito, garantire queste proprietà
è compito del programmatore

In sostanza...

- Una classe che ridefinisce il metodo `equals()` deve altresì ridefinire il metodo `hashCode()`
- Una semplice «regoletta» è:
 1. oggetti uguali \Rightarrow hash code uguali
 2. hash code diversi \Rightarrow oggetti diversi

Nota: non valgono i viceversa!

In sostanza...

- ... e in altre parole:
 - se **`o1.hashCode() != o2.hashCode()`** allora **`o1`** e **`o2`** sono certamente diversi
 - se **`o1.hashCode() == o2.hashCode()`** devo verificare se **`o1.equals(o2)`** per poter dire se **`o1`** e **`o2`** sono uguali

consente di ridurre in maniera significativa il tempo di esecuzione rispetto a invocare **`equals()`**

Esempio di funzione *hash*

- Per le stringhe, potrei pensare di calcolare lo hash code come somma dei codici ASCII dei caratteri che le compongono

$$\text{"ABBA"} \rightarrow 65+66+66+65 = 262$$

$$\text{"ABBB"} \rightarrow 65+66+66+66 = 263$$

$$\text{"ABAB"} \rightarrow 65+66+65+66 = 262$$

- Le collisioni sono molto probabili ...

Come scrivere un «buon» hashCode () ?

- È bene usare gli stessi campi su cui è definito equals ()
- Definire una funzione hash buona (poche collisioni) non è immediato
- Si possono riusare quelle già definite da Java

```
int prime = 31;           attributi (non nulli) di una classe Person
int result = 1;
result = prime * result + firstName.hashCode();
result = prime * result + lastName.hashCode();
return result;
```

- Oppure (a partire da Java 7):

```
return Objects.hash(firstName, lastName);
```

Nel dubbio...

- Gli IDE offrono la possibilità di generare automaticamente il metodo **hashCode ()**
- Inoltre...

```
public int hashCode() { return 0; }
```

- ... è un'implementazione corretta, che soddisfa tutte le proprietà
 - Tuttavia, può generare forti inefficienze (perché?) e **dunque è da evitare**

equals () e hashCode () : l'importanza di usarli correttamente

```
import java.util.*;  
public class Test {  
    Set<Element> s = new HashSet<>();  
    public static void main(String[] args) { new Test(); }  
    public Test() {  
        Random r = new Random();  
        for (int i = 0; i < 10; i++)  
            s.add(new Element(r.nextInt(10)));  
        System.out.println("Set elements: " + s);  
    }  
}
```

Set elements: [8, 8, 6, 0, 1, 0, 9, 5, 8, 5]

```
class Element {  
    int x;  
    public Element(int x) { this.x = x; }  
    public String toString() {  
        return x + "";  
    }  
}
```

```
}
```

equals () e hashCode () : l'importanza di usarli correttamente

```
import java.util.*;  
public class Test {  
    Set<Element> s = new Ha:  
    public static void main  
    public Test() {
```

```
        Random r = new Random();  
        if (this == obj) { return true; }  
        if (obj == null) { return false; }  
        if (getClass() != obj.getClass()) { return false; }  
        if (this.x != ((Element) obj).x) { return false; }  
        return true;  
    }
```

```
        Random r = new Random();  
        for (int i = 0; i < 10; i++)  
            s.add(new Element(r.nextInt(10)));  
        System.out.println("Set elements: " + s);
```

```
Set elements: [9, 7, 5, 8, 5, 6, 5, 8, 1, 4]
```

```
Element toSearch = new Element(5);  
System.out.print("toSearch in set? ");  
if (s.contains(toSearch)) System.out.println("yes");  
else System.out.println("no");
```

```
toSearch in set? no
```

```
}
```

equals() e hashCode(): l'importanza di usarli correttamente

```
import java.util.*;  
public class Test {  
    Set<Element> s = new HashSet<Element>();  
    public static void main(String[] args) {  
        Test t = new Test();  
        Random r = new Random();  
        for (int i = 0; i < 10; i++)  
            s.add(new Element(r.nextInt(10)));  
        System.out.println("Set elements: " + s);  
        System.out.print("Hashcodes: ");  
        for (Element t : s)  
            System.out.print(t.hashCode() + " ");  
        System.out.println("");  
        Element toSearch = new Element(5);  
        System.out.print("toSearch in set? ");  
        if (s.contains(toSearch)) System.out.println("yes");  
        else System.out.println("no");  
        System.out.println("HashCode of toSearch: " + toSearch.hashCode());  
    }  
}
```

```
public boolean equals(Object obj) {  
    if (this == obj) { return true; }  
    if (obj == null) { return false; }  
    if (getClass() != obj.getClass()) { return false; }  
    if (this.x != ((Element) obj).x) { return false; }  
    return true;  
}
```

Set elements:
[9, 7, 5, 8, 5, 6, 5, 8, 1, 4]

Hashcodes:
1550089733 865113938
118352462 1808253012
589431969 1118140819
1028566121 1311053135
1975012498 1442407170

toSearch in set? no

HashCode of toSearch: 1252169911

equals () e hashCode () : l'importanza di usarli correttamente

```
public int hashCode() { return Objects.hash(x); }
```

```
import java.util.*;
public class Test {
    Set<Element> s = new HashSet<>();
    public static void main(String[] args) { new Test(); }
    public Test() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            s.add(new Element(r.nextInt(10)));
        System.out.println("Set elements: " + s);
        System.out.print("Hashcodes: ");
        for (Element t : s)
            System.out.print(t.hashCode() + " ");
        System.out.println("");
        Element toSearch = new Element(5);
        System.out.print("toSearch in set? ");
        if (s.contains(toSearch)) System.out.println("yes");
        else System.out.println("no");
        System.out.println("Hashcode of toSearch: " + toSearch.hashCode());
    }
}
```

Set elements: [1, 2, 5, 7, 8, 0]

Hashcodes:
32 33 36 38 39 31

toSearch in set? yes

Hashcode of toSearch: 36

E se volessimo insiemi ordinati?

```
import java.util.*;  
public class Test {  
    SortedSet<Element> s = new TreeSet<>();  
    public static void main(String[] args) { new Test(); }  
    public Test() {  
        Random r = new Random();  
        for (int i = 0; i < 10; i++)  
            s.add(new Element(r.nextInt(10)));  
        System.out.println("Set elements: " + s);  
    }  
}
```

Exception in thread "main"
java.lang.ClassCastException:
Element cannot be cast to
java.lang.Comparable

Ordinamento e confronto di oggetti

- L'interfaccia **Comparable<T>** consente di definire un **ordinamento totale** ("ordinamento naturale") fra gli oggetti che la implementano
 - Ad esempio, per **String** è l'ordine lessicografico, per **Date** quello cronologico, etc.
- Definisce un unico metodo
int compareTo (T o)
che ritorna
 - un intero negativo se **this** è minore di **o**
 - un intero positivo se **this** è maggiore di **o**
 - 0 se **this** è uguale a **o**

Proprietà di Comparable

1. Per ogni `x` e `y`, deve valere
`-sgn(x.compareTo(y)) == sgn(y.compareTo(x))`
2. La relazione deve essere transitiva:
`(x.compareTo(y)>0 && y.compareTo(z)>0)`
 $\Rightarrow x.compareTo(z)>0$
3. Per ogni `x`, `y`, `z` deve valere
`x.compareTo(y)==0` \Rightarrow
`sgn(x.compareTo(z)) == sgn(y.compareTo(z))`
4. È fortemente consigliato (anche se non strettamente richiesto) che
`(x.compareTo(y)==0) == (x.equals(y))`

E se volessimo insiemi ordinati?

```
class Element implements Comparable<Element> {  
    ...  
    public int compareTo(Element o) {  
        if(this.equals(o)) return 0;  
        if(this.x < o.x) return -1;  
        return 1;  
    }  
  
import java.util.*;  
  
public class Test {  
    SortedSet<Element> s = new TreeSet<>();  
    public static void main(String[] args) { new Test(); }  
    public Test() {  
        Random r = new Random();  
        for (int i = 0; i < 10; i++)  
            s.add(new Element(r.nextInt(10)));  
        System.out.println("Set elements: " + s);  
    }  
}
```

Set elements: [0, 3, 4, 5, 6, 7, 8]

A cosa serve Comparable?

- Gli oggetti che implementano Comparable possono essere elementi di un SortedSet o chiavi in un SortedMap
 - L'inserimento di elementi che non implementano Comparable genera un'eccezione a runtime
- Un oggetto di tipo List oppure array, i cui elementi implementino Comparable può essere ordinato semplicemente invocando **Collections.sort(o)** oppure **Arrays.sort(o)**

Esempio

```
public class TestCar {  
    List<Car> macchine;  
    public static void main(String[] args) { new TestCar(); }  
    TestCar(){  
        macchine = new LinkedList<Car>();  
        Car a = new Car(100, "Fiat Cinquecento");  
        macchine.add(a);  
        Car b = new Car(250, "Porsche Carrera");  
        macchine.add(b);  
        Car c = new Car(180, "Renault Megane");  
        macchine.add(c);  
        System.out.println(macchine);  
        Collections.sort(macchine);  
        System.out.println(macchine);  
    }  
}
```

```
[(100,Fiat Cinquecento), (250,Porsche Carrera), (180,Renault Megane)]  
[(100,Fiat Cinquecento), (180,Renault Megane), (250,Porsche Carrera)]
```

Esempio

```
class Car implements Comparable<Car> {
    public int maxSpeed;
    public String name;
    Car(int v, String name) {
        maxSpeed = v;
        this.name = name;
    }
    public String toString() {
        return "(" + maxSpeed + "," + name + ')';
    }
    public boolean equals(Object o) {...}
    public int compareTo(Car o){
        if (this.equals(o)) return 0;
        if (maxSpeed < o.maxSpeed) return -1;
        return 1;
    }
}
```

con generics
(Java 5+)

Esempio

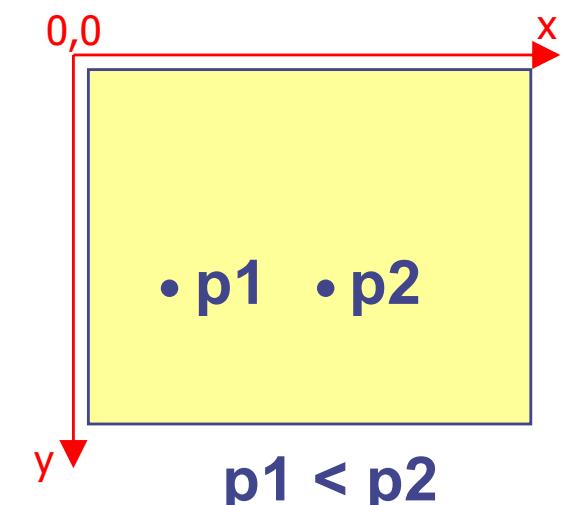
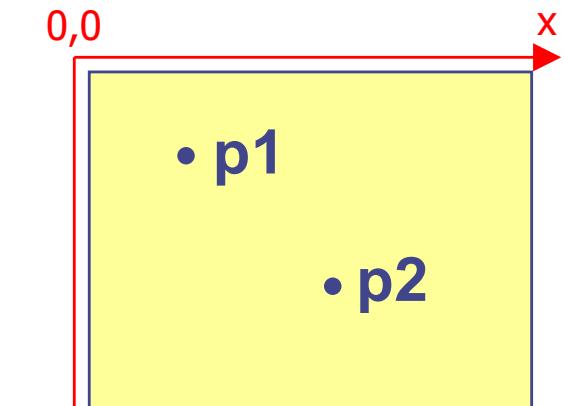
```
class Car implements Comparable {  
    public int maxSpeed;  
    public String name;  
    Car(int v, String name) {  
        maxSpeed = v;  
        this.name = name;  
    }  
    public String toString() {  
        return "(" + maxSpeed + "," + name + ')';  
    }  
    public boolean equals(Object o) {...}  
    public int compareTo(Object o){  
        if (!(o instanceof Car)) System.exit(1);  
        if (this.equals(o)) return 0;  
        if (maxSpeed < ((Car) o).maxSpeed) return -1;  
        return 1;  
    }  
}
```

senza generics
(< Java 5)

Comparable ...



```
class Point implements Comparable<Point> {  
    int x, y;  
    ...  
    public int compareTo(Point p) {  
        // ordino sulle y  
        int retval = y - p.y;  
        // a parità di y ordino sulle x  
        if (retval == 0)  
            retval = x - p.x;  
        return retval;  
    }  
}  
  
class NamedPoint extends Point { ... }
```



Comparable ...

```
public class Test {  
    public static void main(String[] args) { new Test(); }  
    Test() {  
        List<Point> l = new LinkedList<>();  
        l.add(new Point(40,20));  
        l.add(new Point(10,20));  
        l.add(new Point(20,10));  
        l.add(new Point(20,20));  
        System.out.println(l);  
        Collections.sort(l);  
        System.out.println(l);  
        l.clear();  
        l.add(new NamedPoint("B",40,20));  
        l.add(new NamedPoint("D",10,20));  
        l.add(new NamedPoint("C",20,10));  
        l.add(new NamedPoint("A",20,20));  
        System.out.println(l);  
        Collections.sort(l);  
        System.out.println(l);  
    }  
}
```

[(40,20), (10,20), (20,10), (20,20)]
[(20,10), (10,20), (20,20), (40,20)]

E se volessi ordinare per nome?

[(B,40,20), (D,10,20), (C,20,10), (A,20,20)]
[(C,20,10), (D,10,20), (A,20,20), (B,40,20)]

... o Comparator?

- L'interfaccia **Comparator<T>** consente di «delegare» il confronto a una classe separata
 - Consente maggiore flessibilità
- Scelta obbligata se si vuole confrontare con un criterio diverso dall'«ordinamento naturale» rappresentato da **Comparable**
 - ... e fissato all'interno della classe che la implementa
- Fornisce il metodo
int compare(T o1, T o2)

Nota: Comparable e Comparator sono parte del Java Collections Framework, ma si possono usare anche indipendentemente da esso

... o Comparator?

```
class NamedPointComparatorByName implements Comparator<NamedPoint> {  
    public int compare(NamedPoint p1, NamedPoint p2) {  
        return p1.getName().compareTo(p2.getName());  
    }  
}
```

fornisce il confronto per nome,
complementa quello naturale (in **Point**) per coordinata

```
class NamedPointComparatorByXY implements Comparator<NamedPoint> {  
    public int compare(NamedPoint p1, NamedPoint p2) {  
        int retval = p1.y - p2.y;  
        if (retval == 0) retval = p1.x - p2.x;  
        return retval;  
    }  
}
```

equivalente al confronto
naturale in **Point**

con generics
(Java 5+)

... o Comparator?

```
class NamedPointComparatorByName implements Comparator {
    public int compare(Object p1, Object p2) {
        NamedPoint np1 = (NamedPoint) p1;
        NamedPoint np2 = (NamedPoint) p2;
        return (np1.getName().compareTo(np2.getName()));
    }
}
```

fornisce il confronto per nome,
complementa quello naturale (in **Point**) per coordinata

```
class NamedPointComparatorByXY implements Comparator {
    public int compare(Object p1, Object p2) {
        NamedPoint np1 = (NamedPoint) p1;
        NamedPoint np2 = (NamedPoint) p2;
        int retval = np1.y - np2.y;
        if (retval == 0) retval = np1.x - np2.x;
        return retval;
    }
}
```

equivalente al confronto
naturale in **Point**

senza generics
(< Java 5)

... o Comparator?

```

TestCompare() {
    List<Point> l = new LinkedList<>(
        // esempi con Comparable
        l.clear();
        l.add(new NamedPoint("B",40,20));
        l.add(new NamedPoint("D",10,20));
        l.add(new NamedPoint("C",20,10));
        l.add(new NamedPoint("A",20,20));
        System.out.println(l);
        Collections.sort(l, new NamedPointComparatorByXY());
        System.out.println(l);
        l.clear();
        l.add(new NamedPoint("B",40,20));
        l.add(new NamedPoint("D",10,20));
        l.add(new NamedPoint("C",20,10));
        l.add(new NamedPoint("A",20,20));
        System.out.println(l);
        Collections.sort(l, new NamedPointComparatorByName());
        System.out.println(l);
}

```

[(40,20), (10,20), (20,10), (20,20)]
[(20,10), (10,20), (20,20), (40,20)]
[(B,40,20), (D,10,20), (C,20,10), (A,20,20)]
[(C,20,10), (D,10,20), (A,20,20), (B,40,20)]

[(B,40,20), (D,10,20), (C,20,10), (A,20,20)]
[(C,20,10), (D,10,20), (A,20,20), (B,40,20)]

[(B,40,20), (D,10,20), (C,20,10), (A,20,20)]
[(A,20,20), (B,40,20), (C,20,10), (D,10,20)]