

1

Hashcode

# equals e hashCode

Programmers should take note that  
any class that overrides the `Object.equals`  
method must also override the  
`Object.hashCode` method  
in order to satisfy the general contract for the  
`Object.hashCode` method.

In particular, `c1.equals(c2)` implies that  
`c1.hashCode()==c2.hashCode()`  
(the vice versa need not be true)

## 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 ()`

## equals e hashCode

`c1.equals(c2) ==> c1.hashCode()==c2.hashCode()`

Uguaglianza degli oggetti implica hashCode uguali

Diversità di hashCode implica non uguaglianza degli oggetti

`c1.hashCode()!=c2.hashCode() ==> ! c1.equals(c2)`

Non valgono i viceversa!!!

## equals e hashCode

if (c1.hashCode()!=c2.hashCode() )

c1 e c2 sicuramente diversi

if (c1.hashCode()==c2.hashCode() )

per sapere se c1 è uguale a c2

devo usare la equals

**E' un meccanismo di "fail quick"**

## 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

# Regoletta...

Oggetti UGUALI  $\Rightarrow$  Hashcode UGUALI

Hashcode DIVERSI  $\Rightarrow$  Oggetti diversi

**NON VALGONO I VICEVERSA!!!**

# Esempio di funzione *hash*

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

"ABBA" → 65+66+66+65 = 262

"ABBB" → 65+66+66+66 = 263

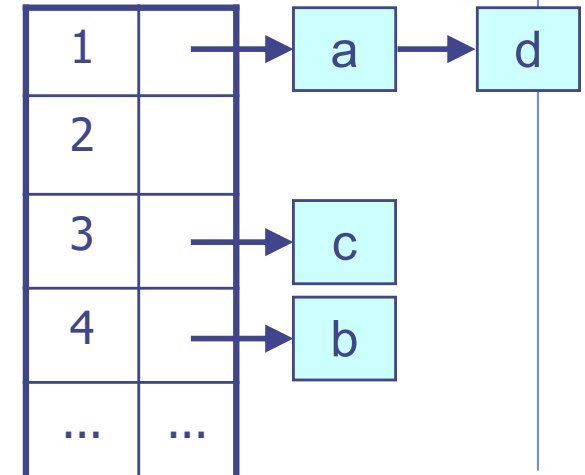
"ABAB" → 65+66+65+66 = 262

- Le collisioni sono piuttosto probabili ...



# 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



# ATTENZIONE!

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects.

(This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java programming language.)

# Nel dubbio...

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

**Inefficiente, ma corretto!**

**Per approfondimenti:**

<http://eclipsesource.com/blogs/2012/09/04/the-3-things-you-should-know-about-hashcode/>

# 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**

# 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;
```

```
int result = 1;
```

```
result = prime * result + firstName.hashCode();
```

```
result = prime * result + lastName.hashCode();
```

```
return result;
```

attributi (non nulli) di una classe `Person`

- Oppure (a partire da Java 7):

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

# Objects.hash

```
public static int hash(Object... values)
```

Generates a hash code for a sequence of input values. ...

.This method is useful for implementing Object.hashCode() on objects containing multiple fields.

For example, if an object that has three fields, x, y, and z, one could write:

```
@Override
```

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

# `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);
    }
}
```

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

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

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

```
import java.util.*;  
public class Test {
```

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

```
        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: [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 () :

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

```
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:**  
[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());
    }
}
```

da NetBeans

```
public int hashCode() {
    int hash = 7;
    hash = 13 * hash + this.x;
    return hash;
}
```

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

Hashcodes:  
32 33 36 38 39 31

toSearch in set? yes

HashCode of toSearch: 36

# Confronto

# E se volessimo insieme ordinati?

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

```
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);
    }
}
```

# Ordinamento e confronto di oggetti

- L'interfaccia **Comparable** 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(Object 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
$$-\text{sgn}(x.\text{compareTo}(y)) == \text{sgn}(y.\text{compareTo}(x))$$
2. La relazione deve essere transitiva:
$$(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0) \\ \Rightarrow \ x.\text{compareTo}(z) > 0$$
3. Per ogni  $x, y, z$  deve valere
$$x.\text{compareTo}(y) == 0 \Rightarrow \\ \text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$$
4. È fortemente consigliato (anche se non strettamente richiesto) che
$$(x.\text{compareTo}(y) == 0) == (x.\text{equals}(y))$$

# E se volessimo insiemi ordinati?

```
import java.util.*;  
public class Test {
```

```
class Element implements Comparable {  
    ...  
    public int compareTo(Object o) {  
        if(this.equals(o)) return 0;  
        if(this.x < ((Element) o).x) return -1;  
        return 1;  
    }  
}
```

```
    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

```
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 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;
    }
}
```

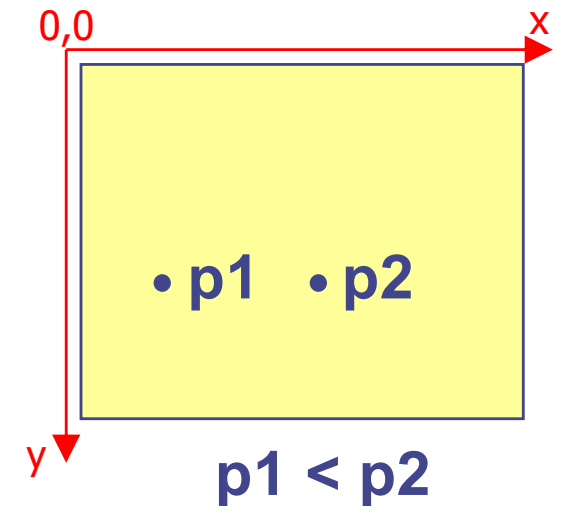
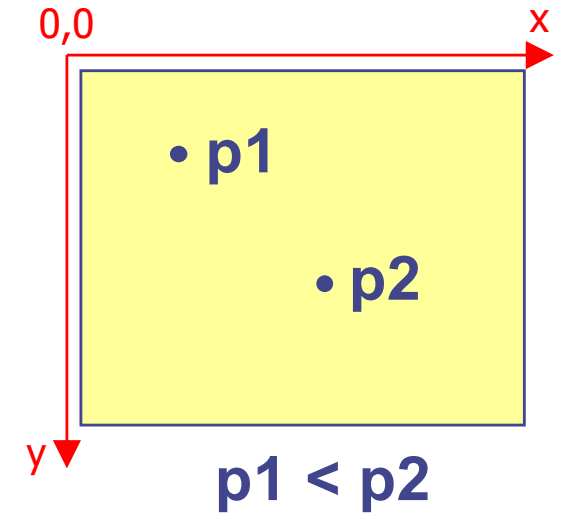
# Esempio

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

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

# Comparable ...

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



# Comparable ...

```
public class TestCompare {  
    public static void main(String[] args) { new TestCompare(); }  
    TestCompare() {  
        List<Point> list = new LinkedList();  
        list.add(new Point(40,20));  
        list.add(new Point(10,20));  
        list.add(new Point(20,10));  
        list.add(new Point(20,20));  
        System.out.println(list);  
        Collections.sort(list);  
        System.out.println(list);  
    }  
}
```

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

E se volessi ordinare  
per nome?

# Comparable ...

```

public class TestCompare {
    public static void main(String[] args) { new TestCompare(); }
    TestCompare() {
        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), (B,10,20), (D,20,20), (A,40,20)]

## ... 0 **Comparator**?

- L'interfaccia **Comparator** 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(Object o1, Object o2)**

## ... 0 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`

# ... 0 Comparator?

```

TestCompare() {
    List<NamedPoint> 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());
    // Collections.sort(l); stesso risultato
    System.out.println(l);

    System.out.println(l);
    Collections.sort(l, new NamedPointComparatorByName());
    System.out.println(l);
}

```

```

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

```

```

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

```



# Altro esempio

```
import java.util.*;

class EmployeeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        EmployeeRecord r1 = (EmployeeRecord) o1;
        EmployeeRecord r2 = (EmployeeRecord) o2;
        return r2.hireDate().compareTo(r1.hireDate());
    }
}

// lista che contiene gli impiegati
List<EmployeeRecord> employees = ...
Collections.sort(employees, new EmployeeComparator());
```