Il linguaggio Java

La superclasse universale Object

La classe **Object**

- La classe **Object** è la superclasse, diretta o indiretta, di ogni classe
- La classe **Object** definisce lo stato ed il comportamento base che ciascun oggetto deve avere e cioè l'abilità di
  - confrontarsi con un altro oggetto
  - convertirsi in una stringa
  - ritornare la classe dell'oggetto
  - attendere su una variabile condition
  - notificare che una variabile condition è cambiata

I metodi di **Object**

- Che possono essere sovrascritti
  - **clone**
  - **equals/hashCode**
  - **finalize**
  - **toString**
- Che non possono essere sovrascritti
  - **getClass**
  - **notify**
  - **notifyAll**
  - **wait**

Il metodo **toString**

- Il metodo **public String toString()** ritorna una rappresentazione testuale dell'oggetto
- Il metodo **toString** viene invocato automaticamente ogni volta che è necessaria una conversione in forma di stringa
  - ***string* + *object***
  - **System.out.println(*object*)**

Il metodo **toString**

- Il metodo **toString** della classe Object ritorna la seguente stringa (comportamento predefinito)

  "*ClassName@HashCode*"

  getClass().getName() + '@' +
  Integer.toHexString(hashCode())

  Esempio: **BankAccount@7ced01**

- Si raccomanda che ogni classe sovrascriva il metodo **toString**
- Il metodo **toString** è comodo in fase di debugging

toString per **BankAccount**

```
public class BankAccount {
    public String toString() {
        return "BankAccoun[balance=" + balance + "]";
    }
    // gli altri membri
}
```

Esempio: **"BankAccount[balance=5.0]"**

Il metodo **equals**

- Il metodo **public boolean equals(Object obj)** confronta questo oggetto con **obj** e ritorna **true** se sono uguali
- Il metodo **equals** di **Object** utilizza **==** per confrontare due oggetti: ritorna **true** se i due oggetti sono effettivamente lo stesso oggetto
- In generale, la nozione di uguaglianza dipende dall'applicazione

equals per **BankAccount**

Due conti bancari sono uguali se hanno lo stesso saldo

```
public class BankAccount {
    public boolean equals(Object obj) {
        if ( !(obj istanceof BankAccount) )
            return false;
        return ((BankAccount)obj).balance == balance;
    }
    // gli altri metodi
}
```

Il metodo equals

Il metodo **equals** deve realizzare una relazione di equivalenza su riferimenti non **null**:

- Riflessiva: per ogni riferimento **x** non **null**, **x.equals(x)** deve tornare **true**
- Simmetrica: per ogni coppia di riferimenti **x** ed **y** non **null**, **x.equals(y)** ritorna **true** se e solo se **y.equals(x)** ritorna **true**
- Transitiva: per ogni terna di riferimenti **x**, **y**, e **z**, non **null**, se **x.equals(y)** ritorna **true** e **y.equals(z)** ritorna **true**, allora **x.equals(z)** ritorna **true**

(*continua*)

Il metodo equals

- Consistente: per ogni coppia di riferimenti **x** ed **y**, invocazioni ripetute di **x.equals(y)** ritornano consistentemente **true**, o **false**, purchè nessuna delle informazioni utilizzate per confrontare gli oggetti sia stata nel frattempo modificata
- Per ogni riferimento **x**, non **null**, **x.equals(null)** deve ritornare **false**

Il metodo **hashCode**

- Quando si sovrascrive il metodo **equals** bisogna sovrascrivere anche il metodo **public int hashCode()**
- Il metodo **hashCode** ritorna un codice hash da utilizzare con la classe **HashTable**
- Il metodo **hashCode** di **Object** ritorna l'indirizzo interno di un oggetto convertito in un intero
- La nozione di **hashCode** può essere dipendente dall'applicazione

Il metodo **hashCode**

Il metodo **hashCode** deve essere realizzato in modo consistente con **equals** in accordo alle seguenti regole:

- Se viene invocato ripetutatmente nell' ambito dell'esecuzone di una applicazione, il metodo deve ritornare sempre lo stesso valore intero purchè nessuana delle informazioni utilizzata da **equals** sia stata modificata

  Questo intero può essere diverso in diverse esecuzioni (*continua*)

Il metodo **hashCode**

- Se due oggetti sono uguali secondo **equals**, allora i metodi **hashCode** di ciascuno di essi devono produrre lo stesso risultato intero
- Non è richiesto, ma è consigliato, che i metodi **hashCode** di due oggetti diversi secondo **equals** producano valori diversi

Il metodo hashCode per BankAccount

```
public class BankAccount {
    private double balance;
    public int hashCode() {
      return (int)balance;
    }
    // gli altri metodi
}
```

Il metodo **getClass**

Il metodo **public final Class getClass()** ritorna un oggetto di classe **Class**, una rappresentazione a runtime della classe dell'oggetto

Un oggetto di classe Class fornisce informazioni sulla classe tra cui:

- il nome della classe
- il nome della superclasse
- i nomi delle interfacce che la classe implementa

Il metodo getClass

Il seguente metodo scrive il nome di una classe su standard output

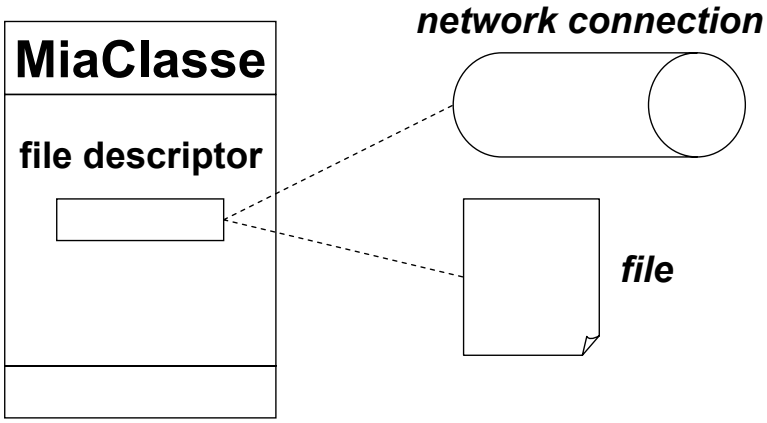
```
void PrintClassName(Object obj) {
   System.out.println("The Object's class is " +
                      obj.getClass().getName());
}
```

Il seguente metodo crea una nuova istanza della stessa classe di **obj**

```
Object createNewInstanceOf(Object obj) {
   return obj.getClass().newInstance();
}
```

Il metodo finalize

Un oggetto, oltre alla memoria può utilizzare altre risorse...



...quando il GC raccoglie l'oggetto, tali risorse devono essere rilasciate (*cleanup*)

Il metodo **finalize**

- Il processo di finalizzazione
  - quando un oggetto viene raccolto, GC invoca il metodo finalize per eseguire il cleanup
- Il cleanup può essere dipendente dall'applicazione perciò il metodo **finalize** può essere sovrascritto
- Non è consigliabile utilizzare il metodo **finalize** per gestire risorse scarse

L'operatore **instanceof**

(versione semplificata)

Expression **instanceof** ***ReferenceType***

- ***Expression*** deve restituire un un valore di tipo riferimento o di tipo **null** (altrimenti compile-time error)
- ***ReferenceType*** deve denotare un tipo riferimento (altrimenti compile-time error)

L'operatore **instanceof** produce il risultato **true** se il valore dell'espressione ***Expression*** è non **null** ed il riferimento può essere "castato" al ***ReferenceType*** (senza dar luogo a **ClassCastException**) Altrimenti, il risultato è **false.**

Il metodo **clone**

Il metodo **clone** permette di costruire un oggetto a partire da uno esistente

protected Object clone()
throws CloneNotSupportedException

Il metodo **clone** costruisce e ritorna una copia di questo oggetto

Il metodo clone di Object

L'implementazione di **clone** nella classe **Object**

- controlla che l'oggetto su cui **clone** è stato invocato implementi **Cloneable** e lancia l'eccezione **CloneNotSupportedException** in caso negativo; altrimenti
- crea un nuovo oggetto dello stesso tipo dell'oggetto originale ed inizializza le variabili membro con gli stessi valori delle corrispondenti variabili dell'oggetto originale (shallow copy)

Il metodo **clone** di **Object**

```
public class Object {
    protected Object clone()
                        throws CloneNotSupportedException {
        if ( this instanceof Cloneable ) {
            // crea il nuovo oggetto del tipo effettivo di this
            // inizializza le variabili istanza copiando le variabili
            //       istanza corrispondenti dell'oggetto originale
            // ritorna un riferimento al nuovo oggetto
        }
        else
            throw new CloneNotSupportedException();
    }
    // …
}
```

Il metodo **clone** di **Object**

- La classe **Object** non implementa **Cloneable** perciò le sottoclassi di **Object** che non implementano esplicitamente **Cloneable** non sono clonabili (eccezione **CloneNotSupportedException**)
- Il metodo **Object.clone** è **protected**, perciò se una classe derivata vuole utilizzare il metodo **clone** deve ridefinirlo (tipicamente **public**)
- Una classe *clonabile* deve implementare l'interfaccia **Cloneable**

L'interfaccia **Cloneable**

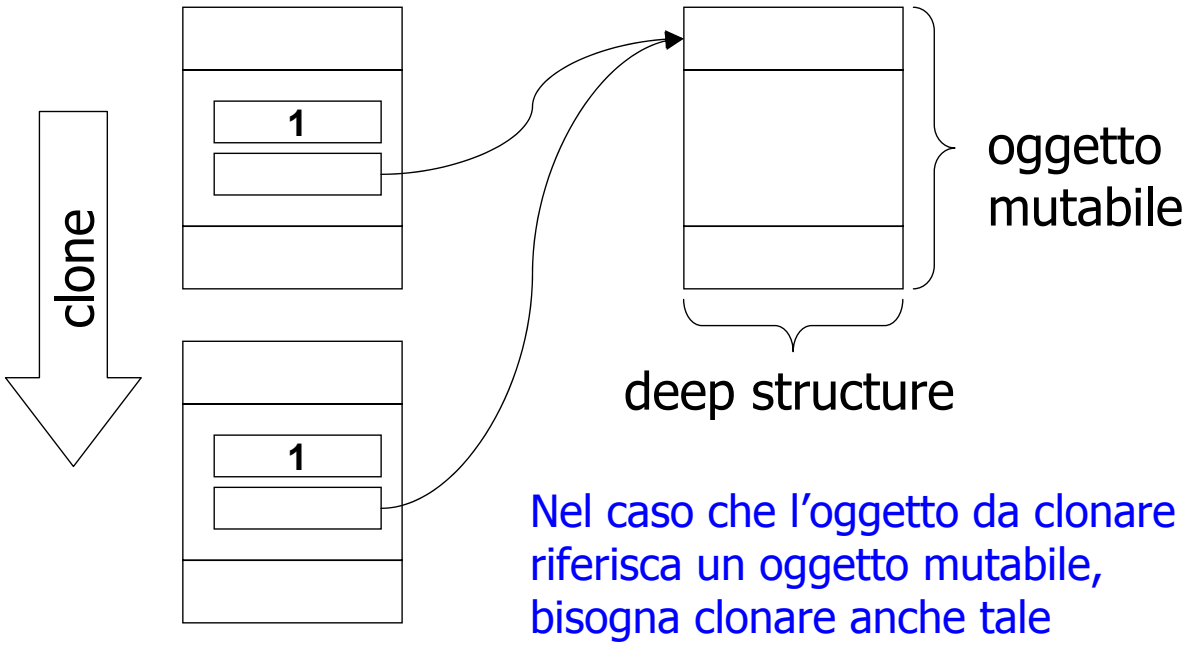
- Una classe implementa l'interfaccia **Cloneable** per indicare a **Object.clone()** che è legale fare una copia campo-per-campo degli oggetti istanza della classe
- L'invocazione del metodo **clone** di **Object** su un oggetto istanza di una classe che non implementa **Cloneable** causa la sollevazione dell'eccezione **CloneNotSupportedException**
- Per convenzione, le classi che implementano **Cloneable** devono sovrascrivere **Object.clone** (**protected**) con un metodo pubblico
- **Cloneable** non contiene il metodo **clone**. Perciò, non è possibile clonare un oggetto semplicemente in virtù del fatto che esso implementa **Cloneable**

Sovrascrittura dell metodo **clone**

```
public class CloneDemo implements Cloneable {
   private int i;          // tipo primitivo
   private String s;       // oggetto immutabile
   protected Object clone()
                     throws CloneNotSupportedException {
      try {
         CloneDemo app = (CloneDemo)super.clone();
         return app;
      } catch(CloneNotSupportedException e) {
         throw e;
      }
   }
   …
}
```

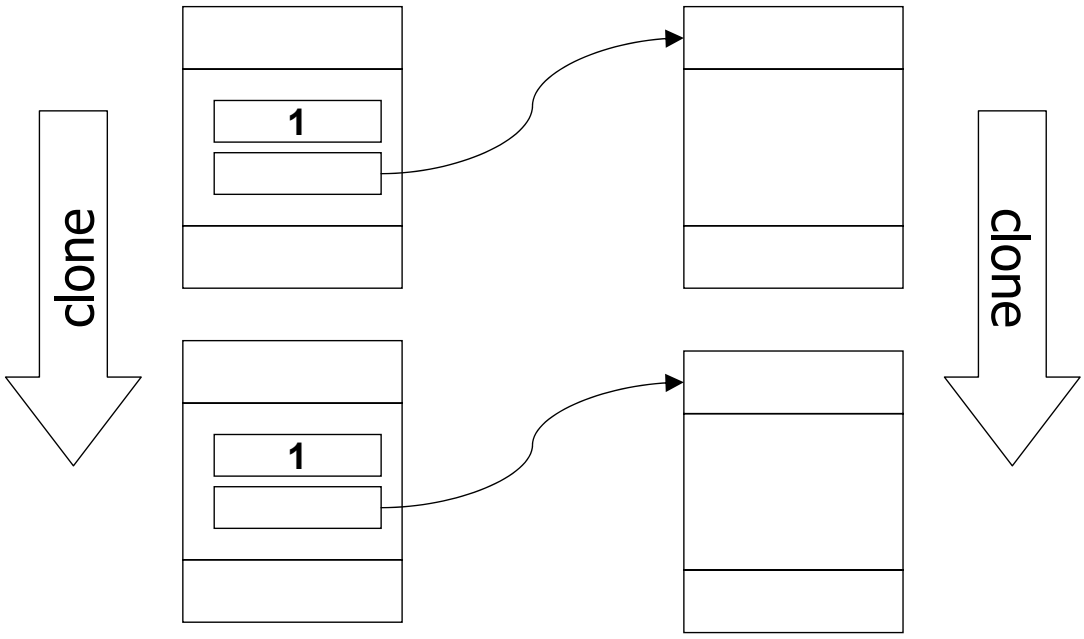
Esempio di clonazione di un oggetto che contiene solo tipi primitivi o riferimenti ad oggetti immutabili (shallow copy)

Shallow copy



Nel caso che l'oggetto da clonare riferisca un oggetto mutabile, bisogna clonare anche tale oggetto

Deep copy



Deep copy

```
public class Stack implements Cloneable {
    private Vector items;
    // campi di tipo primitivo e riferimenti immutabili
    // costruttori e metodi
    public Object clone() {
        try {
            Stack s = (Stack)super.clone();// clone the stack
            s.items = (Vector)items.clone();// clone the vector
            return s;// return the clone
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen because Stack is Cloneable
            throw new InternalError();
        }
    }
}
```

Clonare variabili istanza modificabili

```
public class Customer {
    public Customer(String UnNome) {
        nome = unNome;
        conto = new BankAccount();
    }
    public String getName() {
        return nome;
    }
    public BankAccount getAccount() {
        return conto;
    }
    // altri metodi
    private String nome;
    private BankAccount conto;
}
```


Regole empiriche per la clonazione

- Una classe deve clonare tutti i riferimenti ad oggetti mutabili che distribuisce
- Una classe deve clonare i riferimenti ad oggetti mutabili che riceve

clonare variabili istanza modificabili

```
public class Customer {
    public Customer(String unNome, BankAccount unConto) {
        nome = unNome;
        conto = (BankAccount)unConto.clone();
    }
    public BankAccount getAccount() {
        return (BankAccount)conto.clone();
    }
    // altri membri
}

public class BankAccount implements Cloneable {
    public Object clone() { /* implementazione di clone */}
    // altri membri
}
```


Ricapitolando

- L'interfaccia **Cloneable** non definisce alcun metodo: indica solo che il programmatore ha capito il processo di clonazione
- Il metodo **clone** è ridefinito **public** in modo che gli oggetti possano essere clonati ovunque
- Chi utilizza il metodo **Object.clone** deve catturare l'eccezione

Regole di progettazione di **clone**

- Per ogni oggetto **x**, l'espressione **x.clone() != x** DEVE essere vera
- Per ogni oggetto **x**, l'espressione **x.clone().getClass() == x.getClass()** DEVE essere vera
- Per ogni oggetto **x**, l'espressione **x.clone().equals(x)** PUÒ essere vera

Convenzione implementativa

- Per convenzione il metodo **clone**
  - non deve utilizzare **new** per costruire il clone
  - non deve chiamare costruttori, ma
  - deve utilizzare **super.clone** per costruire il clone
- Se una classe e le sue superclassi rispettano questa convenzione allora **x.clone().getClass() == x.getClass()** è vera