Clonazione

- Importante esempio di eccezione controllata è **`CloneNotSupportedException`** che può essere lanciata dal metodo nativo **`clone()`** della classe **`Object`**.
- La dichiarazione di **`clone()`** è la seguente:

```
protected native Object clone()
    throws CloneNotSupportedException
```

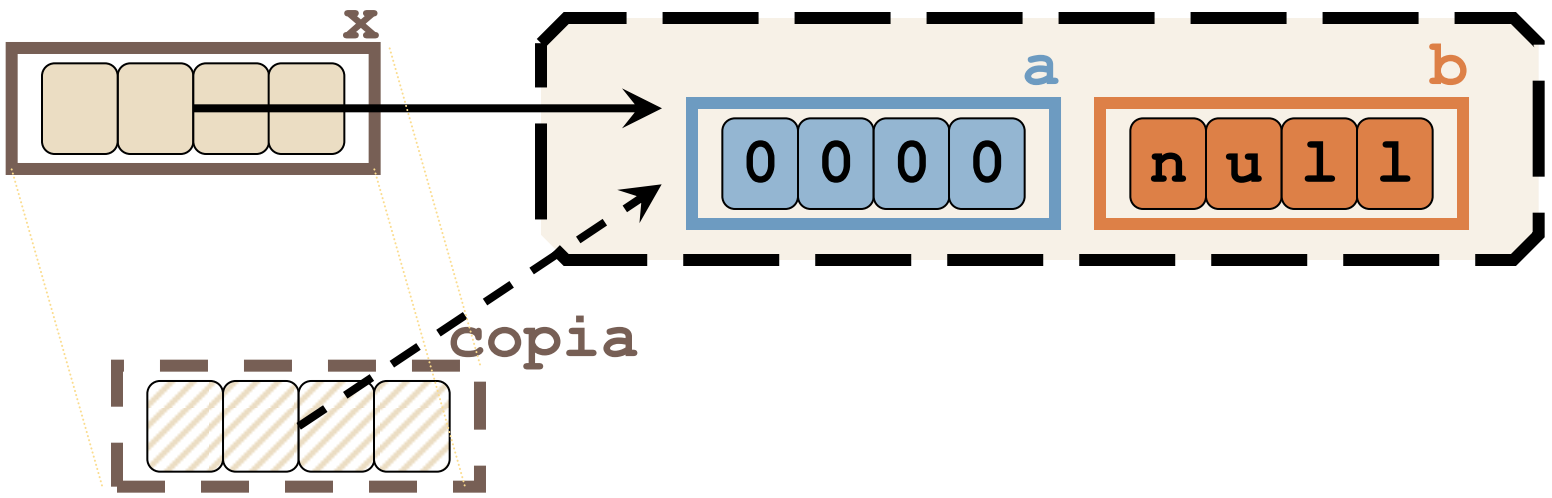
- Lo scopo del metodo è quello di restituire un nuovo oggetto con lo stesso stato di **`this`**

Clonazione

- Del metodo **`clone()`** serve sapere che:
  - ritorna un riferimento a un oggetto **`Object`**
  - richiede che l'oggetto da clonare sia un'istanza dell'interfaccia **`Clonable`** (= clonabile) altrimenti lancia una **`CloneNotSupportedException`**
- L'interfaccia **`Clonable`** in realtà **è vuota**.
  - È un'interfaccia marker: serve solo a dirci che un oggetto è clonabile (lo è se la implementa). Poiché **`CloneNotSupportedException`** è controllata, possiamo usarla per fare le verifiche del caso.

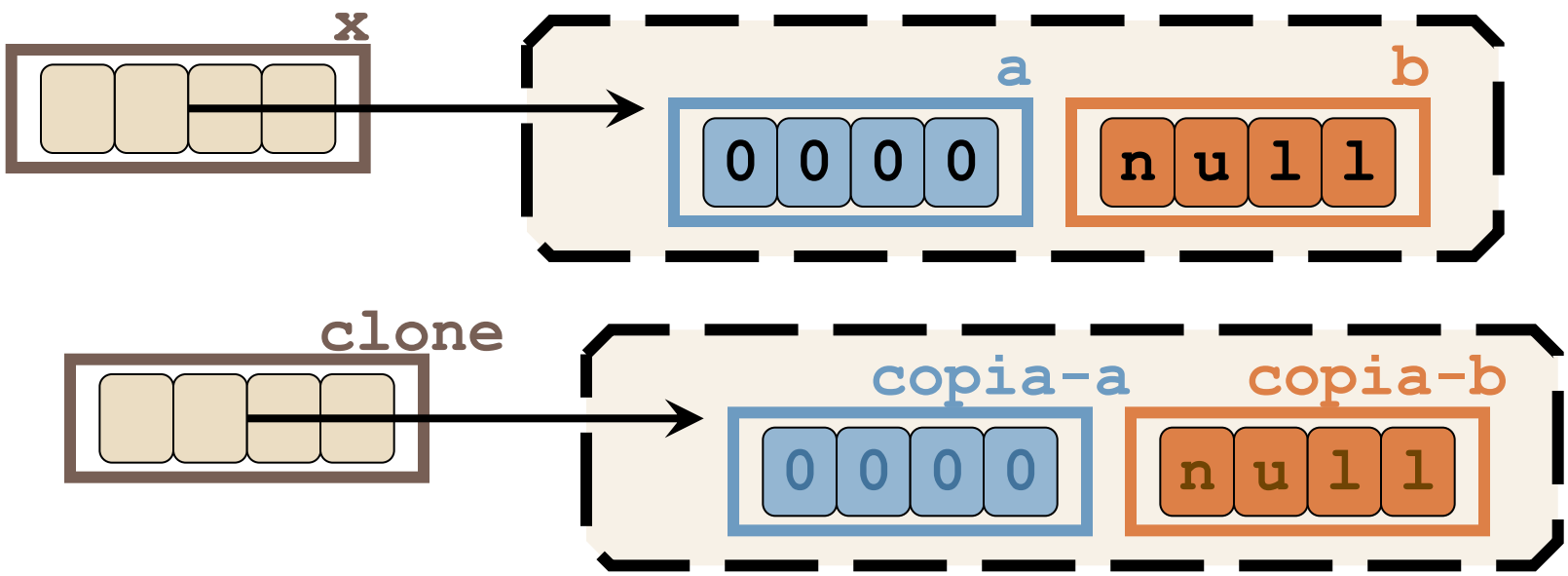
Clonazione

- La clonazione è utile perché la semplice copia di un oggetto porterebbe a copiare solo il riferimento, lasciando condivisi i valori.



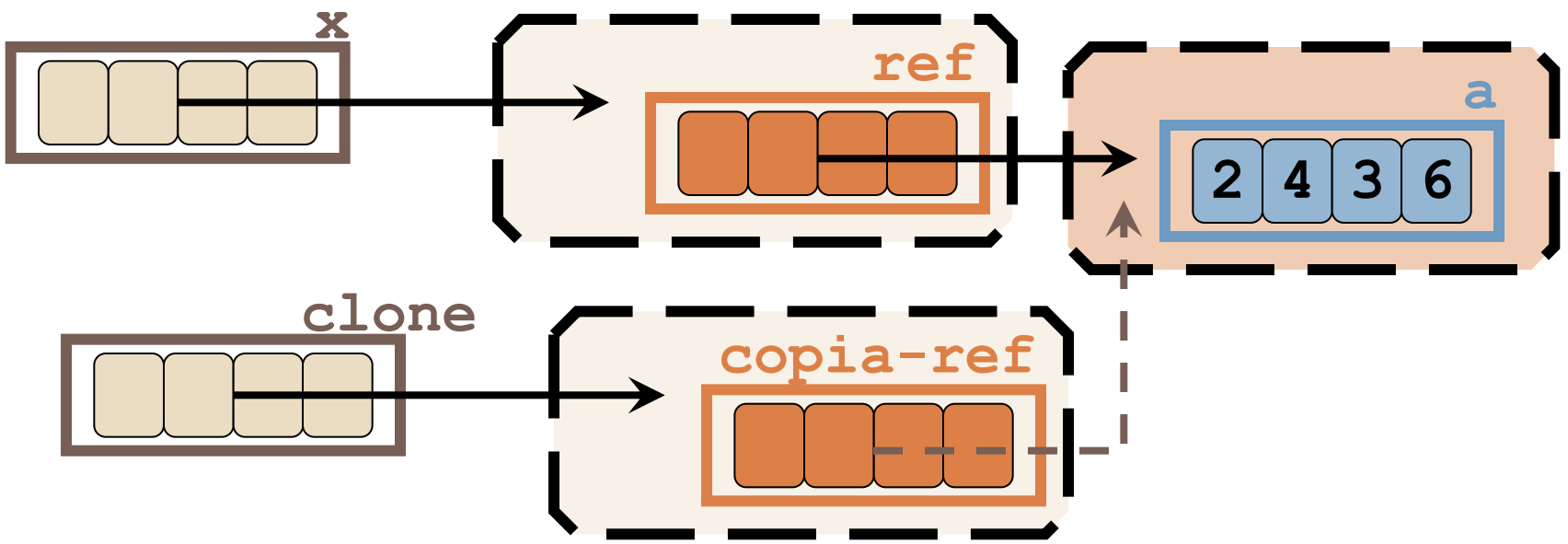
Clonazione

- Occorre invece creare un nuovo oggetto (clone) che però sia identico **internamente** a quello che viene clonato.



Clonazione

- La clonazione non è però immediata quando ci sono più livelli di riferimento. A un certo punto, abbiamo ancora condivisione, non altri oggetti.



Clonazione

- Si parla di **clonazione superficiale** (*shallow cloning*) quando, dopo un primo livello di clonazione, ulteriori livelli vengono **condivisi**.
- L'opposto è la clonazione profonda (*deep cloning*) in cui viene tutto clonato.
- **`clone()`** nativo fa una clonazione superficiale
  - intanto perché deve operare su generici **`Object`**
  - poi perché la clonazione profonda è pericolosa (può anche entrare in circoli viziosi infiniti)

Clonazione

- Generalmente il **`clone()`** nativo va bene (ma è **`protected`**). Va riscritto perché sia pubblico e anche per andare più in profondità, se serve:

```
public class MyClass implements Cloneable
{ ...
  public Object clone()
  { try
    { Object copia = super.clone();
      /* altre istruzioni per copia profonda */
      return copia;
    } catch (CloneNotSupportedException e) {
      throw new InternalError();
    }
  }
}
```

riscritto come **`public`**

(non dovrebbe succedere)

Clonazione

- Attenzione perché **`clone()`** ritorna un **`Object`**. Va quindi fatto un cast nel metodo chiamante.

```
MyClass a = new MyClass(parametri);
MyClass b = (MyClass) a.clone();
```

  - Si può evitare se si cambia il tipo di ritorno nell'override (consentito solo da Java 5 in poi)

```
public class MyClass implements Cloneable
{ ...
  public MyClass clone()
  { try
    { MyClass copia = (MyClass) super.clone();
      /* altre istruzioni per copia profonda */
      return copia;
    } catch (CloneNotSupportedException e) {...}
  }
}
```

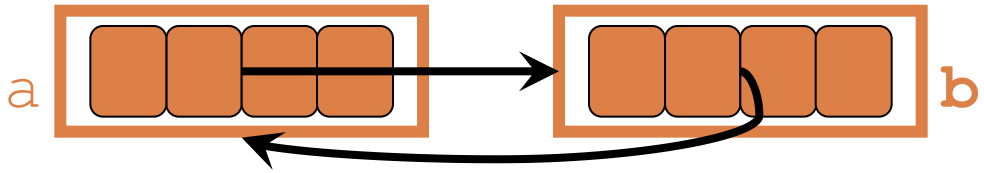
Clonazione

- Facendo così, non c'è obbligo di annunciare ulteriormente la **`throws`** dell'eccezione **`CloneNotSupportedException`**
  - Questo se si prevede che **`MyClass`** e la sua clonazione non vengano ulteriormente estese, altrimenti è bene propagare anche la **`throws`**
- Così invece si propaga semplicemente qualcosa di non controllato (**`InternalError`**)
  - ..il quale però capita soltanto se ci scordiamo di implementare l'interfaccia **`Clonable`**.

Pericoli della clonazione

- La clonazione può causare effetti collaterali se clono un oggetto con un campo riferimento.
- Se un oggetto ha riferimenti ciclici, quando si deve arrestare la clonazione e fare solo copie?

```
class Nodo { public Nodo next; ...}
Nodo a,b; a.next = b; b.next = a;
```



- Quindi in Java si usa una **doppia protezione**: il **`clone()`** è **`protected`** e richiede anche di implementare l'interfaccia marker **`Cloneable`**.

`throws` e override

- Un metodo prevede di poter lanciare eccezioni di un certo tipo, e lo dichiara con **`throws`**. Poi una sottoclasse sovrascrive quel metodo (ossia fa override di quello ereditato). Il nuovo metodo deve avere una **`throws`**?
  - Può averla oppure no. Se il nuovo metodo non lancerà mai quell'eccezione, semplicemente omette **`throws`**.
  - Se invece viene inserita una clausola **`throws`**, questa deve lanciare **lo stesso tipo, o un tipo derivato,** del metodo originale.

`throws` e polimorfismo

- In virtù del polimorfismo, le eccezioni che vengono lanciate possono essere del tipo dichiarato da **`throws`**, o anche di un sottotipo.
- Quindi un metodo che vuole lanciare più tipi di eccezioni che sono tutti sottoclassi di una stessa superclasse, può annunciare solo quest'ultima nella **`throws`**.
- È bene che la **`throws`** sia la più specifica possibile (non mettere **`throws Exception`** e basta, o andrà controllato tutto).

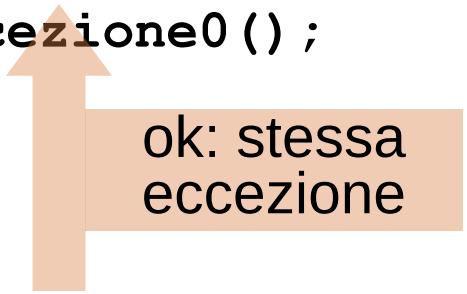
`throws` e polimorfismo

- Ad esempio

```
class Eccezione0 extends Exception {...}
class SottoEcc1 extends Eccezione0 {...}
class SottoEcc2 extends Eccezione0 {...}

class Item
{ public void usa() throws Eccezione0
   { if (inesperto) throw new Eccezione0();
   }
}

class Pergamena extends Item
{ public void usa() throws Eccezione0
   { if (inesperto) throw new Eccezione0();
   }
}
```



`throws` e polimorfismo

- Ad esempio

```
class Eccezione0 extends Exception {...}
class SottoEcc1 extends Eccezione0 {...}
class SottoEcc2 extends Eccezione0 {...}

class Item
{ public void usa() throws Eccezione0
  { if (inesperto) throw new Eccezione0();
  }
}

class Arma extends Item
{ public void usa() throws SottoEcc1, SottoEcc2
  { if (pacifista) throw new SottoEcc1();
    if (karateka) throw new SottoEcc2();
  }
```

ok: istanze più specifiche

`throws` e polimorfismo

- Ad esempio

```
class Eccezione0 extends Exception {...}
class SottoEcc1 extends Eccezione0 {...}
class SottoEcc2 extends Eccezione0 {...}

class Item
{ public void usa() throws Eccezione0
   { if (inesperto) throw new Eccezione0();
   }
}

class Pozione extends Item
{ public void usa() throws Eccezione0
   { throw Eccezione0(); throw new SottoEcc1;
     ... }
}
```

ok: non lancia le eccezioni

`throws` e polimorfismo

- Ad esempio

```
class Eccezione0 extends Exception {...}
class SottoEcc1 extends Eccezione0 {...}
class SottoEcc2 extends Eccezione0 {...}

class Arma extends Item
{ public void usa() throws SottoEcc1, SottoEcc2
  { if (pacifista) throw new SottoEcc1();
    if (karateka) throw new SottoEcc2();
  }

class Spadone extends Arma
{ public void usa() throws Eccezione0
  { if (chierico) throw new Eccezione0();
    ... }
}
```

NO!
superclasse

`throws` e polimorfismo

```
Item ogg1 = new Item(); Item ogg2 = new Arma();
Item ogg3 = new Pozione();

class Item
{ public void usa() throws Eccezione0
  { if (inesperto) throw new Eccezione0();
    ...
class Arma extends Item
{ public void usa() throws SottoEcc1, SottoEcc2
  { if (pacifista) throw new SottoEcc1();
    ...
class Pozione extends Item
{ public void usa() {...}
}
```

è un sistema coerente?

`throws` e polimorfismo

```
Item ogg1 = new Item(); Item ogg2 = new Arma();
Item ogg3 = new Pozione()

try { ogg1.usa() }
catch (Eccezione0 e) {  //contromisura... }

try { ogg2.usa() }
catch (Eccezione0 e) {  //contromisura... }

try { ogg3.usa() }
catch (Eccezione0 e) {  //contromisura... }
```

può sollevare l'`Eccezione0`

può sollevare `SottoEcc1,2`

in realtà non solleva nulla

- Per il compilatore sono tutti oggetti di tipo statico **`Item`** e possono sollevare **`Eccezione0`**; in ogni caso va bene provare a intercettare istanze dell'eccezione controllata **`Eccezione0`** e gestirle con **`try-catch`**