



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE

# Progettazione di eccezioni



# Definizione di un'eccezione

- Ora che conosciamo l'ereditarietà, invece di lanciare una delle eccezioni che troviamo nella libreria standard, **possiamo lanciare un'eccezione che abbia un nome inventato da noi:**

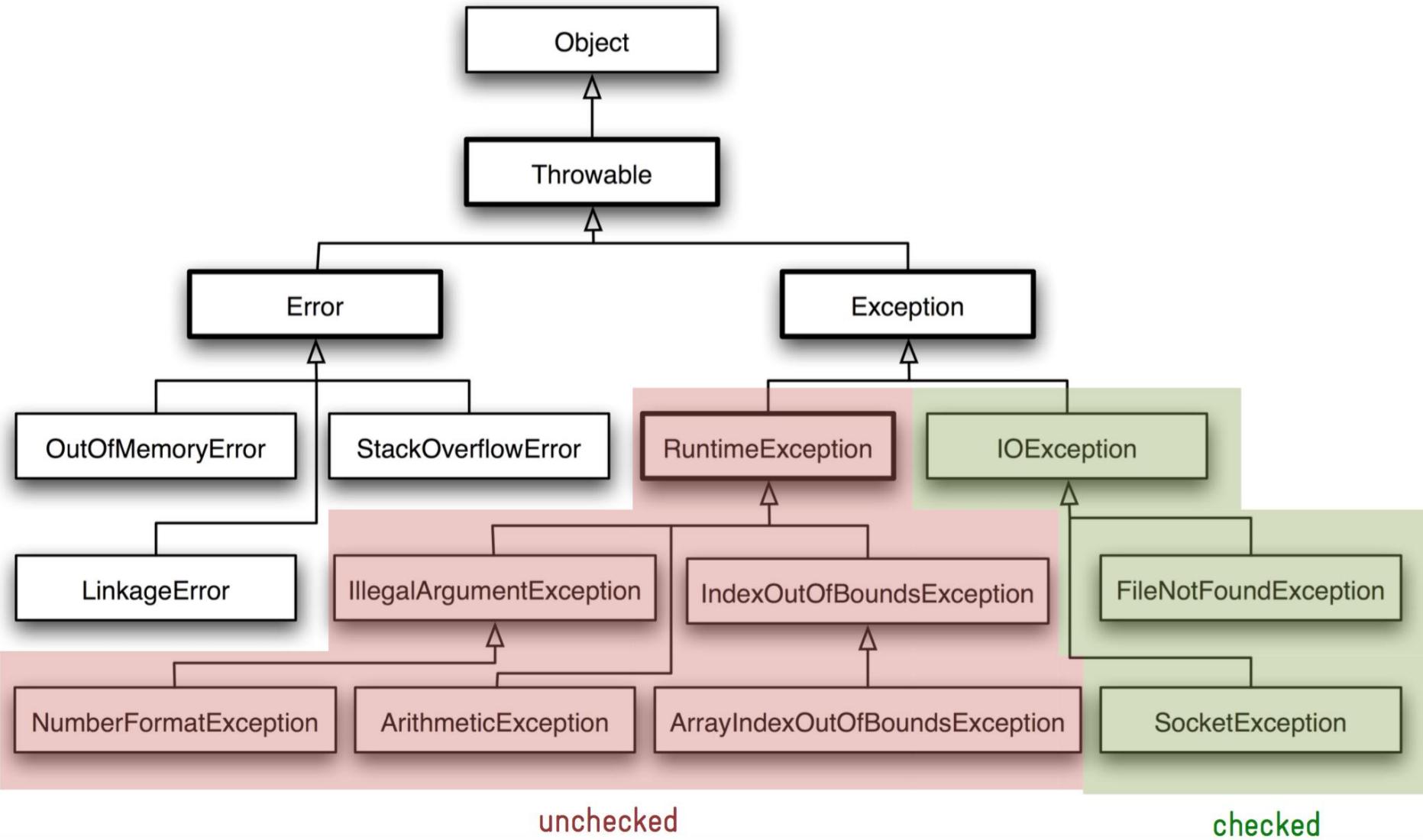
```
public class MyException extends RuntimeException { }
```

- La classe è **vuota**, a cosa serve?
  - serve a definire un nuovo tipo di dato (un'eccezione) che ha le stesse identiche caratteristiche della superclasse da cui viene derivato, ma di cui interessa porre in evidenza il **nome**, che contiene l'informazione di identificazione dell'errore
- Con le eccezioni si fa spesso così
  - in realtà la classe non è vuota, perché contiene tutto ciò che eredita dalla sua superclasse (posso ovviamente sovrascrivere i costruttori/metodi)
- Rimane una domanda: **quale superclasse usiamo?**

# Diversi tipi di eccezioni - riassunto

- In Java esistono diversi tipi di eccezioni (cioè diverse classi di cui le eccezioni sono esemplari) e sono tutte sottoclassi di **Throwable**
  - eccezioni che sono sottoclassi di **Error**
  - eccezioni che sono sottoclassi di **Exception**
    - un sottoinsieme di queste sono sottoclassi di **RuntimeException**
- L'enunciato **throw** consente il lancio di qualsiasi oggetto che sia esemplare di **Throwable** o di una sua sottoclasse (cioè un oggetto che possa essere assegnato, mediante up-casting, a una variabile di tipo **Throwable** )
- La gestione (con **try/catch**) delle sottoclassi di **Error** e di **RuntimeException** è **facoltativa**
  - se non vengono gestite e vengono lanciate, provocano la terminazione del programma
- La gestione delle altre eccezioni è **obbligatoria**

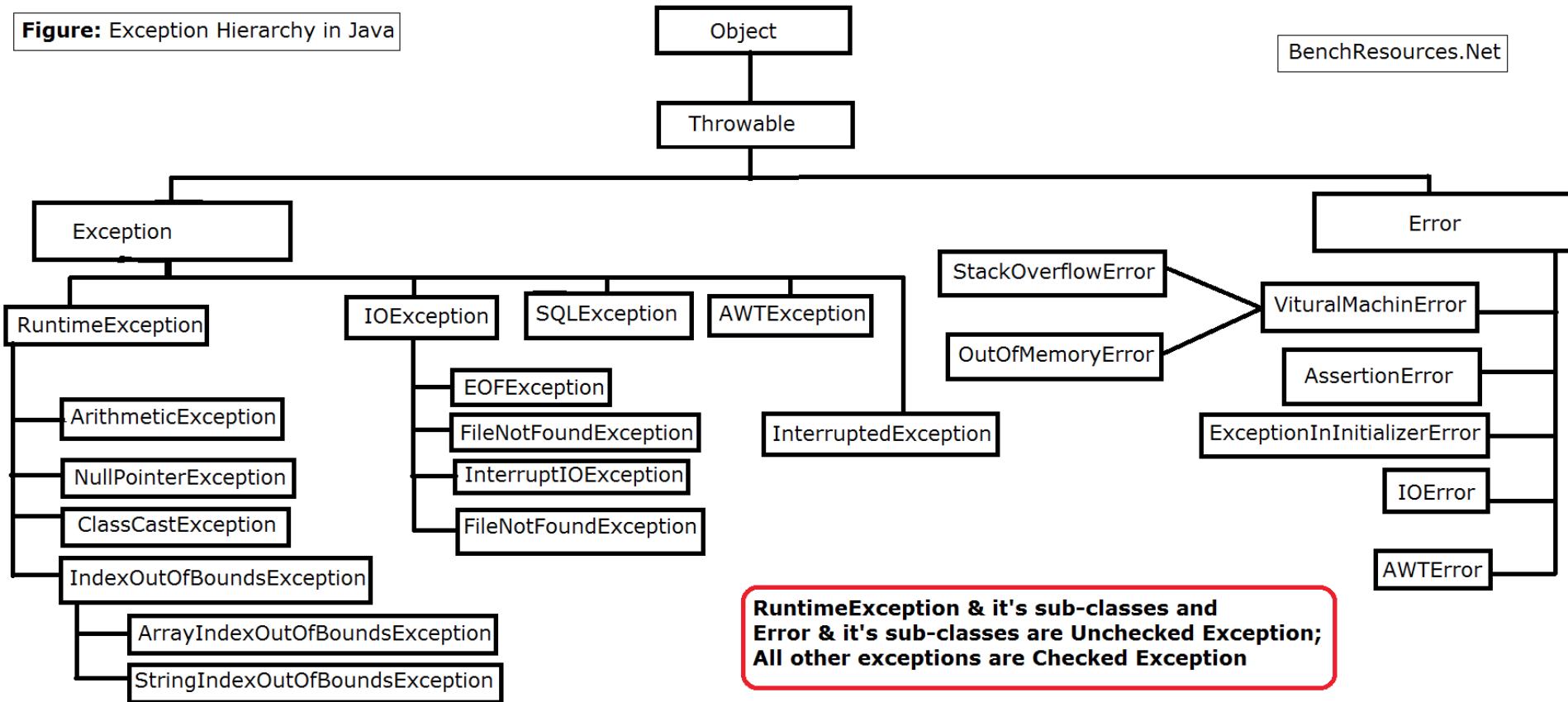
# Gerarchia delle eccezioni



# Un altro schema per vedere piu' eccezioni fornite dalla libreria di Java

Figure: Exception Hierarchy in Java

BenchResources.Net





# Diversi tipi di eccezioni

- Non si dovrebbero definire eccezioni come sottoclassi di **Error**
  - Sono riservate alla JVM e, in generale, corrispondono a eventi che non prevedono “contromisure”
  - Es. **StackOverflowError**, **OutOfMemoryError**
- Quindi, per definire una nostra eccezione, possiamo scegliere di usare come superclasse
  - **Exception** (o una sua sottoclasse che NON sia anche sottoclasse di **RuntimeException**)
    - Sarà un'eccezione **a gestione obbligatoria**
  - **RuntimeException** (o una sua sottoclasse)
    - Sarà un'eccezione **a gestione non obbligatoria**



# Filosofia...

- Si dovrebbero lanciare eccezioni
  - A gestione non obbligatoria
    - Quando si verifica **una situazione di errore provocata da un errore del programmatore**
      - Parametri di metodi in violazione di pre-condizioni
  - A gestione obbligatoria
    - Quando si verifica **una situazione di errore non dipendente dal programmatore**
      - Interazioni con l'utente
      - Interazioni con la rete
      - Interazioni con il sistema operativo (filesystem...)



# Lancio di eccezioni

- Abbiamo già visto che un metodo può lanciare eccezioni a controllo non obbligatorio
  - eccezioni che derivano da **RuntimeException**, come **IllegalArgumentException**
- Se, invece, un metodo vuole lanciare eccezioni a controllo obbligatorio
  - nella **firma** del metodo deve essere dichiarato l'elenco delle possibili eccezioni a controllo obbligatorio lanciate nel metodo
    - si usa la clausola **throws**

```
... method(...) throws AnException
```



# Lancio di eccezioni

- La clausola **throws** deve essere presente nella firma del metodo
  - nella classe che definisce concretamente il metodo

```
... method(...) throws AnException  
{ ... }
```

- in eventuali interfacce che dichiarano il metodo (\*)

```
... method(...) throws AnException;
```

- Se il metodo può lanciare più eccezioni, vanno elencate dopo **throws**, separandole con virgolette

```
... method(...) throws Exception1, Exception2;
```

- si possono elencare anche eccezioni a controllo non obbligatorio, a puro scopo documentale, ma questo NON le rende a controllo obbligatorio



# Lancio di eccezioni

- Attenzione: **throws** non lancia eccezioni, ma dichiara che il metodo può lanciarle
  - è una segnalazione al compilatore, che così obbliga all'uso di un blocco **try/catch** nei metodi che invocano tale metodo

```
... method(...) throws AnException { ... }
```

```
try
{   method(...);
}
catch (AnException e)
{
    ...
}
```

Nel metodo  
invocante



# Lancio di eccezioni

- In un metodo, esiste la possibilità di **non** gestire un'eccezione a gestione obbligatoria
  - Occorre delegare la sua gestione al metodo invocante, inserendo tale eccezione nella clausola **throws** del metodo, che non la lancia, ma la “rilancia” se essa viene lanciata

```
... method(...) throws AnException { ... }
```

```
... anotherMethod(...) throws AnException
{   method(...);
}
```

```
... main(...)
{   // prima o poi qualcuno la deve catturare
    try { anotherMethod(...); }
    catch (AnException e) { ... }
}
```



# Lancio di eccezioni

- Questa strategia, anche se sintatticamente corretta, non è semanticamente accettabile nel **main**
  - A chi si rimanda la gestione? Direttamente alla JVM, che è come dire “non c’è gestione”



# Metodi delle eccezioni (nel catch)

```
try { ... }  
catch (IllegalArgumentException e) { ... }
```

- A cosa serve la variabile **e** di tipo “riferimento a una specifica eccezione” che usiamo nella clausola **catch**?
  - Le eccezioni sono oggetti, esemplari di una classe che ha dei metodi, invocabili con quella variabile
  - Ad esempio, il metodo **printStackTrace**, che richiede come parametro un esemplare di **PrintStream** (come **System.out**) o di **PrintWriter**, può essere invocato per scrivere un messaggio identico a quello che verrebbe scritto su **System.out** dalla JVM nel caso in cui l’eccezione provocasse la terminazione del programma, assieme allo stack di chiamate



# Metodi delle eccezioni (nel catch)

```
try { ... }  
catch (IllegalArgumentException e) { ... }
```

- **getMessage()**
  - Messaggio contenente i dettagli dell'eccezione
- **toString()**
  - Breve descrizione dell'oggetto Throwable e dettagli dell'eccezione se presenti



# Riassumendo...

## □ Ora sappiamo

- programmare nei nostri metodi il lancio di eccezioni appartenenti alla libreria standard (**throw new ...**)
- gestire eccezioni (**try/catch/finally**) o **try-with-resources**
- lasciare al chiamante la gestione di eccezioni sollevate nei nostri metodi (**throws** nell'intestazione del metodo)
- Creare una nostra eccezione (es. **extends RunTimeException**, lasciando il corpo della sottoclasse **vuoto**)



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE

# **Ordinamento e ricerca di oggetti**



# Ordinamento di oggetti

- Abbiamo visto algoritmi di ordinamento efficienti e ne abbiamo verificato il funzionamento con array di numeri, ma spesso si pone il problema **di ordinare dati più complessi**
  - ad esempio, ordinare stringhe
  - in generale, ordinare oggetti
- Vedremo ora che si possono usare gli stessi algoritmi, a patto che gli oggetti da ordinare siano tra loro **confrontabili**



# Ordinamento di oggetti

- Per ordinare numeri è necessario effettuare confronti
- La stessa affermazione è vera per oggetti
  - ▣ **per ordinare oggetti è necessario effettuare confronti tra loro**
- C'è però una differenza
  - ▣ confrontare numeri ha un significato ben definito dalla matematica (struttura ordinale)
  - ▣ **confrontare oggetti ha un significato che dipende dal tipo di oggetto**, e a volte può non avere significato alcuno...



# Ordinamento di oggetti

- **La classe che definisce l'oggetto deve anche definire il significato del confronto**
- Consideriamo la classe String
  - essa definisce il metodo compareTo che attribuisce un significato ben preciso all'ordinamento tra stringhe
    - l'ordinamento lessicografico
- Possiamo quindi riscrivere, ad esempio, il metodo **selectionSort** per ordinare stringhe invece di ordinare numeri, **senza cambiare l'algoritmo**

```
public class ArrayUtils
{  public static void selectionSort(String[] a)
   {  for (int i = 0; i < a.length - 1; i++)
      {  int minPos = findMinPos(a, i);
          if (minPos != i)
              swap(a, minPos, i);
      }
   }
private static void swap(String[] a, int i, int j)
{  String temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
private static int findMinPos(String[] a, int from)
{  int pos = from;
   for (int i = from + 1; i < a.length; i++)
       if (a[i].compareTo(a[pos]) < 0)
           // per gli int era: if (a[i] < a[pos])...
           pos = i;
   return pos;
}
...
}
```



# Ordinamento di oggetti

- Allo stesso modo si possono riscrivere tutti i metodi di ordinamento e ricerca visti per i numeri interi ed usarli per le stringhe. **Ma come fare per altre classi?**
- Possiamo ordinare oggetti di tipo BankAccount in ordine, ad esempio, di saldo crescente?
  - bisogna definire un metodo **compareTo()** nella classe **BankAccount**
  - bisogna **riscrivere i metodi** perché accettino come parametro un array di BankAccount

```
public class ArrayAlgorithms
{
    public static void selectionSort(BankAccount[] a)
    {   for (int i = 0; i < a.length - 1; i++)
        {   int minPos = findMinPos(a, i);
            if (minPos != i)
                swap(a, minPos, i);
        }
    }

    private static void swap(BankAccount[] a, int i, int j)
    {   BankAccount temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    private static int findMinPos(BankAccount[] a, int from)
    {   int pos = from;
        for (int i = from + 1; i < a.length; i++)
            if (a[i].compareTo(a[pos]) < 0)
                pos = i;
        return pos;
    }

    ...
}
```



# Realizzare una proprietà astratta

- In molti casi è desiderabile che più classi condividano delle proprietà comuni
  - Ad esempio che gli oggetti di ciascuna classe siano confrontabili tra loro
- Questo si esprime in Java definendo un'interfaccia (**interface**) e dicendo che una classe realizza (**implements**) l'interfaccia



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE

# Realizzare una proprietà astratta: Interfacce



# Interfacce

- Per **definire un comportamento astratto** in Java si usa la definizione di **interfaccia**, che deve essere **realizzata** (=implementata) da una classe che dichiari di avere tale comportamento
- **Una classe** può estendere una sola classe ma **può contemporaneamente realizzare una o più interfacce**
- Un'interfaccia definisce solo la firma dei metodi che la classe dovrà poi realizzare



# Ad esempio, per l'ordinamento...

- Quello che deve fare una classe perché i suoi esemplari possano essere ordinati è
  - definire un metodo adatto a confrontare esemplari della classe, con lo stesso comportamento di **compareTo**
- Gli oggetti della classe diventano **confrontabili**
  - gli algoritmi di ordinamento e ricerca **non hanno bisogno di conoscere alcun particolare degli oggetti**
  - è sufficiente che gli oggetti siano tra loro **confrontabili**

# Realizzare una proprietà astratta

```
public interface Comparable  
{ int compareTo(Object other);  
}
```

- La definizione di un'interfaccia è simile alla definizione di una classe
  - si usa la parola chiave **interface** al posto di **class**
- Un'interfaccia può essere vista come una “**classe ridotta**”, perché
  - non può avere costruttori
  - non può avere variabili di esemplare
  - **contiene soltanto le firme di uno o più metodi non statici**, ma non può definirne il codice
    - i metodi sono **implicitamente public**

# Realizzare una proprieta' astratta

- Se una classe dichiara di realizzare concretamente un comportamento astratto definito in un'interfaccia, **DEVE implementare l'interfaccia**
- oltre alla dichiarazione, **DEVE definire i metodi specificati nell'interfaccia, con la stessa firma**

```
public class BankAccount implements Comparable
{
    ...
    public int compareTo(Object other)
    { // notare che viene sempre ricevuto un Object
        BankAccount acct = (BankAccount)other;
        if (balance < acct.balance) return -1;
        if (balance > acct.balance) return 1;
        return 0;
    }
}
```



# Conversione da classe a interfaccia

- Non è possibile costruire oggetti da un'interfaccia

```
new Comparable(); // ERRORE DI SINTASSI
```

- È invece possibile definire riferimenti ad oggetti che realizzano un'interfaccia

```
Comparable c = new BankAccount(10);
```

- Queste conversioni tra un oggetto ed un riferimento ad una delle interfacce che sono realizzate dalla sua classe sono automatiche (anche per gli array)

```
... implements interface1, interface2
```

- *Una classe estende sempre una sola altra classe, mentre può realizzare più interfacce*

# Conversione da classe a interfaccia

```
Comparable c = new BankAccount(10); // corretto;
```

- Usando la variabile c non sappiamo esattamente quale è il tipo dell'oggetto a cui essa si riferisce
  - **Non possiamo** utilizzare tutti i metodi di **BankAccount**

```
double saldo = c.getBalance(); // errore in compilazione
```

- Però **siamo sicuri** che quell'oggetto ha un metodo **compareTo**

```
Comparable d = new BankAccount(20);  
if ( c.compareTo(d) > 0 ) //corretto
```



# Conversione da interfaccia a classe

- A volte può essere necessario convertire un riferimento il cui tipo è un interfaccia in un riferimento il cui tipo è una classe che implementa l'interfaccia

```
Comparable c = new BankAccount(10);  
double saldo = c.getBalance(); // errore, come faccio?
```

# Conversione da interfaccia a classe

- Se siamo certi che **c** punta ad un oggetto di tipo **BankAccount** possiamo creare una nuova variabile **acct** di tipo **BankAccount** e **fare un cast** di **c** su **acct**

```
Comparable c = new BankAccount(10);  
BankAccount acct = (BankAccount) c;  
double saldo = acct.getBalance(); //corretto
```

- E se ci sbagliamo? **ClassCastException** in esecuzione



# L'interfaccia Comparable

```
public interface Comparable
{ int compareTo(Object other);
}
```

- L'interfaccia **Comparable** è definita nel pacchetto **java.lang**, per cui non deve essere importata né deve essere definita
  - la classe **String**, ad esempio, realizza **Comparable**
- Come può l'interfaccia **Comparable** risolvere il nostro problema di **definire un metodo di ordinamento valido per tutte le classi?**
  - basta definire un metodo di ordinamento che ordini un array di riferimenti ad oggetti che realizzano l'interfaccia **Comparable**, indipendentemente dal tipo



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE

# **Ordinamento di oggetti: l'interfaccia Comparable**



# Java.lang.Comparable

```
public interface Comparable
{
    /**
     * Definisce l'ordine naturale
     * Confronta questo esemplare con l'esemplare specificato
     * @param other l'esemplare specificato
     * @return un intero negativo se questo esemplare precede
     *         l'esemplare specificato, zero se i due esemplari
     *         coincidono, un intero positivo altrimenti
     * @throws java.lang.ClassCastException se l'esemplare
     *         specificato non è della stessa classe di questo
     *         esemplare o di una sua sottoclasse
    */
    int compareTo(Object other);
}
```

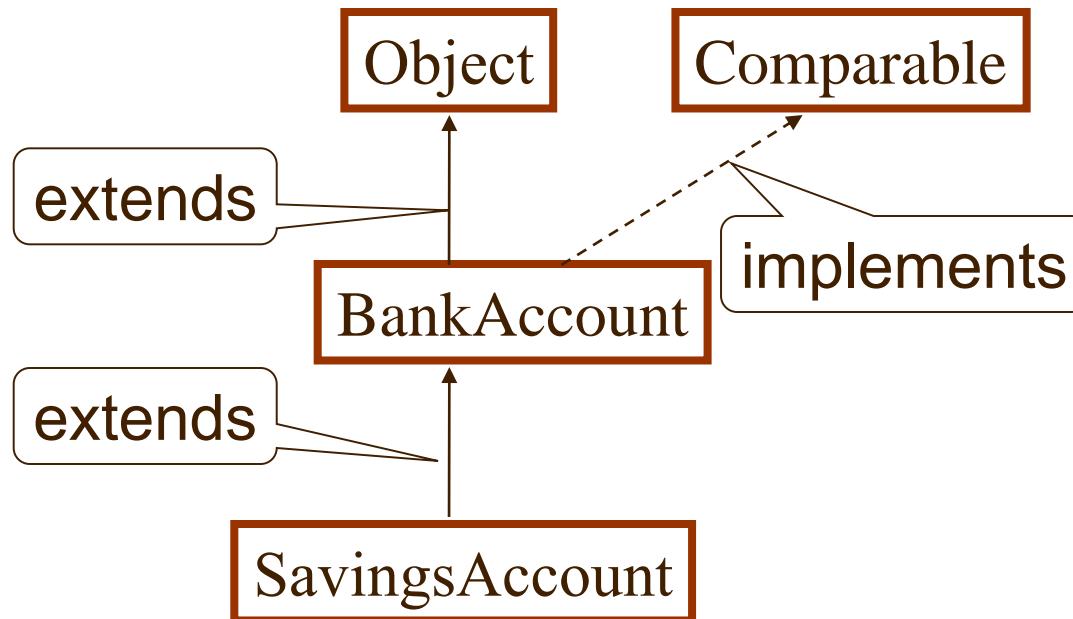


# Note su JDK

**public int compareTo (Object x);**

- A partire dal JDK 1.5.x il metodo **compareTo** usa i Generics (che noi non vediamo!) e non gli Object
  - In alcuni casi può capitare di ottenere un warning
  - La compilazione va comunque a buon fine

# Relazione tra le classi e le interfacce





# Ordinamento di oggetti

- Tutti i metodi di ordinamento e ricerca che abbiamo visto per array di numeri interi possono essere riscritti per array di oggetti Comparable, usando le seguenti “traduzioni”

```
if (a < b)  
  
if (a > b)  
  
if (a == b)  
  
if (a != b)
```



```
if (a.compareTo(b) < 0)  
  
if (a.compareTo(b) > 0)  
  
if (a.compareTo(b) == 0)  
  
if (a.compareTo(b) != 0)
```

```
public class ArrayAlgorithms
{ public static void selectionSort(Comparable[] a)
  { for (int i = 0; i < a.length - 1; i++)
    { int minPos = findMinPos(a, i);
      if (minPos != i)
        swap(a, minPos, i);
    }
  }
private static void swap(Comparable[] a, int i, int j)
{ Comparable temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
private static int findMinPos(Comparable[] a, int from)
{ int pos = from;
  for (int i = from + 1; i < a.length; i++)
    if (a[i].compareTo(a[pos]) < 0) pos = i;
  return pos;
}
...
}
```



# Ordinamento di oggetti

- Definito un algoritmo per ordinare un array di riferimenti **Comparable**, se vogliamo ordinare un array di oggetti **BankAccount** basta fare

```
BankAccount[] v = new BankAccount[10];  
// creazione dei singoli elementi dell'array  
...  
// eventuali modifiche allo stato  
// degli oggetti dell'array  
...  
ArrayAlgorithms.selectionSort(v);
```

- Dato che **BankAccount** realizza **Comparable**, l'array di riferimenti viene convertito automaticamente



# Take home message

- Le interfacce
  - permettono di definire proprietà comuni a classi diverse
  - definiscono queste proprietà in modo astratto attraverso la sola dichiarazione della firma di metodi
- E' compito delle classi che implementano l'interfaccia realizzare concretamente tali metodi secondo le loro specificità
- Possiamo scrivere algoritmi che agiscono su oggetti del tipo dell'interfaccia
  - Tali algoritmi funzioneranno con qualsiasi oggetto creato con una classe che realizzi l'interfaccia