I limiti dell'ereditarietà semplice

- L'ereditarietà semplice non permette la descrizione di numerose situazioni reali
  - Supponiamo di avere una classe Giocattolo ed una classe Automobile
  - In assenza di ereditarietà multipla non posso definire la classe AutomobileGiocattolo
- La soluzione di Java
  - Distingue tra una gerarchia di ereditarietà (semplice) ed una gerarchia di implementazione (multipla) introducendo il costrutto **interface**

Interfacce

- Un'interfaccia è come una classe che può avere solo attributi costanti e i cui metodi sono tutti pubblici e astratti
- Sintassi:

  interface <nome> {
  <lista di definizione di attributi costanti e metodi privi di corpo>
  }
- Gli attributi dichiarati in un'interfaccia sono
  - Visibili alla classe che la implementa
  - Immutabili (dichiarati come static final)

```
public interface Scalable {
    int SMALL=0, MEDIUM=1, BIG=2; //static e final
    void setScale(int size);
}
```

Esempio

```
public interface Shape {
    public String baseclass="shape";
    public void draw();
}

public class Circle implements Shape {
    public void draw(){
       System.out.println("Drawing Circle here");
    }
}

public class Main {
    public static void main(String[] args) {
       Shape circleshape = new Circle();
       circleshape.draw();
    }
}
```

Interfacce ed ereditarietà

- Una interfaccia può ereditare da una o più interfacce

```
interface <nome> extends <nome1>,..,<nomen> {
        …
}
```

La gerarchia di implementazione

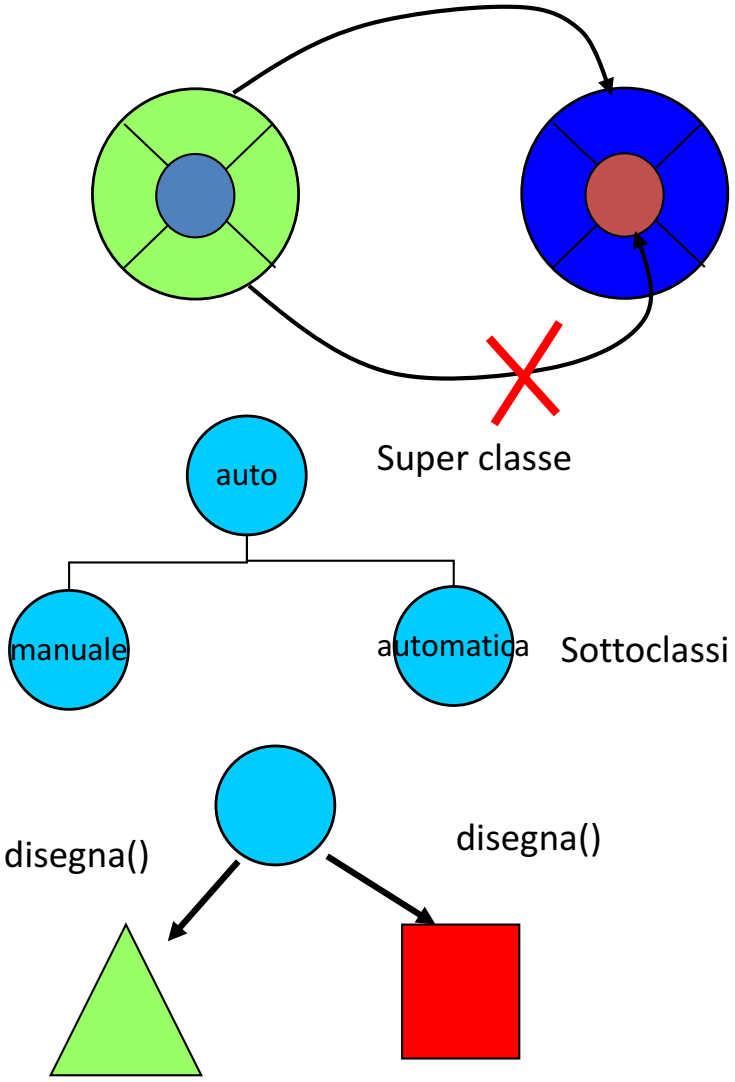
- Una classe può implementare una o più interfacce, ma estendere al più una classe
  - Se la classe non è astratta deve fornire un'implementazione per tutti i metodi presenti nelle interfacce che implementa
  - Altrimenti la classe è astratta

Classi astratte e interfacce

- Classi astratte
  - Possono avere metodi implementati e non
- Interfacce
  - Possono avere solo metodi non implementati
- Classi concrete
  - Hanno tutti i metodi implementati

I tre principi

- **Incapsulamento**
  - Gli oggetti nascondono il loro stato e parte del loro comportamento
- **Ereditarietà**
  - Ogni sottoclasse eredità tutte le proprietà della/delle superclassi
- **Polimorfismo**
  - Stessa interfaccia anche per tipi di dati diversi



Conversioni automatiche di tipo

Promozioni

- byte -> short, int, long, float, double
- short -> int, long, float, double
- int -> long, float, double
- long -> float or double
- float -> double
- char -> int, long, float, double

Conversioni forzate: casting

- È possibile forzare una conversione di tipo attraverso l'operatore di casting:
  - (<tipo>)<espressione>
- Tra tipi primitivi sono consentite le seguenti conversioni forzate (quando possibile e con perdita di informazione)
  - short -> byte, char
  - char -> byte, short
  - int -> byte, short, char
  - long -> byte, short, char, int
  - float -> byte, short, char, int, long
  - double -> byte, short, char, int, long, float
  - byte -> char

Casting in generale

- È possibile forzare esplicitamente la conversione da un tipo riferimento T ad un sottotipo T1 purché:
  - Il tipo dinamico dell'espressione che convertiamo sia un sottotipo di T1

```
Animale a = …;
Gatto mao = …; // eredita da Animale
a = mao; // OK assegnazione polimorfica
mao = (Gatto)a; // corretto (casting) perche' a e' un gatto
```

instanceof

- Per evitare errori runtime e stabilire qual è il tipo dinamico di un oggetto si può usare l'operatore **instanceof**

```
Object a;
int i = System.in.read();
if (i>0) a = new String();
else a = new Integer(5);
if (a instanceof String) return a.equals("abcd")
```

instanceof e equals

```
public class Data {
   private int giorno;
   private int mese;
   private int anno;

   public int leggiGiorno(){…}
   public boolean equals(Object o) {
    if (!(o instanceof Data)) return false;
    Data d= (Data) o;
    return (giorno==d.giorno &&
        mese == d.mese && anno == d.anno);
   }
}
```

ArrayList

- Gli arrayList sono contenitori “estendibili” e “accorciabili” dinamicamente
  - Prima di Java 5 potevano contenere solo oggetti Object
  - Da Java 5 sono parametrici (generici) rispetto al tipo degli oggetti contenuti

ArrayList

```
import java.util.ArrayList;

ArrayList<Person> team = new ArrayList<Person>();

team.add(new Person("Bob"));
team.add(new Person("Joe"));

team.size()
```

ArrayList

- Per accedere agli elementi occorre usare i metodi get e set (fanno riferimento a indici che iniziano da 0)

```
team.get(1); //restituisce la Person di nome Joe
team.set(0, new Person("Mary")); //sostituisce Mary a Bob
```

- I metodi add(indice, oggetto) e remove(indice) aggiungono e tolgono un elemento nella posizione indicata, alterando la lunghezza dell'ArrayList

```
team.add(1, new Person("Sue")); //ora ci sono Mary, Sue, Joe
team.remove(0); // rimuove Mary, ora ci sono Sue e Joe
```

ArrayList

- add e remove sono operazioni “costose” perché comportano la “traslazione” di segmenti dell’ArrayList
  - Il metodo set **non** deve essere usato per inserire un elemento in una posizione che non c’è

```
team.set(4, new Person("Jack")); // scorretto
```

  - …ma solo per sostituire un oggetto già presente in quella posizione

```
for(Person p: team){
   //fa qualcosa con la persona p
}
```

Genericità e sottotipi

- Alcune cose sembrano andare contro l'intuizione

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; // ERRORE a compile time
```

- Una lista di String è una lista di Object?
  - NO! Se fosse vero, potremmo
    - Inserire una String in una lista di Object
    - Estrarre quello stesso oggetto
    - …che però potrebbe essere assegnato solo ad un riferimento tipo Object
  - …e, chiamando metodi di String, il compilatore direbbe che non sono disponibili in Object
  - In generale, se ClassB è sottoclasse di ClassA allora Gen<ClassB> **non** è sottoclasse di Gen<ClassA>

Metodi generici

```
static <T> void fromArrayToCollection
                                (T[] a, Collection<T> c) {
    for (T o : a) {c.add(o);}
}
```

- Per rendere corretto l'inserimento c.add(o) dobbiamo chiamare fromArrayToCollection con parametri in cui il tipo dell'elemento della collezione sia supertipo di quello dell'array
- Non occorre indicare i parametri attuali in corrispondenza ai parametri formali di un metodo generico
- Ci pensa il compilatore a inferire il tipo del parametro attuale scegliendo il tipo più specifico
  - …cioè il minimo, nell'ordinamento tra tipi definito dalla gerarchia di ereditarietà, tra tutti quelli che rendono legale l'invocazione del metodo generico

Esempi

```
static <T> void fromArrayToCollection
                                (T[] a, Collection<T> c) {
    for (T o : a) {c.add(o);}
}

 String[] sa = new String[100];

 Collection<String> cs = new ArrayList<String>();
 fromArrayToCollection(sa, cs); // T sarà String

 Collection<Object> co = new ArrayList<Object>();
 fromArrayToCollection(sa, co); // T sarà Object

 Integer[] ia = new Integer[100];
 fromArrayToCollection(ia, cs); // ERRORE! String NON è
                                   supertipo di Integer
```

Gestione delle eccezioni

Situazioni eccezionali

- Un metodo deve poter segnalare l'impossibilità di produrre un risultato significativo o la propria terminazione scorretta
  - Apertura di un file (ma il file non esiste)
  - Calcolo della radice quadrata di un numero (ma il numero è negativo)
- Soluzioni
  - Terminazione del programma
  - Restituire un valore convenzionale che rappresenti l'errore
    - Può non essere fattibile
    - Il chiamante potrebbe dimenticarsi di controllare
  - Portare il programma in uno stato scorretto
    - Si usa una variabile globale ERROR
    - …ma il chiamante deve ricordarsi di controllare!
  - Usare un metodo predefinito per la gestione degli errori
    - Ad esempio, si chiama un metodo ERROR(...)
      - Centralizza la gestione degli errori (che spetterebbe al chiamante)
      - Rende difficoltoso il ripristino

Gestione esplicita delle eccezioni

- Una procedura può terminare normalmente
  - …con un risultato valido o “sollevando” un’**eccezione**
- Un’eccezione è un oggetto speciale restituito dal metodo
  - Le eccezioni vengono segnalate al chiamante che può gestirle nella maniera più opportuna
  - Le eccezioni hanno un tipo (Classe)
    - Esempi: DivisionByZeroException, NullPointerException
  - Le eccezioni possono contenere dati che danno indicazioni sul problema incontrato
  - Le eccezioni possono anche essere definite dall’utente (personalizzazione)

Eccezioni in Java

- Un'eccezione può essere catturata e gestita attraverso il costrutto **try/catch**
- try {<blocco>} catch(ClasseEccezione e) {<codice di gestione>}

```
try {
   x = x/y;
}
catch (DivisionByZeroException e) {
   // l'oggetto e è l'oggetto eccezione
   // qui c'è il codice per gestire l'eccezione
   // qui e' possibile usare e
}

// istruzione successiva, da eseguire se non c'e' stata
// eccezione o se catch e' riuscito a "recuperare"
```

Più rami catch

- Un ramo catch(Ex e) può gestire un'eccezione di tipo T se T è di tipo Ex o T è un sottotipo di Ex
- Più clausole catch possono seguire lo stesso blocco try
  - Ciascuna cattura l'eccezione del proprio tipo

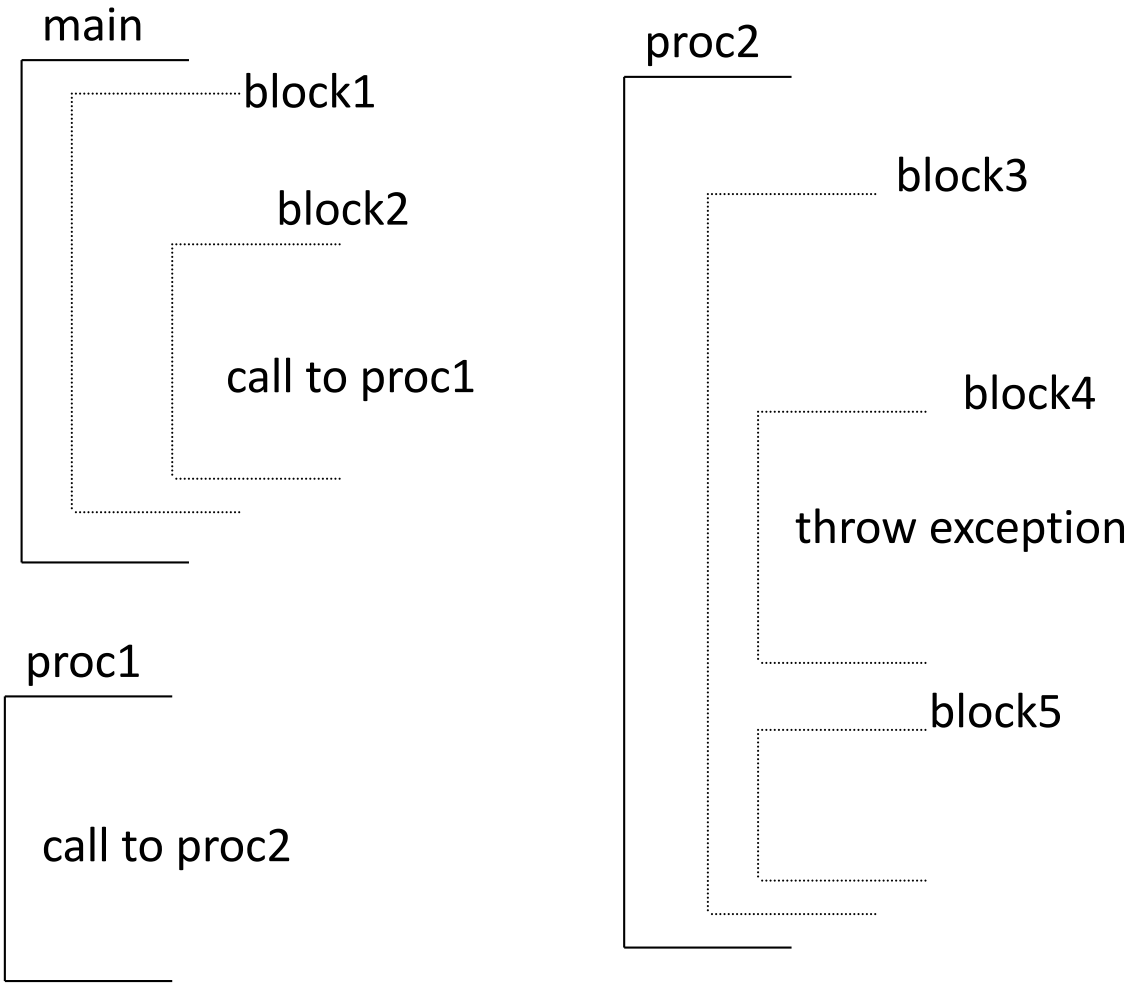
```
public void faiQualcosa() {
        try {
                leggiFile();
        }
        catch(FileInesistenteException fi) {
                System.out.println("Ooops! Il file " +
                        fi.getNomeFile() + " non esiste!");
        }
        catch(FileDanneggiatoException fd) {
                System.out.println("Ooops! Il file " +
                        fd.getNomeFile() + " ha dati scorretti!");
        }
}
```

Propagazione delle eccezioni

- Si termina l'esecuzione del blocco di codice in cui si è verificata l'eccezione e...
  - …se il blocco di codice corrente è un blocco try/catch ed esiste un catch in grado di gestire l'eccezione, si passa il controllo al primo di tali rami catch e, completato questo, alla prima istruzione dopo il blocco, altrimenti…
  - …si risalgono eventuali blocchi di codice più esterni fino a trovare un blocco try/catch che contenga un ramo catch che sia in grado di gestire l'eccezione, altrimenti...
  - …l'eccezione viene propagata al chiamante, fino a che si trova un blocco try/catch che gestisce l'eccezione…
  - …se tale blocco non si trova, il programma termina

Flusso in presenza di eccezioni

- Flusso:
  - main attivato
  - blocco 1
  - blocco 2
  - proc1 invocata
  - proc2 invocata
  - blocco 3
  - blocco 4
  - ... eccezione!
  - propagazione dell'eccezione!



Il ramo finally

- Un blocco try/catch può avere un ramo **finally** in aggiunta a uno o più rami catch
- Il ramo finally è comunque eseguito
  - Sia che all'interno del blocco try non vengano sollevate eccezioni
  - Sia che all'interno del ramo try vengano sollevate eccezioni gestite da un catch
    - In tal caso il ramo finally viene eseguito dopo il ramo catch che gestisce l'eccezione
  - Sia che all'interno del blocco try vengano sollevate eccezioni **non** gestite da un catch

Esempio

```
public class Prova {
   public void read(String fileName) {
     try{
            FileInputStream f=new FileInputStream(fileName);
            ... // use f
     }
      catch(IOException ex) {...}
     finally {f.close();}
   }
}
```

Metodi con eccezioni

- Il fatto che un metodo possa terminare sollevando un'eccezione è dichiarato nella sua interfaccia per mezzo della clausola **throws** per
  - Segnalare un comportamento anomalo incontrato durante l'esecuzione di un'istruzione

```
public int leggiInteroDaInput() throws IOException
```

  - Notificare che una precondizione su un'operazione è stata violata

```
public int fact(int n) throws NegativeException
```

  - Restituire un valore convenzionale

```
public int search(int[] a, int x) throws
  NullPointerException, NotFoundException
```

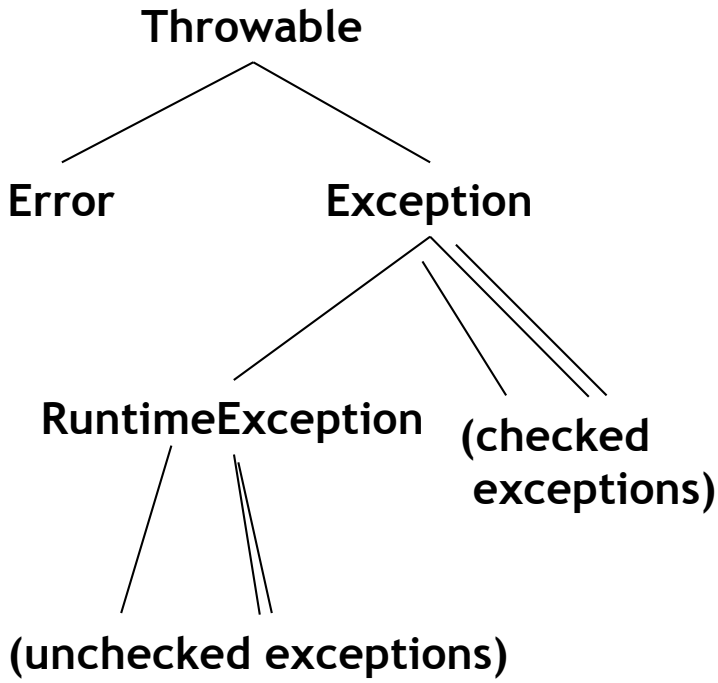
Sollevare eccezioni

- Per sollevare esplicitamente un'eccezione, si usa il comando **throw**, seguito dall'oggetto (del tipo dell'eccezione) da "lanciare" al chiamante
- Informalmente **throw**
  - Termina l'esecuzione del blocco di codice che lo contiene, generando un'eccezione del tipo specificato

```
public int fact(int n) throws NegativeException {
    if (n<0) throw new NegativeException();
    else if (n==0 || n==1) return 1;
    else return (n*fact(n-1));
}
```

Tipi di eccezioni

- Eccezioni definite tramite classi, sottotipo del tipo Throwable
- Esistono due tipi di eccezioni:
  - Eccezioni **checked**
    - Sottotipo di Exception
  - Eccezioni **unchecked**
    - Sottotipo di RuntimeException



Eccezioni checked

- Devono essere dichiarate dai metodi che possono sollevarle (altrimenti si ha un errore in compilazione)
- Quando un metodo M1 invoca un altro metodo M2 che può sollevare un'eccezione di tipo Ex (checked), una delle due deve essere vera
  - L'invocazione di M2 in M1 avviene internamente ad un blocco try/catch che gestisce eccezioni di tipo Ex (quindi, M1 gestisce l'eventuale eccezione)
  - Il tipo Ex (o un suo super-tipo) fa parte delle eccezioni dichiarate nella clausola throws del metodo M1 (quindi, M1 propaga l'eventuale eccezione)

Eccezioni unchecked

- Possono propagarsi senza essere dichiarate in alcuna intestazione di metodo e senza essere gestite da nessun blocco try/catch
- Può essere meglio includerle comunque in throws, per renderne esplicita la presenza (ma per il compilatore è irrilevante)

Definizione di nuove eccezioni

- Gli oggetti di un qualunque tipo T definito dall'utente possono essere usati per sollevare e propagare eccezioni se T è definito come sotto-tipo della classe Exception o RuntimeException
- La definizione della classe che descrive un'eccezione non differisce dalla definizione di una qualsiasi classe definita dall'utente

Definizione di nuove eccezioni

- Può possedere attributi e metodi propri usati per fornire informazioni aggiuntive al gestore dell'eccezione

```
public class NewKindOfException extends Exception {
   public NewKindOfException(){super();}
   public NewKindOfException(String s){super(s);}
}
```

- I due costruttori richiamano semplicemente i costruttori di Exception

```
throw new NewKindOfException("problema!!!")

try{....}
catch(NewKindOfException ecc){
   String s = ecc.toString();
   System.out.println(s);
}
```

Eccezioni con un costruttore

```
public class ProvaEcc {
    public static void main(String[] args) {
     int g,m,a;
     Data d;
     ... // leggi g, m, a
     try {d=new Data(g,m,a);}
     catch(DataIllegaleException e) {
              System.out.println("Inserita una data illegale");
              System.exit(-1);
}}}

public class Data {
    private int giorno, mese, anno;
    private boolean corretta(int g,int m,int a) {...}

    public Data(int g, int m, int a) throws DataIllegaleException {
     if(!corretta(g,m,a)) throw new DataIllegaleException();
     giorno=g; mese=m; anno=a;
}}

class DataIllegaleException extends Exception {};
```

Eccezioni unchecked

- Il loro uso dovrebbe essere limitato ai casi in cui
  - Si tratta di eccezioni di tipo aritmetico/logico
  - C'è un modo conveniente e poco costoso di evitarle
    - Ad esempio, le eccezioni aritmetiche: posso sempre, se mi serve, controllare prima di eseguire il calcolo
    - Ad esempio, per gli array, le eccezioni di tipo OutOfBoundException possono essere evitate controllando in anticipo il valore dell'attributo length dell'array
  - L'eccezione è usata solo in un contesto ristretto
  - Meglio dichiararle comunque in throws quando un metodo le può lanciare

Masking

- Dopo la gestione dell'eccezione, l'esecuzione continua seguendo il normale flusso del programma
  - L'eccezione viene gestita e non si propaga al chiamante
- Eccezione usata per verificare una condizione

```
public static boolean sorted (int[] a) {
        int prev;
        try {prev=a[0];} // lancia eccezione se array e' vuoto
                         // (era meglio check diretto su a)
        catch (IndexOutOfBoundsException e){return true;}
        for (int i=1; i<a.length; i++) {
                if (prev <= a[i]) prev=a[i];
                else return false;
        }
        return true;
}
```

Consigli utili

- Aggiungere ai dati correlati con l'eccezione l'indicazione del metodo che l'ha sollevata
- Nel caso in cui la gestione di un'eccezione comporti un'ulteriore eccezione (**reflecting**), conservare le informazioni da una eccezione alla successiva
- Sebbene sia possibile scegliere liberamente i nomi delle nuove eccezioni definite, è buona convenzione farli terminare con la parola Exception
- Può essere talvolta utile prevedere un package contenente tutte le nuove eccezioni definite
  - A volte invece conviene definire eccezioni come classi private…