Ingegneria del Software

Esercitazione 3

Animals

Definire un package che abbia la classe astratta Animal, le classi Bear Bird Panda con le caratteristiche descritte sotto indicate.

***Specifiche**:*

- *Un animale si ciba (carnivoro, erbivoro, onnivoro).*
- *Un animale puo' migrare (oppure no).*
- *Un animale puo' andare in letargo (oppure no).*
- *Ogni caratteristica deve essere modellata tramite un'opportuna entita' (classe o interfaccia) sfruttando, dove vantaggioso, i vantaggi dati dall'ereditarieta'*
- *Le singole caratteristiche possono essere modellate tramite opportuni metodi che stampano una stringa a video (esempio: carnivoro →"Eat meat")*
- *Orso: Carnivoro, va in letargo. Uccello: Onnivoro, migra. Panda: Erbivoro, va in letargo e migra.*

Animals (i)

Si puo' fare con solo ereditarieta' e classi astratte?
E se aggiungiamo l'uso di interfacce?

Animals (ii)

Si puo' fare con solo ereditarieta' e classi astratte?

No. Java non permette ereditarieta' multipla.

E se aggiungiamo l'uso di interfacce?

Si.

Animals (iii)

- ***Prima versione***:
  - *Package wrong*
  - *Uso solo di classi astratte e ereditarieta'*
  - *Questa versione non funziona*

Animals (iv)

- ***Seconda versione**:*
  - *Package correct*
  - *Uso di classi astratte, ereditarieta' e interfacce*

Animals (iv)

- ***Terza versione***:
  - *Package optimized*
  - *Uso classi astratte, ereditarieta' e interfacce*
  - *Uso dei metodi default di interfaccia*

Collections

Il coltellino svizzero delle strutture dati Java

- ***Interfacce:***
  - List, Set, Map, Queue, Deque, Stack
- ***Implementazioni:***
  - ArrayList, HashSet, HashMap…

Exceptions

Ask for forgiveness

```
try{
    set.add(new Complex(1.0,1.0));
}catch(FullStackException e){
    System.err.println("Stack is full");
}
```

Ask for permission

```
if (!set.isFull()) {
    set.add(new Complex(1.0, 1.0));
}
```

Eccezioni

Cosa stampa il seguente programma? Quando e come termina l'esecuzione?

Eccezioni

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

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

class EccGrandeGrande extends EccGrande {
  public EccGrandeGrande(String s) {super(s);}
}
```

```
public static void main (String[] args) throws
       EccGrandeGrande {
    while (true) {
      try { m1();
            m2();
            m3();
            m4();
      } catch (EccPiccolo e) {
          System.out.println("  Piccolo: " + e);
      } catch (EccGrandeGrande e) {
          System.out.println("  GrandeGrande: " + e);
          throw new EccGrandeGrande("");
      } catch (EccGrande e) {
          System.out.println("  Grande: " + e);
      }}}}
```

```
public class Eccezioni {
static void m1() throws EccPiccolo {
  System.out.println("Entro in m1");
  if (Math.random() < 0.4)
    throw new EccPiccolo("m1");
  System.out.println("Esco da m1");
}

static void m2() throws EccPiccolo,
     EccGrande{
  System.out.println("Entro in m2");
  double x = Math.random();
  if (x < 0.4)
    throw new EccPiccolo("m2");
  if (x > 0.6)
    throw new EccGrande("m2");
  System.out.println("Esco da m2");
}
```

```
static void m3() throws EccGrande,
       EccGrandeGrande {
   System.out.println("Entro in m3");
   double x = Math.random();
   if (x > 0.7)
     throw new
       EccGrandeGrande("m3");
   if (x > 0.6)
     throw new EccGrande("m3");
   System.out.println("Esco da m3");
}

static void m4() throws EccPiccolo {
  System.out.println("Entro in m4");
  m1();
  System.out.println("Esco da m4");
}
```

Stack with Exceptions

Definire la classe Stack

***Specifiche**:*

- *pop(), push()*
- *Se lo stack e' vuoto, pop() lancia l'eccezione OutOfDataException*
- *Se lo stack e' pieno, push lancia l'eccezione FullOfDataException e la gestisce internamente al metodo.*
- *FullOfDataException ha un metodo message() che ritorna la stringa "Ops, troppi dati!"*

Vettori di interi

Implementare la classe Vettore

Specifiche

- Un vettore ha una funzione aggiungiElemento che prende in input un oggetto e lancia l'eccezione *NotValidAddException*
- Un vettore ha una funzione somma che implementa la somma tra due vettori e lancia *Exception* in caso di situazioni impreviste

Stack with Generics

Definire la classe Stack con i Generics.

***Specifiche**:*

- *pop()*
- *push()*

Sort with Generics

Che differenze ci sono tra questi metodi per ordinare un array di Person?
I casting sono necessari?
E' necessario modificare la classe Person?

Ricordate che Person ha i metodi compareTo(Object o) e compareTo(Person p) e Person extends Comparable

Sort with Generics

```
public static List<Comparable> sortAscending(List<Comparable> v) {
        List<Comparable> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((Object) result.get(j)) > 0) {
                    Comparable tmp1 = result.get(j);
                    Comparable tmp2 = result.get(i);
                    …
        }}}
        return result;}
```

Sort with Generics

```
public static List<Comparable> sortAscending(List<Comparable> v) {
        List<Comparable> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((Object) result.get(j)) > 0) {
                    Comparable tmp1 = result.get(j);
                    Comparable tmp2 = result.get(i);
                    …
        }}}
        return result;}
```

Va bene ma Java lancia un warning facendo presente che non stiamo parametrizzando la funzione e quindi non puo' garantirci la sicurezza rispetto ai tipi, specialmente nelle operazioni di casting.

Sort with Generics

```
public static List<Comparable<Person>> sortAscending(List<Comparable<Person>> v) {
        List<Comparable<Person>> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((Person) result.get(j)) > 0) {
                    Comparable<Person> tmp1 = result.get(j);
                    Comparable<Person> tmp2 = result.get(i);
                    …
                }}}
        return result;}
```

Sort with Generics

```
public static List<Comparable<Person>> sortAscending(List<Comparable<Person>> v) {
        List<Comparable<Person>> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((Person) result.get(j)) > 0) {
                    Comparable<Person> tmp1 = result.get(j);
                    Comparable<Person> tmp2 = result.get(i);
                    …
                }}}
        return result;}
```

Ora abbiamo specificato il tipo dell'interfaccia Comparable. Ma Person extends Comparable, non Comparable<Person>. Non abbiamo piu' warning all'interno del metodo ma questo ora accetta solo Liste di oggetti che implementano l'interfaccia Comparable<Person> e quindi che hanno il metodo compareTo(Object o). La nostra classe Person estende Comparable, quindi ha esposto solo il metodo compareTo(Object o). Il casting e' quindi ancora necessario perche' v[j] e' dichiarato nella signature come Comparable<Person> e non Person. Inoltre, quando creiamo nel main la List<Comparable<Person>> e la popoliamo abbiamo comunque dei warning riguardo la sicurezza del tipo

Sort with Generics

```
public static <T> List<Comparable<T>> sortAscending(List<Comparable<T>> v) {
        List<Comparable<T>> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((T) result.get(j)) > 0) {
                    Comparable<T> tmp1 = result.get(j);
                    Comparable<T> tmp2 = result.get(i);
                    …
                }}}
        return result;}
```

Sort with Generics

```
public static <T> List<Comparable<T>> sortAscending(List<Comparable<T>> v) {
        List<Comparable<T>> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((T) result.get(j)) > 0) {
                    Comparable<T> tmp1 = result.get(j);
                    Comparable<T> tmp2 = result.get(i);
                    …
                }}}
        return result;}
```

Il metodo ora e' parametrico. Prende in input una Lista di Comparable di tipo T. Anche qui il casting e' necessario (a compilazione) ma viene lanciato un warning a causa del type erasure. Il casting e' un'operazione che viene fatta a runtime, e Java quando compila il codice sostituisce i tipi dentro al diamond (<>) con la sovraclasse piu' alta. Abbiamo comunque un warning sul casting perche' durante la compilazione Java non e' in grado di assicurarci che a runtime l'oggetto result.get(j) sia di tipo T.

Sort with Generics

```
public static <T extends Comparable<T>> List<Comparable<T>> sortAscending(List<Comparable<T>> v) {
        List<Comparable<T>> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((T) result.get(j)) > 0) {
                    Comparable<T> tmp1 = result.get(j);
                    Comparable<T> tmp2 = result.get(i);
                    …
                }}}
        return result;}
```

Sort with Generics

```
public static <T extends Comparable<T>> List<Comparable<T>> sortAscending(List<Comparable<T>> v) {
        List<Comparable<T>> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo((T) result.get(j)) > 0) {
                    Comparable<T> tmp1 = result.get(j);
                    Comparable<T> tmp2 = result.get(i);
                    …
                }}}
        return result;}
```

Ora stiamo dando informazioni in piu' su T, dicendo che estende Comparable<T> e quindi ha un metodo compareTo(T objectInstance), cosa non ancora corretta (Person extends Comparable e non Comparable<Person>). Quindi, con il type erasure, viene invocato ancora compareTo(Object o). Abbiamo ancora lo stesso warning sul casting.

Sort with Generics

```
public static <T extends Comparable<T>> List<T> sortAscending(List<T> v) {
        List<T> result = new ArrayList<>(v);
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1; j < v.size(); j++) {
                if (v.get(i).compareTo(result.get(j)) > 0) {
                    T tmp1 = result.get(j);
                    T tmp2 = result.get(i);
                    …
                }}}
        return result;}
```

Ora stiamo dicendo che in input il metodo prende, nel nostro caso, una List<Person>. Cosi' facendo il casting non e piu' necessario. Quando invochiamo il metodo pero' il warning sulla sicurezza di tipo e' ancora presente e viene ancora invocato compareTo(Object o)

Sort with Generics

Come evitare i warning?

Sort with Generics

Come evitare i warning?

```
Person extends Comparable<Person> {
        …
        @Override
        public int compareTo(Person o) {
                …
        }

}
```

In questo modo il type erasure non sostituisce Object (il capostipite della gerarchia di oggetti) ma Person perche' e' la classe piu' alta nella gerarchia dell'ereditarieta' delle nostre classi che implementa compareTo(Person p)! A questo punto i warning quando invochiamo il metodo spariscono e viene invocata solo compareTo(Person p) e non compareTo(Object o).
P.s. se eseguite i codici precedenti mettendo delle stampe dentro ai duei metodi compareTo() viene sempre chiamato compareTo(Object o) per primo.

Esercizio: CodaIllimitata

- CodaIllimitata può contenere oggetti di qualsiasi tipo. Inserito il primo elemento però tutti i successivi devono essere dello stesso tipo del primo
- Usare un'interfaccia *Accodabile* che definisce il metodo *confrontaClasse*