Laboratorio di Programmazione

Lezione 15 – OOP II e Eccezioni

Marco Anisetti (teoria)
Dipartimento di Informatica
marco.anisetti@unimi.it

Matteo Luperto (lab. turno A)
Dipartimento di Informatica
matteo.luperto@unimi.it

Nicola Bena (lab. turno B)
Dipartimento di Informatica
nicola.bena@unimi.it

Eccezioni: Intro

- La gestione degli errori in `C` prevede che le funzioni in cui si possono verificare errori restituiscano un `int`
- *Convenzione* tra sviluppatore della funzione e utilizzatore della funzione sulla semantica del valore restituito
- Formalmente, non c'è nessuna differenza tra una funzione che restituisce un intero perché è il risultato della sua computazione e una funzione che restituisce un intero come modo per segnalare un errore
- Chi intercetta l'errore poi "avvisa" scrivendo un messaggio su `stderr` (`o via error.h`)

Eccezioni-*ish* in C

```
int save_list(struct LinkedList * list, char * filename) {
  FILE * file = fopen(filename, "w");
  if (filename == NULL || list == NULL) return -2;
  if (file == NULL) return -1;
  // niente da fare
  if (list->size == 0) return 1;
  struct Node * current = list->head;
  while (current != NULL){
    current = current->next;
      if (fprintf(file, "%d\n", current->data) < 0) {
      return -3;
      }
  }
  if (fclose(file) != 0) return -4;
  return 1;
}
```

Eccezioni

- Costrutto per la gestione strutturata dei possibili errori che si possono verificare nell'esecuzione del programma
- Classi "speciali" usate esclusivamente per la gestione degli errori
- Un metodo viene marcato con l'elenco delle eccezioni che si possono verificare durante la sua esecuzione
- L'esecuzione si interrompe quando un'istruzione solleva un'eccezione, a quel punto il metodo in esecuzione può
  - Gestire localmente l'eccezione: verificare se l'eccezione è stata *sollevata* durante l'esecuzione e agire di conseguenza (*cattura di un'eccezione*)
    - Non occorre marcare il metodo, perché l'eccezione viene gestita in quel punto
  - Demandare la gestione al chiamante: il metodo sarà anch'esso marcato come metodo in cui si può verificare un'eccezione (*propagazione di un'eccezione*)

Eccezioni

```
public static double divisione(double a, double b) throws Exception {
  if (b == 0){
    throw new Exception("b dev'essere != 0");
  }
  return a / b;
}
```

Eccezioni

```
public static double divisione(double a, double b) throws Exception {
  if (b == 0){
    throw new Exception("b dev'essere != 0");
  }
  return a / b;
}
```

`throws` <*nome classe eccezione che può essere sollevata*>

Sollevo un'eccezione: interrompo il flusso di esecuzione

Il tipo dell'eccezione nel prototipo deve corrispondere al tipo sollevato (oppure `throws Exception`)

Eccezioni

```
public class Example {

  public static double divisione(double a, double b) throws Exception {
    if (b == 0){
      throw new Exception("b dev'essere != 0");
    }
    return a / b;
  }

  public static void main(String[] args){
    double n1 = 10.5, n2 = 3.14;
    double result = divisione(n1, n2);
  }
}
```

Eccezioni

```
public class Example {

  public static double divisione(double a, double b) throws Exception {
    if (b == 0){
      throw new Exception("b dev'essere != 0");
    }
    return a / b;
  }

  public static void main(String[] args){
    double n1 = 10.5, n2 = 3.14;
    double result = divisione(n1, n2);
  }
}
```

Errore: il metodo chiamante (`main`) non specifica come gestire l'eccezione del metodo chiamato (`divisione`): il codice non compila

→ Sono obbligato a gestire gli errori (ottima cosa!)

Eccezioni

```
public class Example {
  public static double divisione(double a, double b) throws Exception {
    if (b == 0){
      throw new Exception("b dev'essere != 0");
    }
   return a / b;
  }
  public static void main(String[] args){
      double n1 = Double.valueOf(args[0]), n2 = Double.valueOf(args[1]);
      try {
        double result = divisione(n1, n2);
        System.out.printf("Risultato: %f\n", result);
      } catch (Exception e) {
        System.out.printf("Si è verificato un errore: %s\n", e.toString());
      }
  }
}
```

Propagare un'Eccezione

- Un metodo che esegue un'istruzione che può sollevare un'eccezione può demandarne la gestione al chiamante
  - Il metodo deve obbligatoriamente indicarlo nel suo prototipo
  - Tutte le eccezioni che possono essere sollevate e non vengono catturate devono essere aggiunte a `throws`

```
public static double divisione(double a, double b) throws Exception {
  if (b == 0){
    throw new Exception("b dev'essere != 0");
  }
  return a / b;
}

public static double calcolatrice(double n1, double n2) throws Exception {
  double result = divisione(n1, n2);
  // altre operazioni
  return result;
}
```

Propagare un'Eccezione

- Un metodo che esegue un'istruzione che può sollevare un'eccezione può demandarne la gestione al chiamante
  - Il metodo deve obbligatoriamente indicarlo nel suo prototipo
  - Tutte le eccezioni che possono essere sollevate e non vengono catturate devono essere aggiunte a `throws`

```
public static double divisione(double a, double b) throws Exception {
  if (b == 0){
    throw new Exception("b dev'essere != 0");
  }
  return a / b;
}

public static double calcolatrice(double n1, double n2) throws Exception {
  double result = divisione(n1, n2);
  // altre operazioni
  return result;
}
```

Metodo che può sollevare eccezione

Chiamata

Metodo che non gestisce eccezione localmente, quindi la propaga al chiamante, e così a cascata

Catturare un'Eccezione: Try/Catch

```
public class Example {
  public static double divisione(double a, double b) throws Exception {
    if (b == 0){
      throw new Exception("b dev'essere != 0");
    }
   return a / b;
  }
  public static void main(String[] args){
      double n1 = Double.valueOf(args[0]), n2 = Double.valueOf(args[1]);
      try {
        double result = divisione(n1, n2);
        System.out.printf("Risultato: %f\n", result);
      } catch (Exception e) {
        System.out.printf("Si è verificato un errore: %s\n", e.toString());
      }
  }
}
```

L'istruzione problematica viene racchiusa nel blocco `try`: se durante la sua esecuzione viene sollevata un'eccezione, allora viene eseguito il blocco di istruzioni nel `catch`; in ogni caso, poi, si esegue la prima istruzione che segue il blocco `try/catch`

Catturare un'Eccezione: Try/Catch

```
public class Example {
  public static double divisione(double a, double b) throws Exception {
    if (b == 0){
      throw new Exception("b dev'essere != 0");
    }
   return a / b;
  }
  public static void main(String[] args){
      double n1 = Double.valueOf(args[0]), n2 = Double.valueOf(args[1]);
      try {
        double result = divisione(n1, n2);
        System.out.printf("Risultato: %f\n", result);
      } catch (Exception e) {
        System.out.printf("Si è verificato un errore: %s\n", e.toString());
      }
  }
}
```

```
> java Example 15.0 3.0
Risultato: 5.000000
```

L'istruzione problematica viene racchiusa nel blocco `try`: se durante la sua esecuzione viene sollevata un'eccezione, allora viene eseguito il blocco di istruzioni nel `catch`; in ogni caso, poi, si esegue la prima istruzione che segue il blocco `try/catch`

Catturare un'Eccezione: Try/Catch

```java
public class Example {
  public static double divisione(double a, double b) throws Exception {
    if (b == 0){
      throw new Exception("b dev'essere != 0");
    }
   return a / b;
  }
  public static void main(String[] args){
      double n1 = Double.valueOf(args[0]), n2 = Double.valueOf(args[1]);
      try {
        double result = divisione(n1, n2);
        System.out.printf("Risultato: %f\n", result);
      } catch (Exception e) {
        System.out.printf("Si è verificato un errore: %s\n", e.toString());
      }
  }
}
```

```
> java Example 15.0 0
Si è verificato un errore: java.lang.Exception: b dev'essere != 0
```

L'istruzione problematica viene racchiusa nel blocco `try`: se durante la sua esecuzione viene sollevata un'eccezione, allora viene eseguito il blocco di istruzioni nel `catch`; in ogni caso, poi, si esegue la prima istruzione che segue il blocco `try/catch`

Try/Catch Avanzato

- Per ogni `try` è possibile specificare più di un blocco `catch`, ciascuno con una specifica eccezione (o un insieme di eccezioni)
- Catturando `Exception` si catturano tutte le eccezioni *figlie* di `Exception`, quindi *tutte*

Try/Catch Avanzato

```java
public class SimpleList {
  private List<Integer> list;
  private int max_size;
  public SimpleList(int n) {
    this.list = new ArrayList<Integer>(n);
    this.max_size = n;
  }
  public void save(String file) throws
        IOException, IndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(
        new FileWriter(file));
   for (int i = 0; i < this.max_size; i++) {
     out.printf("%f\n", this.list.get(i));
   }
   out.close();
  }
}
```

Non è un codice da prendere a esempio

```java
public static void main(String []args){
  SimpleList l = new SimpleList(10);
  try {
    l.save("output.txt");
  } catch (Exception e){
    e.printStackTrace();
  }
}
```

Con questo `catch` catturo qualsiasi eccezione possibile

```
❯ java ExampleMultiCatch1
java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0
        at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
        at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
        at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:248)
        at java.base/java.util.Objects.checkIndex(Objects.java:372)
        at java.base/java.util.ArrayList.get(ArrayList.java:459)
        at SimpleList.save(SimpleList.java:21)
        at ExampleMultiCatch1.main(ExampleMultiCatch1.java:14)
```

Try/Catch Avanzato

```
public class SimpleList {
  private List<Integer> list;
  private int max_size;
  public SimpleList(int n) {
    this.list = new ArrayList<Integer>(n);
    this.max_size = n;
  }
  public void save(String file) throws
        IOException, IndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(
        new FileWriter(file));
   for (int i = 0; i < this.max_size; i++) {
     out.printf("%f\n", this.list.get(i));
   }
   out.close();
  }
}
```

Non è un codice da prendere a esempio

```
public static void main(String []args){
  SimpleList l = new SimpleList(10);
  try {
    l.save("output.txt");

  } catch (IOException|IndexOutOfBoundsException e){

    e.printStackTrace();
  }
}
```

Specifico l'elenco delle eccezioni da catturare in questo blocco `catch`

`e` sarà un oggetto di tipo `IOException` oppure `IndexOutOfBoundException`, a seconda dell'eccezione sollevata a run time

Try/Catch Avanzato

```
public class SimpleList {
  private List<Integer> list;
  private int max_size;
  public SimpleList(int n) {
    this.list = new ArrayList<Integer>(n);
    this.max_size = n;
  }
  public void save(String file) throws
        IOException, IndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(
        new FileWriter(file));
    for (int i = 0; i < this.max_size; i++) {
     out.printf("%f\n", this.list.get(i));
    }
    out.close();
  }
}
```

Non è un codice da prendere a esempio

```
public static void main(String []args){
  SimpleList l = new SimpleList(10);
  try {
    l.save("output.txt");

  } catch (IOException e){
    // errore su file
  } catch (IndexOutOfBoundsException e){
    // errore su accesso lista
  }
}
```

Diversi blocchi `catch`: il programma può gestire in modo diverso il tipo di errore

- `IOException`: il file non è valido, chiedo all'utente di re-inserirlo (ad esempio)
- `IndexOutOfBoundsException`: c'è un bug nel codice, non accedo correttamente agli elementi della lista

Eccezioni Custom

- Possiamo definire nuove eccezioni definendo una classe che estende `Exception`
- Poi usabile nel codice come qualsiasi altra eccezione
- Comodo per dare una semantica migliore, e consentire una gestione più granulare

```
public class ListSavingException extends Exception {
  public ListSavingException(String message, Throwable cause) {
    super(message, cause);
  }
}
```

Try/Catch Avanzato: Finally

Non è un codice da prendere a esempio

```
public class SimpleList_Exception {
  private List<Integer> list;
  private int max_size;
  public SimpleList_Exception(int n) {
    this.list = new ArrayList<Integer>(n);
    this.max_size = n;
  }
  public void save(String file) throws ListSavingException {
    PrintWriter out;
    try {
      out = new PrintWriter(new FileWriter(file));
      for (int i = 0; i < this.max_size; i++) {
        out.printf("%f\n", this.list.get(i));
      }
    } catch (IOException|IndexOutOfBoundsException e){
      throw new ListSavingException("error", e);

    } finally {
     if (out != null) {
       out.close();
     }
    }
  }
}
```

Try/Catch Avanzato: Finally

Non è un codice da prendere a esempio

```
public class SimpleList_Exception {
  private List<Integer> list;
  private int max_size;
  public SimpleList_Exception(int n) {
    this.list = new ArrayList<Integer>(n);
    this.max_size = n;
  }
  public void save(String file) throws ListSavingException {
PrintWriter out;
    try {
      out = new PrintWriter(new FileWriter(file));
      for (int i = 0; i < this.max_size; i++) {
        out.printf("%f\n", this.list.get(i));
      }
    } catch (IOException|IndexOutOfBoundsException e){
      throw new ListSavingException("error", e);

    } finally {
     if (out != null) {
       out.close();
     }
    }
  }
}
```

`catch` cattura le eccezioni e ne solleva un'altra

Il blocco `finally` viene eseguito in ogni caso dopo l'esecuzione del blocco `try`

Prima di tornare al chiamante sollevando `ListSavingException`, si esegue il blocco `finally`, chiudendo le eventuali risorse aperte

Catturare vs Propagare

- La cattura va usata solo
  - Nei metodi di più alto livello, ovvero i metodi che compongono ed interagiscono con i diversi oggetti (es `main`)
    - Altrimenti, il rischio è che l'errore non venga notato da chi usa le nostre classi, assumendo che sia andato tutto bene
  - Quando la computazione può procedure comunque senza violare il contratto di uso delle classi
    - Esempio: caricamento di una risorsa esterna, per cui si può procedure con un default

Catturare vs Propagare

- La cattura va usata solo
  - Nei metodi di più alto livello, ovvero i metodi che compongono ed interagiscono con i diversi oggetti (es `main`)
    - Altrimenti, il rischio è che l'errore non venga notato da chi usa le nostre classi, assumendo che sia andato tutto bene
  - Quando la computazione può procedure comunque senza violare il contratto di uso delle classi
    - Esempio: caricamento di una risorsa esterna, per cui si può procedure con un default
- La propagazione mantiene l'informazione del punto esatto in cui si è originato l'errore attraverso la pila di chiamate
  - Utile per debugging
- → lasciamo all'*utente* del nostro codice la scelta della gestione le eccezioni

Catturare vs Propagare

```java
public class Example {
  static double divisione(double a, double b) throws
Exception {
    if (b == 0){ throw new Exception("b dev'essere
!= 0"); }
    return a / b;
  }
  static double calcolatrice(double a, double b) {
    double result;
    // altre operazioni
    try {
      result = divisione(a, b);
    } catch (Exception e){ e.printStackTrace(); }
    return result;
  }
  public static void main(String[] args){
    double n1 = Double.valueOf(args[0]);
    double n2 = Double.valueOf(args[1]);
    calcolatrice(n1, n2);
  }
}
```

```java
public class Example {
  static double divisione(double a, double b) throws
Exception {
    if (b == 0){ throw new Exception("b dev'essere !=
0"); }
    return a / b;
  }

  static double calcolatrice(double a, double b) throws
Exception {
    double result;
    // altre operazioni
    result = divisione(a, b);
   return result;
  }

  public static void main(String[] args){
     double n1 = Double.valueOf(args[0]);
     double n2 = Double.valueOf(args[1]);
     try {
      double result = calcolatrice(n1, n2);
      System.out.printf("Risultato: %f\n", result);
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Eccezioni Unchecked

- Tutte le classi che rappresentano un'eccezione estendono la classe `Exception`
- Le eccezioni che estendono `RuntimeException` (che a sua volta estende `Exception`) sono detto *unchecked*: sono eccezioni che non è obbligatorio gestire
- I metodi che sollevano eccezioni unchecked sono comunque annotati con le (alcune delle) relative eccezioni che possono sollevare
  - Al momento della chiamata, però, non è obligatorio gestire esplicitamente l'eccezione
  - Se l'eccezione si verifica e non è gestita, il programma termina comunque in quel punto
- Esempi
  - `NullPointerException` se si accede a un attributo/chiama metodo di una variabile non inizializzata
  - metodo `parseDouble` della classe Double può sollevare un'eccezione `NumberFormatException`
- → se possibile, usiamo eccezioni non unchecked
- Oracle Docs: *Unchecked Exceptions - The Controversy (https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)*

Ereditarietà

- Data una classe, è possibile *estenderla* modificandone il comportamento
- La classe che viene estesa viene detta *classe padre*
- La nuova classe che la estende viene detta *classe figlia* (o *sotto-classe*), ed *eredita* attributi e metodi della classe padre
- L'estensione non è arbitraria, si può soltanto
  - Aggiungere attributi
  - Aggiungere metodi
  - Modificare metodi esistenti rispettandone il prototipo
- Le modifiche sono ulteriormente regolate dalla visibilità della classe padre e dei suoi componenti
- Una classe astratta (`abstract`) è una classe che non può essere istanziata ed è progettata per essere estesa, ovvero fornire attributi e metodi di base da utilizzare all'interno di classi figlie

Ereditarietà

- Le modifiche sono ulteriormente regolate dalla visibilità della classe padre e dei suoi componenti
  - Attributo `private`: non visibile nella/e classe figlia/e se non tramite get/set
  - Metodo `private`: non visibile nella/e classe figlia/e
  - Attributo `protected`: visibile nella/e classe figlia/e
  - Metodo `protected`: visibile e modificabile nella/e classe/i figlia/e
  - Attributi e metodi `public`: sempre visibili
- Se prevediamo che la nostra classe debba essere estesa, usiamo `protected` per gli attributi/metodi necessari
- `Overriding`: modificare un metodo di una classe padre (senza modificarne il prototipo)
  - È quello che abbiamo fatto con `toString` e `equals`, perché sono ereditati dalla classe padre di default, cioè Object
- `final`: metodo di cui non si può fare override

Ereditarietà: Esempio

- Progettiamo una gerarchia di classi per gestire un concessionario che vende veicoli a motore di vario tipo. I veicoli sono
  - Auto (per uso privato)
  - Moto (per uso privato)
  - Camion (per uso commerciale)
  - Autobus (per uso pubblico)
  - Aereo (per uso di linea)
- Dev’essere possibile
  - Mettere in moto e spegnere un veicolo
  - Vendere un veicolo
  - Altri metodi specifici sulla base del veicolo

Ereditarietà: Esempio

- La classe padre deve contenere le caratteristiche comuni di tutte le classi figlie, in termini di attributi e metodi
- La gerarchia specializza man mano le classi

```
public abstract class Vehicle {
  protected double price;
  // modellazione più sofisticata avrebbe definito
  // una classe astratta Engine e relative classi figlie.
  protected EngineType engineType; // enum
  protected boolean isOn;
  protected Vehicle(double price, EngineType engineType) {
    this.price = price;
    this.engineType = engineType;
    this.isOn = false;
  }
  public double getPrice(){
    return this.price;
  }
  public void turnOn(){
    // se già accesa, non succede nulla.
    // avremmo potuto usare eccezioni.
    this.isOn = true;
  }
  public void turnOff(){
    this.isOn = false;
  }
}
```

Ereditarietà: Esempio

- La prima differenza in termine di attributi e metodi deriva dall'uso del veicolo: per uso privato o per trasporto di passeggeri
  - Per comodità, assimiliamo uso privato e uso commerciale
- In alternativa: tipologia di veicolo (stradale, aviazione, etc)

- Una classe figlia deve obbligatoriamente chiamare il costruttore della classe padre all'interno del proprio costruttore, in modo da inizializzare anche la classe padre
  - In generale, per riferirci a un attributo/metodo della classe padre usiamo la keyword `super`
  - `super(params)` è il modo per chiamare il costruttore della classe padre
  - `super.method(params)` è il modo per chiamare il metodo `method` della classe padre

Ereditarietà: Esempio

```
public abstract class PrivateVehicle extends Vehicle {
  protected int numeroOccupanti;
  protected PrivateVehicle(double price, EngineType engineType, int numeroOccupanti){
    super(price, engineType);
    this.numeroOccupanti = numeroOccupanti;
  }
}
```

Metodi `PrivateVehicle.getNumeroOccupanti`, `PublicVehicle.getNumeroEquipaggio`, e `PublicVehicle.getNumeroOccupanti` omessi per brevità. Visibilità: `public` per riuso

```
public abstract class PublicVehicle extends Vehicle {
  protected int numeroEquipaggio;
  protected int numeroPasseggeri;
  protected PublicVehicle(double price, EngineType engineType, int numeroEquipaggio,
        int numeroPasseggeri){
    super(price, engineType);
    this.numeroEquipaggio = numeroEquipaggio;
    this.numeroPasseggeri = numeroPasseggeri;
  }
}
```

Ereditarietà: Esempio

Dichiarazione con riferimento alla classe padre

```
public abstract class PrivateVehicle extends Vehicle {

   protected int numeroOccupanti;

   protected PrivateVehicle(double price, EngineType engineType, int numeroOccupanti){
     super(price, engineType);
     this.numeroOccupanti = numeroOccupanti;
   }
}
```

```
public abstract class PublicVehicle extends Vehicle {

   protected int numeroEquipaggio;
   protected int numeroPasseggeri;

   protected PublicVehicle(double price, EngineType engineType, int numeroEquipaggio,
         int numeroPasseggeri){
     super(price, engineType);
     this.numeroEquipaggio = numeroEquipaggio;
     this.numeroPasseggeri = numeroPasseggeri;
   }
}
```

Ereditarietà: Esempio

Dichiarazione attributi. Oltre a questi, eredita gli attributi della classe padre

```
public abstract class PrivateVehicle extends Vehicle {

  protected int numeroOccupanti;

  protected PrivateVehicle(double price, EngineType engineType, int numeroOccupanti){
    super(price, engineType);
    this.numeroOccupanti = numeroOccupanti;
  }
}

public abstract class PublicVehicle extends Vehicle {

  protected int numeroEquipaggio;
  protected int numeroPasseggeri;

  protected PublicVehicle(double price, EngineType engineType, int numeroEquipaggio,
        int numeroPasseggeri){
    super(price, engineType);
    this.numeroEquipaggio = numeroEquipaggio;
    this.numeroPasseggeri = numeroPasseggeri;
  }
}
```

Ereditarietà: Esempio

```
public abstract class PrivateVehicle extends Vehicle {

  protected int numeroOccupanti;

  protected PrivateVehicle(double price, EngineType engineType, int numeroOccupanti){
    super(price, engineType);
    this.numeroOccupanti = numeroOccupanti;
  }
}

public abstract class PublicVehicle extends Vehicle {

  protected int numeroEquipaggio;
  protected int numeroPasseggeri;

  protected PublicVehicle(double price, EngineType engineType, int numeroEquipaggio,
        int numeroPasseggeri){
    super(price, engineType);
    this.numeroEquipaggio = numeroEquipaggio;
    this.numeroPasseggeri = numeroPasseggeri;
  }
}
```

Dichiarazione costruttore con chiamata al costruttore padre

Ereditarietà: Esempio

```
public class Truck extends PrivateVehicle {
  private double maxLoad;
  private double currentLoad;
  private boolean containerDoorStatus;

  public Truck(double price, EngineType engineType,
         double load) {
    super(price, engineType, 1);
    this.maxLoad = load;
    this.currentLoad = 0.0;
    // false: chiusa.
    this.containerDoorStatus = false;
  }

  public double getLoad(){
    return this.currentLoad;
}
```

Ereditarietà: Esempio

```
public class Truck extends PrivateVehicle {
  private double maxLoad;
  private double currentLoad;
  private boolean containerDoorStatus;

  public Truck(double price, EngineType engineType,
         double load) {
    super(price, engineType, 1);
    this.maxLoad = load;
    this.currentLoad = 0.0;
    // false: chiusa.
    this.containerDoorStatus = false;
  }

  public double getLoad(){
    return this.currentLoad;
}
```

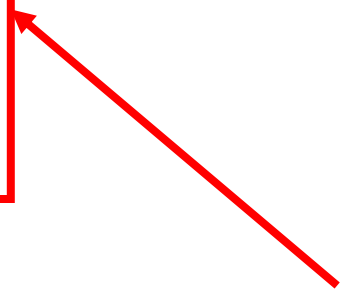
Dichiarazione attributi (capacità di carico, carico corrente, stato della porta del vano merci)

Ereditarietà: Esempio

```
public class Truck extends PrivateVehicle {
  private double maxLoad;
  private double currentLoad;
  private boolean containerDoorStatus;

  public Truck(double price, EngineType engineType,
         double load) {
    super(price, engineType, 1);
    this.maxLoad = load;
    this.currentLoad = 0.0;
    // false: chiusa.
    this.containerDoorStatus = false;
  }

  public double getLoad(){
    return this.currentLoad;
}
```



Costruttore: nella chiamate al padre specifica direttamente il numero di occupanti (`1` - terzo parametro), senza prenderlo in ingresso

Ereditarietà: Esempio

```
public class Truck extends PrivateVehicle {
  private double maxLoad;
  private double currentLoad;
  private boolean containerDoorStatus;

  public Truck(double price, EngineType engineType,
         double load) {
    super(price, engineType, 1);
    this.maxLoad = load;
    this.currentLoad = 0.0;
    // false: chiusa.
    this.containerDoorStatus = false;
  }

  public double getLoad(){
    return this.currentLoad;
}
```

```
  public double load(double weight) {
    if (this.currentLoad + weight > this.maxLoad) {
      throw new IllegalArgumentException("too much!");
    }
    this.currentLoad += weight;
    return currentLoad;
  }
  public double unload(double weight){
    if (this.currentLoad - weight < 0) {
      throw new IllegalArgumentException("too much!");
    }
    this.currentLoad -= weight;
    return currentLoad;
  }

  @Override
  public void turnOn(){
    // se porta aperto, la chiudo
    if (this.containerDoorStatus) {
      this.containerDoorStatus = false;
    }
    // poi, chiamo versione originale.
    super.turnOn();
    }
}
```

Ereditarietà: Esempio

```
public class Truck extends PrivateVehicle {
  private double maxLoad;
  private double currentLoad;
  private boolean containerDoorStatus;

  public Truck(double price, EngineType engineType,
         double load) {
    super(price, engineType, 1);
    this.maxLoad = load;
    this.currentLoad = 0.0;
    // false: chiusa.
    this.containerDoorStatus = false;
  }

  public double getLoad(){
    return this.currentLoad;
}
```

Metodi specifici per il tipo di veicolo

```
  public double load(double weight) {
    if (this.currentLoad + weight > this.maxLoad) {
      throw new IllegalArgumentException("too much!");
    }
    this.currentLoad += weight;
    return currentLoad;
  }
  public double unload(double weight){
    if (this.currentLoad - weight < 0) {
      throw new IllegalArgumentException("too much!");
    }
    this.currentLoad -= weight;
    return currentLoad;
  }

  @Override
  public void turnOn(){
    // se porta aperto, la chiudo
    if (this.containerDoorStatus) {
    this.containerDoorStatus = false;
    }
    // poi, chiamo versione originale.
    super.turnOn();
    }
}
```

Ereditarietà: Esempio

```
public class Truck extends PrivateVehicle {
  private double maxLoad;
  private double currentLoad;
  private boolean containerDoorStatus;

  public Truck(double price, EngineType engineType,
         double load) {
    super(price, engineType, 1);
    this.maxLoad = load;
    this.currentLoad = 0.0;
    // false: chiusa.
    this.containerDoorStatus = false;
  }

  public double getLoad(){
    return this.currentLoad;
}
```

Override del metodo `turnOn`

```
public double load(double weight) {
  if (this.currentLoad + weight > this.maxLoad) {
    throw new IllegalArgumentException("too much!");
  }
  this.currentLoad += weight;
  return currentLoad;
}
public double unload(double weight){
  if (this.currentLoad - weight < 0) {
    throw new IllegalArgumentException("too much!");
  }
  this.currentLoad -= weight;
  return currentLoad;
}

@Override
public void turnOn(){
  // se porta aperta, la chiudo
  if (this.containerDoorStatus) {
    this.containerDoorStatus = false;
  }
  // poi, chiamo versione originale.
  super.turnOn();
  }
}
```

Ereditarietà: Esempio

```
import java.util.ArrayList;
import java.util.List;

public class Concessionaire {
  private List<Vehicle> vehiclesToSell;

  public Concessionaire(){
    this.vehiclesToSell = new ArrayList<>();
  }
  public void addVehicle(Vehicle v){
    this.vehiclesToSell.add(v);
  }

  public Vehicle sell(int vehicleIndex, double price) throws StingyException {
    Vehicle v = this.vehiclesToSell.get(vehicleIndex);

      if (price >= v.getPrice()){
        // posso vendere.
        return this.vehiclesToSell.remove(vehicleIndex);
      } else {
        // not enough!
        throw new StingyException(v, price);
      }
   }
}
```

Ereditarietà: Esempio

```
import java.util.ArrayList;
import java.util.List;

public class Concessionaire {
  private List<Vehicle> vehiclesToSell;

  public Concessionaire(){
    this.vehiclesToSell = new ArrayList<>();
  }
  public void addVehicle(Vehicle v){
    this.vehiclesToSell.add(v);
  }

  public Vehicle sell(int vehicleIndex, double price) throws StingyException {

    Vehicle v = this.vehiclesToSell.get(vehicleIndex);
      if (price >= v.getPrice()){
        // posso vendere.
        return this.vehiclesToSell.remove(vehicleIndex);
      } else {
        // not enough!
        throw new StingyException(v, price);
      }
   }
}
```

Dichiariamo una lista che contiene oggetti di tipo `Vehicle`: può contenere qualsiasi oggetto di una classe figlia di `Vehicle`

I metodi prendono in ingresso una variabile di tipo `Vehicle` e restituiscono un oggetto di tipo `Vehicle`

Ereditarietà

- Una classe astratta può definire dei metodi `abstract`, ovvero metodi di cui specifica solo il prototipo
- Le classi figlie *non astratte* devono **obbligatoriamente** definirne l'implementazione rispettando il prototipo

Ereditarietà

- Una classe astratta può definire dei metodi `abstract`, ovvero metodi di cui specifica solo il prototipo
- Le classi figlie *non astratte* devono **obbligatoriamente** definirne l'implementazione rispettando il prototipo
- Usiamo questo costrutto per rifinire l'esempio proposto, definendo un metodo `double roadTax()` che calcola il bollo da pagare sulla base delle caratteristiche del veicolo

Ereditarietà: Esempio

```
public abstract class Vehicle {

  // metodi e attributi già visti

  public abstract double roadTax();
}
```

Ereditarietà: Esempio

```
public abstract class Vehicle {

  // metodi e attributi già visti

  public abstract double roadTax();
}
```

```
public class Truck extends PrivateVehicle {

 // metodi e attributi già visti

  @Override
  public double roadTax() {
    if (this.engineType == EngineType.ELECTRIC)
      return this.maxLoad * 100;
    else
      return this.maxLoad * 500;
  }
}
```

Interfacce e Polimorfismo

- Interfaccia: classe composta di soli metodi astratti
- Definisce appunto una *interfaccia*, cioè un comportamento, senza specificarne l'implementazione
- Una classe può *estendere* una sola classe
- Una classe può *implementare* una (o più) interfaccia
- Prassi
  - Quando si dichiara una variabile, non si specifica il tipo concreto ma l'interfaccia implementata
  - Quando la si inizializza, si usa il costruttore del tipo concreto
  - Quando si chiama un metodo su di essa, viene esegue il metodo del tipo concreto
- → chiamata uniforme rispetto alla specifica implementazione

Interfacce e Polimorfismo: Esempio 1

- Le classi `LinkedList` e `ArrayList` implementano le funzionalità di una lista
- L'interfaccia `List` definisce quali sono i metodi che ogni classe che implementa una lista deve avere
  - dichiaro variabile `list` di tipo `List`
  - Inizializzo variabile `list` di tipo `LinkedList` o `ArrayList`
  - A run time, `list.add(element)` corrisponderà al metodo di una delle due classi a seconda di quale è stata istanziata

Interfacce e Polimorfismo: Esempio 1

- Le classi `LinkedList` e `ArrayList` implementano le funzionalità di una lista
- L'interfaccia `List` definisce quali sono i metodi che ogni classe che implementa una lista deve avere

```java
import java.util.ArrayList;

public class ListUsage {

  private ArrayList<String> list;

  public ListUsage(){
    this.list = new ArrayList<>();
  }
}
```

```java
import java.util.ArrayList;
import java.util.List;

public class ListUsage {

  private List<String> list;

  public ListUsage(){
    this.list = new ArrayList<>();
  }
}
```

Interfacce e Polimorfismo: Esempio 2

- Rifiniamo l'esempio introducendo un'interfaccia
- "Fattorizziamo" un comportamento comune, ovvero la possibilità di essere acceso o spento
- Definiamo un'interfaccia `TurnOnable`
  - Per convenzione, le interfaccia finiscono in `-able`, es `Closable`

```
public interface TurnOnable {
  public void turnOn();
  public void turnOff();
}
```

Interfacce e Polimorfismo: Esempio 2

- Rifiniamo l'esempio introducendo un'interfaccia
- "Fattorizziamo" un comportamento comune, ovvero la possibilità di essere acceso o spento
- Definiamo un'interfaccia `TurnOnable`
  - Per convenzione, le interfaccia finiscono in `-able`, es `Closable`

```
public interface TurnOnable {
    public void turnOn();
    public void turnOff();
}
```

Non serve specificare che i metodi sono `abstract`

Interfacce e Polimorfismo: Esempio 2

- Specifichiamo che `Vehicle` implementa questa interfaccia, ovvero implementa dei metodi con il stesso prototipo di quelli dell'interfaccia
- In questo caso `Vehicle` contiene anche un'implementazione di default dei due metodi
  - Altrimenti, implementazione demandata alle classi figlie

```
public abstract class Vehicle implements TurnOnable {
  // ...
  public void turnOn(){
    // se già accesa, non succede nulla.
    // avremmo potuto usare eccezioni
    this.isOn = true;
  }
  public void turnOff(){
    this.isOn = false;
  }
}
```

Interfacce e Polimorfismo: Esempio 2

- E’ possibile quindi dichiarare una variabile di tipo `TurnOnable`, anziché di tipo `Truck`, `Car`, `Bus`
- Vantaggio: se chi usa le nostre classi deve solo fare delle operazioni di accensione e spegnimento, non è interessato al tipo specifico, ma solo al fatto che possano essere accendibili o spegnibili, quindi che siano “di tipo” `TurnOnable`
- Altri comportamenti comuni fattorizzabili?

Interfacce e Polimorfismo: Esempio 2

- E’ possibile quindi dichiarare una variabile di tipo `TurnOnAble`, anziché di tipo `Truck`, `Car`, `Bus`
- Vantaggio: se chi usa le nostre classi deve solo fare delle operazioni di accensione e spegnimento, non è interessato al tipo specifico, ma solo al fatto che possano essere accendibili o spegnibili, quindi che siano “di tipo” `TurnOnable`
- Altri comportamenti comuni fattorizzabili?
  - `Loadable` per `load` e `unload` di merci/bagagli, applicabile a `Truck` e `Aircraft`

Interfacce e Polimorfismo: Esempio 2

```
public static void main(String[] args){
  Concessionaire c = new Concessionaire();
  c.addVehicle(new Truck(1000, EngineType.COMBUSTION, 10));
  c.addVehicle(new Car(500, EngineType.ELECTRIC));

  try {
    TurnOnable vehicle1 = c.sell(0, 1000);
    vehicle1.turnOn();

    TurnOnable vehicle2 = c.sell(1, 1000);
    vehicle2.turnOn();

  } catch (StingyException e){
    e.printStackTrace();
  }
}
```

Interfacce e Polimorfismo: Esempio 2

```
public static void main(String[] args){
  Concessionaire c = new Concessionaire();
  c.addVehicle(new Truck(1000, EngineType.COMBUSTION, 10));
  c.addVehicle(new Car(500, EngineType.ELECTRIC));

  try {
    TurnOnable vehicle1 = c.sell(0, 1000);
    vehicle1.turnOn();

    TurnOnable vehicle2 = c.sell(1, 1000);
    vehicle2.turnOn();

  } catch (StingyException e){
    e.printStackTrace();
  }
}
```

`vehicle1` è di tipo `Truck`, quindi `turnOn` corrisponde all'implementazione presente in `Truck`

Interfacce e Polimorfismo: Esempio 2

```
public static void main(String[] args){
  Concessionaire c = new Concessionaire();
  c.addVehicle(new Truck(1000, EngineType.COMBUSTION, 10));
  c.addVehicle(new Car(500, EngineType.ELECTRICAL));

  try {
    TurnOnable vehicle1 = c.sell(0, 1000);
    vehicle1.turnOn();

    TurnOnable vehicle2 = c.sell(1, 1000);
    vehicle2.turnOn();

  } catch (StingyException e){
    e.printStackTrace();
  }
}
```

`vehicle2` è di tipo `Car`, che non fa override di `turnOn`. Quindi, questo corrisponde all'implementazione nella classe `Vehicle`

Interfacce e Polimorfismo: Esempio 3

- L'interfaccia `Comparable<Type>` definisce il metodo `int compareTo(Type o)`
- Il metodo restituisce un `int` tale che
  - `<0`: l'oggetto su cui il metodo è invocato è minore di `o`
  - `=0`: l'oggetto su cui il metodo è invocato è uguale a `o`
  - `>0`: l'oggetto su cui il metodo è invocato è maggiore di `o`
- Siamo noi a definire cosa vuol dire minore/uguale/maggiore nell'implementazione del metodo

Interfacce e Polimorfismo: Esempio 3

- Vantaggio: una collezione di oggetti la cui classe implementa l'interfaccia `Comparable` può essere ordinata senza dover scrivere l'algoritmo di ordinamento
- Tutto quello che serve all'algoritmo di ordinamento è sapere come confrontare due elementi, ovvero che gli oggetti siano istanza di una classe che implementa `Comparable`

Interfacce e Polimorfismo: Esempio 3

```
public abstract class Vehicle implements TurnOnable, Comparable<Vehicle> {

  // altri attributi e metodi.

  public int compareTo(Vehicle other){
    return Double.compare(this.price, other.price);
  }
}
```

Interfacce e Polimorfismo: Esempio 3

```
public abstract class Vehicle implements TurnOnable, Comparable<Vehicle> {

  // altri attributi e metodi.

  public int compareTo(Vehicle other){
    return Double.compare(this.price, other.price);
  }
}
```

Specifica delle interfacce implementate

Ordinamento basato sull'attributo `price`

Interfacce e Polimorfismo: Esempio 3

```
public abstract class Vehicle implements TurnOnable, Comparable<Vehicle> {

  // altri attributi e metodi.

  public int compareTo(Vehicle other){
    return Double.compare(this.price, other.price);
  }
}
```

```
public class Concessionaire {

  private List<Vehicle> vehiclesToSell;

  // altri attributi e metodi.

  public void sortVehicles(){
    this.vehiclesToSell.sort(null);
  }
```

Interfacce e Polimorfismo: Esempio 3

```java
public abstract class Vehicle implements TurnOnable, Comparable<Vehicle> {

  // altri attributi e metodi.

  public int compareTo(Vehicle other){
    return Double.compare(this.price, other.price);
  }
}
```

Grazie all'implementazione di `Comparable`, possiamo sfruttare il metodo `sort` già definito per ordinare la List di `Vehicle`

```java
public class Concessionaire {

  private List<Vehicle> vehiclesToSell;

  // altri attributi e metodi.

  public void sortVehicles(){
    this.vehiclesToSell.sort(null);
  }
}
```

Iteratori

- Convenzione per scorrere una collezione di elementi indipendentemente da come sono memorizzati
  - Lista
  - Tabella di hash
  - Albero di ricerca
  - Veicoli del concessionario
- Interfaccia (generica) `Iterator` che definisce due metodi principali
- `hasNext()`: restituisce `true` se ci sono altri elementi nella collezione
- `next()`: restituisce il prossimo elemento nella collezione, solleva `NoSuchElementException` se tutti gli elementi sono stati restituiti
- `remove()`: rimuove l'ultimo elemento restituito dalla collezione

Iteratori

- Interfaccia che può essere implementata per le classi che scriviamo, se prevediamo che l'utilizzatore delle nostre classi debba ciclare su un insieme di elementi memorizzati
- Le classi che fanno parte del framework *Collection* (contenitori di altri oggetti) implementano l'interfaccia `Iterator`
- Convenzione: il metodo `iterator()` restituisce un oggetto che implementa l'interfaccia `Iterator`
  - Come lo fa dietro le quinte? Non importa, a noi basta `hasNext()` per sapere quando interrompere il ciclo e `next()` per ottenere il prossimo elemento

Iteratori: Esempio 1

```
List<String> l = new ArrayList<>();

Iterator<String> it = l.iterator();

while(it.hasNext()){
  String element = it.next();
  // fai qualcosa con element
}
```

Iteratori: Esempio 1

```
List<String> l = new ArrayList<>();

Iterator<String> it = l.iterator();

while(it.hasNext()){
  String element = it.next();
  // fai qualcosa con element
}
```

1) Ottengo un oggetto che implementa l'interfaccia `Iterator`

Iteratori: Esempio 1

```
List<String> l = new ArrayList<>();

Iterator<String> it = l.iterator();

while(it.hasNext()){
  String element = it.next();
  // fai qualcosa con element
}
```

1) Ottengo un oggetto che implementa l'interfaccia `Iterator`

Attenzione: dichiariamo il tipo generico dell'`Iterator` in modo che sia compatibile con l'elemento ottenuto a ogni chiamata a `next()`

Iteratori: Esempio 1

```
List<String> l = new ArrayList<>();

Iterator<String> it = l.iterator();

while(it.hasNext()){
  String element = it.next();
  // fai qualcosa con element
}
```


2) Scorro gli elementi

Iteratori: Esempio 1

```
List<String> l = new ArrayList<>();

Iterator<String> it = l.iterator();

while(it.hasNext()){
  String element = it.next();
  // fai qualcosa con element
}
```

E’ un esempio di polimorfismo!

Iteratori: Esempio 2

```
List<Integer> l = new ArrayList<>();

Iterator<Integer> it = l.iterator();

while(it.hasNext()){

  int element = it.next();
    if (element == 0) {
      it.remove();
   }
}
```

Iteratori: Esempio 2

```
List<Integer> l = new ArrayList<>();

Iterator<Integer> it = l.iterator();

while(it.hasNext()){

   int element = it.next();
     if (element == 0) {
       it.remove();
    }
}
```

Downcasting

- Operazione per cui si fa il casting di un oggetto da una classe padre a una classe figlia
  - Assumendo che la variabile sia una variabile che istanzia la classe figlia
- Per le prassi viste fin'ora, può capitare di trovarsi con una variabile il cui tipo è una classe padre ma, nella pratica, ha un tipo più specifico
- Per poter utilizzare i metodi della classe figlia, occorre prima farne il casting alla classe figlia
- Esempio
  - Variabile di tipo `Truck` dichiarata di tipo `Vehicle`: per usare `load` e `unload` dobbiamo prima avere una variabile di tipo Truck, perché `load` e `unload` non sono definiti per la classe `Vehicle`

Downcasting

```
Concessionaire c = new Concessionaire();
c.addVehicle(new Truck(1000, EngineType.COMBUSTION, 10));
c.addVehicle(new Car(500, EngineType.ELECTRIC));

// acquisto il primo veicolo
try {

  Vehicle vehicle = c.sell(0, 1000);
  Truck truck1 = (Truck) vehicle;
  // ora che ho una variabile di tipo Truck,
  // posso invocare il metodo load.
  truck1.load(500);
} catch (StingyException e){
  e.printStackTrace();
}
```

Downcasting

```
Concessionaire c = new Concessionaire();
c.addVehicle(new Truck(1000, EngineType.COMBUSTION, 10));
c.addVehicle(new Car(500, EngineType.ELECTRIC));

// acquisto il primo veicolo
try {
  Vehicle vehicle = c.sell(0, 1000);
  Truck truck1 = (Truck) vehicle;
  // ora che ho una variabile di tipo Truck,
  // posso invocare il metodo load.
  truck1.load(500);
} catch (StingyException e){
  e.printStackTrace();
}
```

Cast da `Vehicle` a `Truck`: necessario per poter invocare il metodo `load`

Riferimenti e Copie

- Gli oggetti ricevuti in ingresso e restituiti da metodi sono, dietro le quinte, dei puntatori
- Questo può portare ad avere più puntatori alla stessa zona di memoria, e quindi a modifiche inavvertite
  - Copia di puntatori = copia *shallow*, come negli assegnamenti con `=`
- Es: restituendo un attributo via `get`, il chiamante può modificare l'oggetto ottenuto, e quindi modificare anche l'attributo (lez. 14)
  - Non è detto sia un comportamento sbagliato, ma un fatto da tenere presente

Riferimenti e Copie

- Per evitarlo, è necessario fare una copia *deep* dell'oggetto preso in ingresso/restituito, copiando ricorsivamente tutti gli attributi
- Dove
  - Nel costruttore, in modo che l'oggetto abbia una sua versione, separata rispetto al chiamante, di tutti gli attributi
  - Nei metodi get
- Prima di farlo, pensate bene se possa servire o meno
  - Ad esempio, se l'oggetto restituito non può essere modificato tramite i suoi metodi, non serve

Riferimenti e Copie: Esempio

```
public class Engine {

  private int power;
  private String producer;
  private EngineType engineType;
  private boolean status;

  public Engine(int power, String producer,
        EngineType engineType) {
    this.power = power;
    this.producer = producer;
    this.engineType = type;
    this.status = false;
  }
}
```

```
public class Car {

  private String producer;
  private String model;
  private Engine engine;

  public Car(String producer, String model,
        Engine engine) {
    this.producer = producer;
    this.model = model;
    this.engine = engine;
  }

  public Engine getEngine() {
    return this.engine;
  }
}
```

Riferimenti e Copie: Esempio

```
public class Engine {

  private int power;
  private String producer;
  private EngineType engineType;
  private boolean status;

  public Engine(int power, String producer,
        EngineType engineType) {
    this.power = power;
    this.producer = producer;
    this.engineType = type;
    this.status = false;
  }
}
```

Il chiamante ottiene lo stesso `Engine` memorizzato come attributo e può modificarlo: quindi può <u>modificare un oggetto senza usare i metodi dell'oggetto</u>!

```
public class Car {

  private String producer;
  private String model;
  private Engine engine;

  public Car(String producer, String model,
        Engine engine) {
    this.producer = producer;
    this.model = model;
    this.engine = engine;
  }

  public Engine getEngine() {
    return this.engine;
  }
}
```

Riferimenti e Copie: Esempio

```
public class Engine {
  private int power;
  private String producer;
  private EngineType engineType;
  private boolean status;
  public Engine(int power, String producer,
        EngineType engineType) {
    this.power = power;
    this.producer = producer;
    this.engineType = type;
    this.status = false;
  }

  public Engine copy(){
    return new Engine(this.power, this.producer,
        this.engineType);
  }
}
```

Restituisco un nuovo oggetto copiando gli attributi. In questo caso sono attributi "semplici", quindi nel costruttore:

- `power` è di tipo `int` quindi viene passato per valore
- `producer` è di tipo `String` quindi viene passato per indirizzo: sarebbe un problema perché la nuova istanza contiene un puntatore alla stessa stringa, ma eventuali modifiche creano sempre una nuova istanza di `String`, quindi va bene
- `EngineType` è un enumerator quindi immutabile

→ Il costruttore crea una copia completamente deep

Riferimenti e Copie: Esempio

```java
public class Engine {
  private int power;
  private String producer;
  private EngineType engineType;
  private boolean status;
  public Engine(int power, String producer,
        EngineType engineType) {
    this.power = power;
    this.producer = producer;
    this.engineType = type;
    this.status = false;
  }

  public Engine copy(){
    return new Engine(this.power, this.producer,
        this.engineType);
  }
}
```

```java
public class Car {
  private String producer;
  private String model;
  private Engine engine;

  public Car(String producer, String model,
        Engine engine) {
    this.producer = producer;
    this.model = model;
    this.engine = engine.copy();
  }

  public Engine getEngine() {
    return this.engine.copy();
  }

  public Car copy(){
    return new Car(this.producer, this.model,
        this.engine.copy());
  }
}
```

Riferimenti e Copie: Esempio

```java
public class Engine {
  private int power;
  private String producer;
  private EngineType engineType;
  private boolean status;
  public Engine(int power, String producer,
        EngineType engineType) {
    this.power = power;
    this.producer = producer;
    this.engineType = type;
    this.status = false;
  }

  public Engine copy(){
    return new Engine(this.power, this.producer,
        this.engineType);
  }
}
```

```java
public class Car {
  private String producer;
  private String model;
  private Engine engine;

  public Car(String producer, String model,
        Engine engine) {
    this.producer = producer;
    this.model = model;
    this.engine = engine.copy();
  }

  public Engine getEngine() {
    return this.engine.copy();
  }

  public Car copy(){
    return new Car(this.producer, this.model,
        this.engine.copy());
  }
}
```

Nel costruttore chiamo `copy` dove necessario, in modo da memorizzare una copia deep

Riferimenti e Copie: Esempio

```java
public class Engine {
  private int power;
  private String producer;
  private EngineType engineType;
  private boolean status;
  public Engine(int power, String producer,
        EngineType engineType) {
    this.power = power;
    this.producer = producer;
    this.engineType = type;
    this.status = false;
  }

  public Engine copy(){
    return new Engine(this.power, this.producer,
        this.engineType);
  }
}
```

```java
public class Car {
  private String producer;
  private String model;
  private Engine engine;

  public Car(String producer, String model,
        Engine engine) {
    this.producer = producer;
    this.model = model;
    this.engine = engine.copy();
  }

  public Engine getEngine() {
    return this.engine.copy();
  }

  public Car copy(){
    return new Car(this.producer, this.model,
        this.engine.copy());
  }
}
```

Nei `get` chiamo `copy` dove necessario,
in modo da restituire una copia deep

Riferimenti e Copie: Esempio

```java
public class Engine {
  private int power;
  private String producer;
  private EngineType engineType;
  private boolean status;
  public Engine(int power, String producer,
        EngineType engineType) {
    this.power = power;
    this.producer = producer;
    this.engineType = type;
    this.status = false;
  }

  public Engine copy(){
    return new Engine(this.power, this.producer,
        this.engineType);
  }
}
```

```java
public class Car {
  private String producer;
  private String model;
  private Engine engine;

  public Car(String producer, String model,
        Engine engine) {
    this.producer = producer;
    this.model = model;
    this.engine = engine.copy();
  }

  public Engine getEngine() {
    return this.engine.copy();
  }

  public Car copy(){
    return new Car(this.producer, this.model,
        this.engine.copy());
  }
}
```

Nell'implementazione di `copy` credo una copia deep, chiamando `copy` a cascata sugli attributi per cui è necessario