
Classi astratte

Dal materiale del prof. Angelo Di Iorio
Angelo.diiorio@unibo.it

Classi astratte

- A volte non ha senso definire operazioni particolari per una classe “generica”
- Pensiamo ad es. alla classe **Shape** e le sue sottoclassi **Circle** e **Rectangle**
 - Che senso ha creare oggetti di classe Shape senza sapere quale figura geometrica creare?
 - Che senso ha ritornare l'area o il perimetro di una figura indefinita?

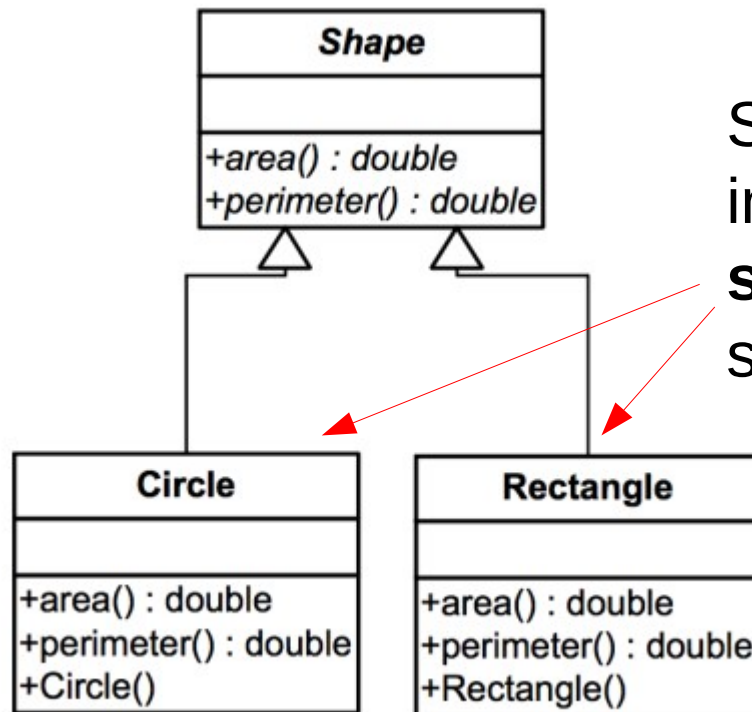
Classi astratte

```
public class Shape {  
    public double getArea() {  
        return 0;  
    }  
    public double getPerimeter() {  
        return 0;  
    }  
    public void printInfo(){  
        System.out.println(  
            "Perimeter: " + this.getPerimeter() +  
            " - Area: " + this.getArea());  
    }  
}
```

Potrebbe aver senso
(Shape indefinita =
punto) ma evitabile...

Classi astratte

- Si può mantenere la gerarchia di classi, **senza implementare** area e perimeter



Si “*delega*” la loro implementazione alle **sottoclassi** che specializzano Shape

Classi astratte

- Non ha senso definire area e perimetro per una forma indefinita...
 - ...Ma ha senso farlo per una forma “*ben definita*”
 - Es. un cerchio o un triangolo... o anche un punto!
- Serve un meccanismo per:
 - **Evitare** l'implementazione di metodi “indefiniti”
 - **Forzare** l'implementazione di tali metodi per quelle **sottoclassi** di cui si vuole creare oggetti

Classi astratte

- Possibile soluzione: **metodi astratti**
 - dichiarati dalla classe ma **non implementati**
- Si dichiarano con la keyword **abstract**
- Se una classe dichiara **almeno** un metodo astratto, allora è una **classe astratta**
 - Anche le classi astratte vanno dichiarate con la keyword **abstract**

```
Public abstract class Shape {  
    abstract public double getArea(); Non c'è body!  
    abstract public double getPerimeter();
```

```
public class Rectangle extends Shape {  
    //variabili e costruttore omissi  
  
    @Override  
    public double getArea() {return s1 * s2;}  
  
    @Override  
    public double getPerimeter() {return (s1 + s2) * 2;}
```

```
...  
public class Circle extends Shape {  
    //variabili e costruttore omissi  
  
    @Override  
    public double getArea() {return r * r * 3.14;}  
  
    @Override  
    public double getPerimeter() {return 2 * 3.14 * r;}  
...  
...
```

Classi astratte

- Una classe astratta **A** è un **tipo** di dato che offre **almeno** una **funzionalità generica** (metodo astratto) che “ha senso” per **sotto-classi concrete** di A
 - Cioè sottoclassi di A che **implementano tutti** i metodi astratti di A
- **Non** si possono creare **oggetti** di una classe astratta!
 - Come si dovrebbero comportare se richiamati su uno dei suoi metodi astratti?

Classi astratte

- Classe astratta = **base** per le **sottoclassi concrete** che **devono** implementarne i metodi astratti
- Le classi astratte **possono** implementare anche metodi **non astratti**
 - E.g. `Shape.printInfo`
- Si possono avere **gerarchie** di classi **astratte**

Esempio

- Es. la classe astratta Shape potrebbe avere le **sottoclassi astratte** ConvexShape e ConcaveShape
 - Nessuna di queste classi può essere istanziata
- Circle e Rectangle diventerebbero sottoclassi di ConvexShape
 - Ad es. ConcaveShape potrebbe aggiungere nuovi metodi astratti, ad es. **convexHull()**

Esempio

- **AbstractMap** è una classe astratta **built-in**
 - <https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html>

```
public abstract class AbstractMap<K,V>  
extends Object  
implements Map<K,V>
```

This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.

To implement an unmodifiable map, the programmer needs only to extend this class and provide an implementation for the `entrySet` method, which returns a set-view of the map's mappings. Typically, the returned set will, in turn, be implemented atop `AbstractSet`. This set should not support the `add` or `remove` methods, and its iterator should not support the `remove` method.

To implement a modifiable map, the programmer must additionally override this class's `put` method (which otherwise throws an `UnsupportedOperationException`), and the iterator returned by `entrySet().iterator()` must additionally implement its `remove` method.

The programmer should generally provide a void (no argument) and map constructor, as per the recommendation in the Map interface specification.

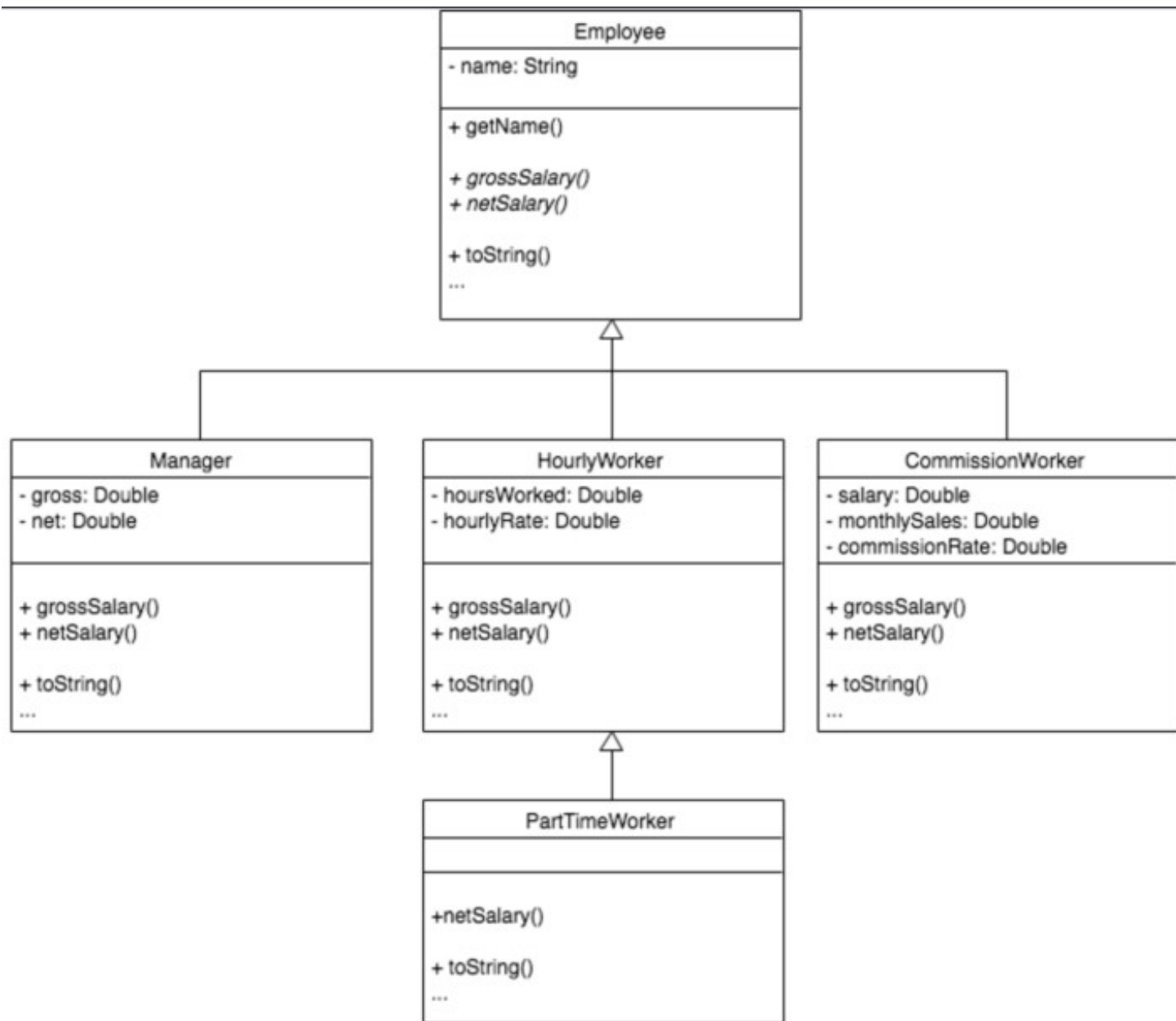
The documentation for each non-abstract method in this class describes its implementation in detail. Each of these methods may be overridden if the map being implemented admits a more efficient implementation.

Esempio: classe Employee

- Modelliamo lo scenario di un'azienda con 4 tipi di lavoratori: Manager, HourlyWorker, PartTimeWorker, CommissionWorker
- Gli stipendi sono calcolati a seconda del ruolo. Per ogni lavoratore si tiene traccia di:
 - Nome
 - Stipendio lordo
 - Stipendio netto

Esempio: classe Employee

- **Manager**: $\text{netto} = \text{lordo} - 10\% \text{ trattenute}$
- **HourlyWorker**: $\text{lordo} = \text{n.ore lavorate} * \text{paga oraria}$; 5% trattenute
- **PartTime**: come HourlyWorker, ma senza trattenute
- **CommissionWorker**: $\text{lordo} = \text{salario fisso} + \text{eventuale bonus vendite}$; 10% trattenute



Classe Employee

```
public abstract class Employee {  
    private String name;  
  
    public Employee(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    abstract public double grossSalary();  
    abstract public double netSalary();  
}
```



Metodi
Astratti

```
public class Manager extends Employee {  
  
    private double gross;  
    private double net;  
  
    public Manager(String name, double salary) {  
        super(name);  
        gross = salary;  
        net = 0.9 * gross;  
    }  
  
    public double grossSalary() {  
        return gross;  
    }  
  
    public double netSalary() {  
        return net;  
    }  
  
    public String toString() {  
        return "Manager[" + "name = " + getName() + ",  
            gross = " + gross + ", net = " + net + "];"  
    }  
}
```



```
public class HourlyWorker extends Employee {  
    private double hoursWorked;  
    private double hourlyRate;  
  
    public HourlyWorker(String name, double hoursWorked,  
double hourlyRate) {  
        super(name);  
        this.hoursWorked = hoursWorked;  
        this.hourlyRate = hourlyRate;  
    }  
  
    public double grossSalary() {  
        return hoursWorked * hourlyRate;  
    }  
  
    public double netSalary() {  
        return this.grossSalary() * 0.95;  
    }  
  
    public String toString() {  
        return "HourlyWorker[" + "name = " + getName() + ",  
gross = " + grossSalary() + ", net = " + netSalary() + "];"  
    }  
}
```

```
public class CommissionWorker extends Employee{
    private double salary;
    private double monthlySales;
    private double commissionRate;

    public CommissionWorker(String name, double salary, double
monthlySales, double commissionRate) {
        super(name);
        this.salary = salary;
        this.monthlySales = monthlySales;
        this.commissionRate = commissionRate;
    }

    public double grossSalary() {
        return salary + monthlySales * commissionRate / 100;}

    public double netSalary() {return this.grossSalary() * 0.9;}

    public String toString() {
        return "CommissionWorker[" + "name = " + getName() + ",
gross = " + grossSalary() + ", net = " + netSalary() + "];"
    }
}
```

CommissionWorker.java

Cicli “polimorfi”

- Ogni classe astratta è un **tipo** di dato
 - **Non** posso creare oggetti di quel tipo
- Si possono usare “**cicli polimorfi**” per iterare su elementi che hanno come tipo **statico** la classe astratta
 - Come nell’esempio con la classe Shape
- A tempo di esecuzione (**binding dinamico**) verranno associate le opportune **sottoclassi concrete**

Cicli “polimorfi”

- **Esempio:** usiamo un **array di Employee**, stampiamo lo stipendio di ognuno e alla fine calcoliamo la spesa totale per gli stipendi
- Non possiamo creare oggetti di classe Employee, ma possiamo dichiarare che il **tipo** degli elementi dell'array è Employee
- Il binding dinamico di Java richiamerà i metodi opportuni in base al ruolo

```
public class EmployeesDemo {
```

EmployeesDemo.java

```
    private Employee[] staff;
```

```
    private double totalGrossSalary;
```

```
    private double totalBenefits;
```

```
    private double totalNetSalary;
```

```
    public void doTest() {
```

```
        Employee[] staff = new Employee[5];
```

```
        staff[0] = new Manager("Fred", 800);
```

```
        staff[1] = new Manager("Ellen", 700);
```

```
        staff[2] = new HourlyWorker("John", 37, 13.50);
```

```
        staff[3] = new PartTimeWorker("Gord", 35, 12.75);
```

```
        staff[4] = new CommissionWorker("Mary", 400, 15000, 3.5);
```

// continua in slide successiva

// continua doTest() da slide precedente

EmployeesDemo.java

```
totalGrossSalary = 0.0;
```

```
totalNetSalary = 0.0;
```

```
for (int i = 0; i < staff.length; i++) {  
    totalGrossSalary = totalGrossSalary + staff[i].grossSalary()  
    totalNetSalary = totalNetSalary + staff[i].netSalary();  
    System.out.println(staff[i]);  
};
```

```
totalBenefits = totalGrossSalary - totalNetSalary;
```

```
System.out.println("Total gross salary: " + totalGrossSalary);
```

```
System.out.println("Total benefits: " + totalBenefits);
```

```
System.out.println("Total net salary: " + totalNetSalary);
```

```
...
```

Output doTest()

```
Manager[name = Fred, gross = 800.0, net = 720.0]  
Manager[name = Ellen, gross = 700.0, net = 630.0]  
HourlyWorker[name = John, gross = 499.5, net = 474.525]  
PartTimeWorker[name = Gord, gross = 446.25, net = 446.25]  
CommissionWorker[name = Mary, gross = 925.0, net = 832.5]  
Total gross salary: 3370.75  
Total benefits: 267.47499999999999  
Total net salary: 3103.275
```

Esercizio motori

- Implementare le classi Java per descrivere il motore di un'automobile. Ogni motore è caratterizzato da:
 - cilindrata (intero)
 - numero_cilindri (intero)
- Da queste informazioni è possibile derivare la potenza (in cavalli, di tipo double) in base al tipo di motore.
- Esistono tre tipi di motore:
 - benzina – potenza: $(\text{cilindrata} / \text{numero_cilindri}) * 0.1$
 - diesel – potenza: $(\text{cilindrata} / \text{numero_cilindri}) * 0.2$
 - metano – potenza: $((\text{cilindrata} * 0.8) / \text{numero_cilindri}) * 0.25$

Esercizio motori

- Definire la classe astratta Motore e le opportune classi concrete
- Implementare una classe test per verificare il funzionamento delle classi e dei metodi
- La classe di test deve contenere un metodo che prende in input un vettore di motori e restituisce la media tra le potenze dei motori nel vettore

Esercizio animali

- Implementare le classi per modellare animali. Ogni animale ha un certo numero di zampe e fa un verso:
 - Il gatto è quadrupede e miagola
 - Il cane è quadrupede e abbaia
 - Il tacchino è bipede e gogglotta
- Definire le classi Animale, Quadrupede, Bipede, Cane e Tacchino e i metodi per leggere il verso dell'animale (stringa) e il numero di zampe (intero)
- Implementare il metodo **toString()** e una classe per testare il funzionamento di classi e dei metodi

Esercizio animali

- Aggiungere le seguenti funzionalità:
- Ogni animale ha un **nome** e un **anno** di nascita. Per semplicità l'età si calcola come il numero di anni trascorsi dalla **data corrente**
- E' possibile confrontare un animale con un altro (anche di specie diversa) in base ai loro anni. La classe `Animale` espone un metodo **`piuGrandeDi(Animale a)`** che restituisce `true` se l'istanza su cui è invocata ha età maggiore di quella passata come parametro, `false` altrimenti.

Interfacce

Dal materiale del prof. Angelo Di Iorio
Angelo.diiorio@unibo.it


Interfacce

- Uno dei punti chiave della **OOP** è la separazione tra:
 - **Interfaccia**: elenca le funzionalità di una classe (metodi e attributi public)
 - **Implementazione**: definisce il funzionamento interno di ogni metodo
- Posso invocare i metodi di un'interfaccia **senza conoscerne** l'implementazione
- Java permette la **separazione** tra interfaccia e implementazione

Interfacce

- Le interfacce Java sono entità che possono essere **implementate** da diverse classi

```
public interface IShape {  
    //Calcola area  
    public double getArea();  
  
    //Calcola perimetro  
    public double getPerimeter();  
}
```



Interfacce

- Un'interfaccia si definisce con la keyword **interface**
- Contiene le **intestazioni** dei metodi pubblici
- Può anche definire **costanti** pubbliche
- Può contenere anche la **documentazione** per gli utilizzatori dell'interfaccia
 - Buona norma!
- Per convenzione il nome di un'interfaccia inizia con la lettera **maiuscola**
 - Come per le classi

Interfacce

- Un'interfaccia contiene **solo prototipi** e **nessuna implementazione**
 - ~ “*classi totalmente astratte*”
- Una classe può **implementare** un'interfaccia: si usa la keyword **implements**
 - Anzichè **extends**
- Una classe che implementa un'interfaccia **DEVE implementarne TUTTI i metodi**

Implementare interfacce

```
public class Rectangle implements IShape {
```

```
    //variabili e costruttore omessi
```

```
    @Override
```

```
    public double getArea() {return s1 * s2;}
```

```
    @Override
```

```
    public double getPerimeter() {return (s1 + s2) * 2;}
```

```
...
```

```
public class Circle implements IShape {
```

```
    //variabili e costruttore omessi
```

```
    @Override
```

```
    public double getArea() {return r * r * 3.14;}
```

```
    @Override
```

```
    public double getPerimeter() {return 2 * 3.14 * r;}
```

```
...
```

Perchè interfacce?

- Permettono ai progettisti SW di **specificare funzionalità** (metodi) ai programmatori
 - Senza dover implementare nulla
- Es. voglio che la mia interfaccia Persona esibisca un metodo codiceFiscale che ne ritorni il codice fiscale
 - Quali parametri?
- Le interfacce aiutano a rendere il codice modulare e interoperabile

Interfacce

- Un'interfaccia **non** include **costruttori**
 - Specifica funzionalità ma **non come creare** oggetti che la implementano
- Una classe che implementa un'interfaccia **deve** implementarne tutti i metodi, ma può anche **aggiungerne** altri
- Un'interfaccia può essere implementata da **più classi**

Interfacce multiple

- Una classe Java **non** può estendere più di un'altra classe
 - **No ereditarietà multipla**: al più una superclasse
- Può però **implementare più interfacce**
- Se una classe C implementa le interfacce I_1, I_2, \dots, I_n allora è considerata **sottotipo** di I_1, I_2, \dots, I_n
 - **$x \text{ instanceof } I_k = \text{true}$** per ogni oggetto x di classe C e per $k=1, \dots, n$

Interfacce multiple

```
public enum Color {BLACK, WHITE, YELLOW, RED, GREEN}

public interface IColorable {

    public Color getColor();

    public void setColor(Color c);

}
```

Interfacce multiple

```
public class Rectangle implements IShape, IColorable {  
    //variabili e costruttore omessi
```

```
@Override  
public double getArea() {return s1 * s2;}
```

```
@Override  
public double getPerimeter() {return (s1 + s2) * 2;}
```

```
@Override  
public Color getColor() { return this.c; }
```

```
@Override  
public void setColor(Color c) {this.c = c;}
```

```
...
```

Interfacce vs classi astratte

Proprietà	Interfacce	Classi astratte
Ereditarietà	Una classe può implementare più interfacce	Una classe può avere una sola classe padre
Implementazione	Un'interfaccia non implementa nessun metodo; è completamente astratta	Una classe astratta può fornire implementazioni complete o parziali
Modificatori	Un'interfaccia definisce solo metodi pubblici	Una classe astratta può contenere metodi con diverse visibilità
Core vs. Periferiche (Omogeneità)	Un'interfaccia è solitamente usata per definire proprietà <i>non-core</i> e condivise da classi diverse (anche molto diverse tra loro)	Una classe astratta è solitamente usata per definire proprietà <i>core</i> e condivise dalle sottoclassi (omogenee tra loro)

Parametri interfaccia

```
import java.util.Random;

public class GeometryDemoInterface {

    private static void randomColor(IColorable s){

        Random r = new Random();

        Color nc = Color.values()[r.nextInt(Color.values().length)];

        s.setColor(nc);
    }

    public static void main(String[] args) {

        Rectangle rect1 = new Rectangle(2, 3);
        Circle circle1 = new Circle(5);

        randomColor(rect1);
        randomColor(circle1);

    }
}
```

OK se entrambe le classi implementano IColorable

Estendere interfacce

- Anche le interfacce possono essere **estese**
- Si può definire un'interfaccia da una esistente, aggiungendo **nuovi metodi**
- Come per le classi si usa la keyword **extends**
- **NOTA:** mentre le classi non supportano ereditarietà multipla, le interfacce possono estendere **più interfacce**

Estendere interfacce

```
public interface IColorableTransparent extends IColorable
{
    public void setTransparency(double t);
    public double getTransparency();
}
```

```
public class Rectangle implements IShape,
IColorableTransparent {
    double s1;
    double s2;
    Color c;
    double t;
```

```
...
```

Estendere interfacce

```
public class Rectangle implements IShape,  
IColorableTransparent {
```

```
... // campi, costruttori e metodi di Shape omessi
```

```
@Override  
public Color getColor() {return this.c;}  
  
@Override  
public void setColor(Color c) {this.c = c;}
```

```
@Override  
public void setTransparency(double t) {this.t = t;}  
  
@Override  
public double getTransparency() {return this.t;}
```

Esempio: interfaccia Set

Interface Set<E>

Type Parameters:

E - the type of elements maintained by this set

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

NavigableSet<E>, SortedSet<E>

All Known Implementing Classes:

AbstractSet, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

```
public interface Set<E>
    extends Collection<E>
```

<https://docs.oracle.com/javase/7/docs/api/java/util/Set.html>

Esempio: Comparable

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

ChronoLocalDate, ChronoLocalDateTime<D>, Chronology, ChronoZonedDateTime<D>, Delayed, Name, Path, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:

AbstractChronology, AbstractRegionPainter.PaintContext.CacheMode, AccessMode, AclEntryFlag, AclEntryPermission, AclEntryType, AddressingFeature.Responses, Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, CertPathValidatorException.BasicReason, Character, Character.UnicodeScript, CharBuffer, Charset, ChronoField, ChronoUnit, ClientInfoStatus, CollationKey, Collector.Characteristics, Component.BaselineResizeBehavior, CompositeName, CompoundName, CRLReason, CryptoPrimitive, Date, Date, DayOfWeek, Desktop.Action, Diagnostic.Kind, Dialog.ModalExclusionType, Dialog.ModalityType, DocumentationTool.Location, Double, DoubleBuffer, DropMode, Duration, ElementKind, ElementType, Enum, File, FileTime, FileVisitOption, FileVisitResult, Float, FloatBuffer, FormatStyle, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType, GraphicsDevice.WindowTranslucency, GregorianCalendar, GroupLayout.Alignment, HijrahChronology, HijrahDate, HijrahEra, Instant, IntBuffer, Integer, IsoChronology, IsoEra, JapaneseChronology, JapaneseDate, JavaFileObject.Kind, JDBCType, JTable.PrintMode, KeyRep.Type, LayoutStyle.ComponentPlacement, LdapName, LinkOption, LocalDate, LocalDateTime, Locale.Category, Locale.FilteringMode, LocalTime, Long, LongBuffer, MappedByteBuffer, MemoryType, MessageContext.Scope, MinguoChronology, MinguoDate, MinguoEra, Modifier, Month, MonthDay, MultipleGradientPaint.ColorSpaceType, MultipleGradientPaint.CycleMethod, NestingKind, Normalizer.Form, NumericShaper.Range, ObjectName, ObjectOutputStreamField, OffsetDateTime, OffsetTime, PKIXReason, PKIXRevocationChecker.Option, PosixFilePermission, ProcessBuilder.Redirect.Type, Proxy.Type, **PseudoColumnUsage**, Rdn, ResolverStyle, Resource.AuthenticationType, RetentionPolicy, RoundingMode, RowFilter.ComparisonType, RowIdLifetime, RowSorterEvent.Type, Service.Mode, Short, ShortBuffer, SignStyle, SOAPBinding.ParameterStyle, SOAPBinding.Style, ~~SOAPBinding.Use~~, ~~SortOrder~~, ~~SourceVersion~~, ~~SSLEngineResult~~, ~~HandshakeStatus~~, ~~SSLEngineResult~~, ~~Status~~, ~~StandardCopyOption~~

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

Esempio: Comparable

- **Comparable** è una classe **built-in** molto usata per imporre un **ordinamento** tra oggetti della classe che la implementa
- Definisce un solo metodo:
public int compareTo(Object x)
- Quali proprietà definiscono una relazione d'ordine? Qual'è la differenza tra ordine parziale e totale?

Esempio: Comparable

- Se una classe implementa **compareTo(x)** deve ritornare:
 - Un **numero** < 0 , se l'oggetto di invocazione “**precede x**” nell'ordinamento
 - **0**, se l'oggetto di invocazione è “**uguale a**” **x**
 - Un **numero** > 0 , se l'oggetto di invocazione “**segue x**” nell'ordinamento
- A cosa serve un'oggetto Comparable?

Esempio: Comparable book

```
public class Book implements Comparable {  
  
    private String title;  
    private Integer pubYear;  
  
    // Costruttore e metodi setter/getter omessi  
  
    public int compareTo (Object o) {  
        return 0;  
    }  
}
```


Esempio: Comparable book

```
import java.util.Arrays;

public class ComparablesDemo {


    public static void main(String[] args) {

        Book[] books = new Book[3];

        books[0] = new Book("Harry Potter ...", 1997);
        books[1] = new Book("The Lord of the Rings", 1954);
        books[2] = new Book("Don Quixote", 1605);

        Arrays.sort(books);

        for (Book b : books) {
            System.out.println(b);
        }
    }
}
```



```
Book [title=Harry Potter ..., pubYear=1997]
Book [title=The Lord of the Rings, pubYear=1954]
Book [title=Don Quixote, pubYear=1605]
```

Esempio: Comparable book

```
public int compareTo(Object o) {  
    if ( (o != null) && (o instanceof Book)) {  
        Book nb = (Book) o;  
  
        if (this.pubYear > nb.pubYear)  
            return 1;  
        else if (this.pubYear < nb.pubYear)  
            return -1;  
        else  
            return 0;  
  
        //return (this.pubYear.compareTo(nb.getPubYear()));  
        //return (this.title.compareTo(nb.getTitle()));  
  
    }  
    return -1;    // default se si confronta  
                //Book con null o altre classi  
}
```

Cosa fanno?

Altri ordinamenti

- Crescente per anno di pubblicazione

```
Book [title=Don Quixote, pubYear=1605]  
Book [title=The Lord of the Rings, pubYear=1954]  
Book [title=Harry Potter ..., pubYear=1997]
```

- Per titolo in ordine alfabetico

```
Book [title=Harry Potter ..., pubYear=1997]  
Book [title=Don Quixote, pubYear=1605]  
Book [title=The Lord of the Rings, pubYear=1954]
```