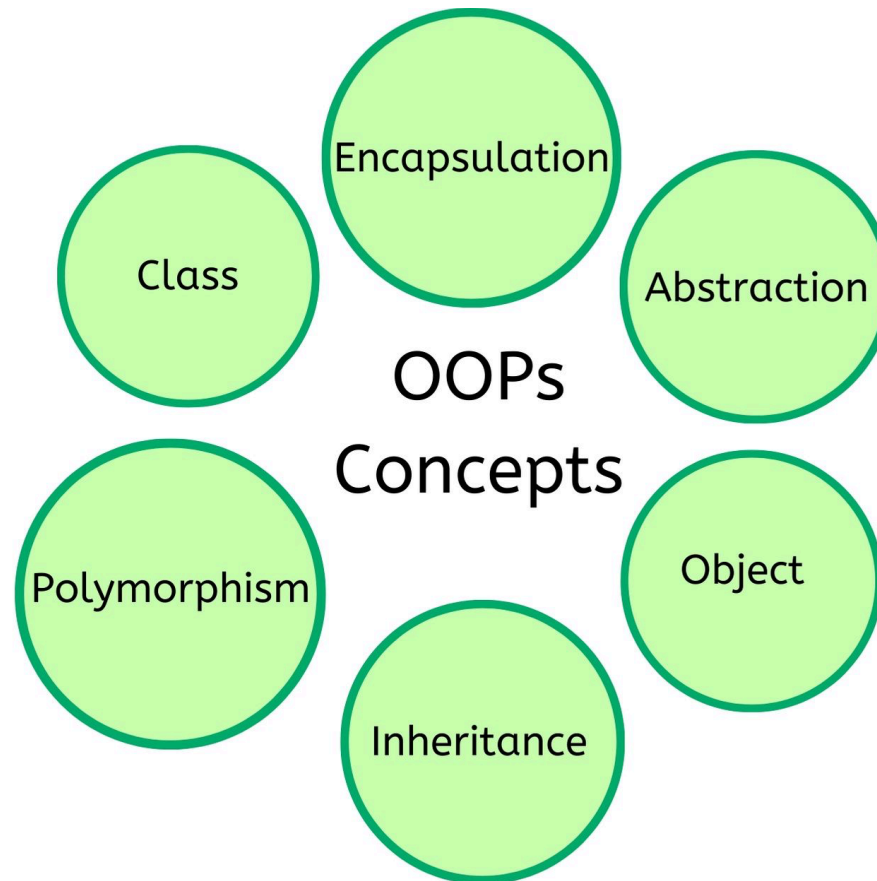




Riflessione:

Object Oriented Programming Obiettivi e principi di design



OOP - obiettivi

- **Robustezza**

- Vogliamo scrivere programmi capaci di gestire situazioni **inaspettate** (per es. input inattesi)
- Requisito particolarmente importante in applicazioni “life-critical”

- **Adattabilità**

- Vogliamo scrivere programmi capaci di **evolvere** (per es. girare su diverse architetture o avere nuove funzionalità)
- Un concetto correlato è quello di **portabilità**

- **Riusabilità**

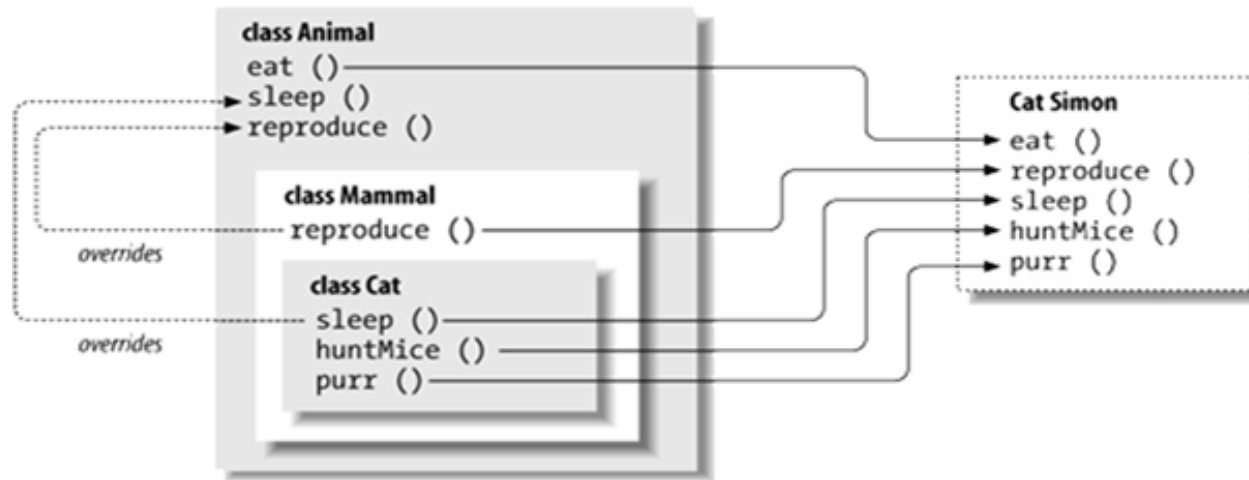
- Vogliamo che il nostro codice sia utilizzabile come **componente** di diversi sistemi in varie applicazioni
- Deve essere chiaro cosa il nostro codice fa, e cosa non fa

OOP - principi di design

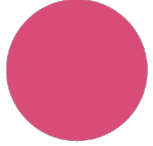
- **Astrazione**
 - “Distillare” i **concetti** che meglio rappresentano un oggetto o un sistema
- **Information hiding**
 - Nascondere l'informazione a utenti esterni, lasciando vedere solo **l'interfaccia**
 - In questo modo parti di un programma possono cambiare senza effetti sulle altre
- **Modularità**
 - Organizzare un sistema software in **componenti funzionali** separate
- Tutti questi principi di design
 - Sono supportati in un linguaggio di programmazione **OO**
 - Aiutano a ottenere robustezza, adattabilità, riusabilità

OOP - Paradigmi

- Abbiamo esaminato i **tre principali paradigmi** di programmazione che caratterizzano un linguaggio OO
 - **Classi, oggetti, incapsulamento**
 - L'informazione viene nascosta dentro scatole nere (le classi), e l'accesso a essa è controllato
 - **Ereditarietà**
 - Una classe può essere estesa da sottoclassi, che ne ereditano le funzioni e le specializzano
 - **Polimorfismo**
 - Il tipo di una variabile non determina completamente il tipo dell'oggetto a cui essa si riferisce



La superclasse universale Object



La classe *Object*

- Sappiamo già che ogni classe di Java è (in maniera diretta o indiretta) **sottoclasse di Object**
 - Quindi ogni classe di Java **eredita tutti i metodi** della classe **Object**
- Alcuni dei più utili metodi di **Object** sono i seguenti

Metodo	Obiettivo
<code>String toString()</code>	Restituisce una stringa che descrive l'oggetto
<code>boolean equals(Object otherObject)</code>	Verifica se l'oggetto è uguale a un altro
<code>Object clone()</code>	Crea una copia completa dell'oggetto

- Ma perché siano davvero utili, nella maggior parte dei casi bisogna **sovrascriverli**.

Sovrascrivere i metodi `toString` ed `equals`

Metodi “magici”

- Il metodo **println**, è in grado di ricevere come parametro un oggetto **di qualsiasi tipo**
 - Ora siamo in grado di comprendere questa “magia”
 - qualsiasi riferimento può sempre essere convertito automaticamente in un riferimento di tipo **Object**

```
public void println(Object obj) //invocato per oggetti generici
{   println(obj.toString()); }
public void println(String s)   // invocato per le stringhe
{   ... }
```

- Ma un esemplare di **String** è anche di tipo **Object**...
 - come viene scelto, tra i metodi sovraccarichi, quello giusto quando si invoca **println** con una stringa?
 - Il compilatore cerca sempre di “**fare il minor numero di conversioni**”
 - viene usato, quindi, **il metodo “più specifico”**

Il metodo toString

- Passando un oggetto qualsiasi a **System.out.println** si visualizza la sua descrizione testuale standard
 - println** invoca **toString** dell'oggetto, e l'invocazione è possibile perché **tutte le classi** hanno il metodo **toString**, eventualmente ereditato da **Object**

```
BankAccount a = new BankAccount(1000);
System.out.println(a);
```

- Il metodo **toString** di una classe viene invocato implicitamente anche per **concatenare** un oggetto con una stringa:

```
BankAccount acct = new BankAccount();
String s = "Conto " + acct;
```

- La seconda riga viene interpretata dal compilatore come se fosse stata scritta così:

```
String s = "Conto " + acct.toString();
```

Il metodo toString

- Il metodo **toString** della classe **Object** ha la firma

```
public String toString()
```

- L'invocazione di questo metodo **per qualsiasi oggetto** ne restituisce la **descrizione testuale standard**
 - il nome della classe** seguito dal carattere **@** e dallo **hashcode** dell'oggetto
 - Che è un numero univocamente determinato dall'indirizzo in memoria dell'oggetto stesso



```
BankAccount a = new BankAccount(1000) ;  
System.out.println(a) ;
```

```
BankAccount@111f71
```

- In generale la descrizione testuale standard non è particolarmente utile
 - È più utile ottenere una stringa di testo contenente **informazioni sullo stato** dell'oggetto in esame

Sovrascrivere il metodo `toString`

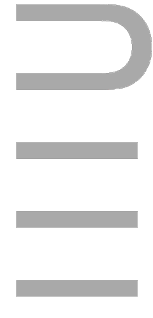
- Sovrascrivere il metodo **`toString`** nelle classi che si scrivono è considerato un buon stile di programmazione
- **`toString`** dovrebbe sempre produrre una stringa contenente tutte le **informazioni di stato dell'oggetto**
 - Il valore di tutte le sue variabili di esemplare
 - Il valore di variabili statiche non costanti della classe
- Questo stile di programmazione è molto utile per il **debugging** ed è usato nella libreria standard
 - Esempio: descrizione testuale di **BankAccount**

```
System.out.println(anAccount);
```

```
BankAccount[balance=1500]
```

- Bisogna **sovrascrivere** `toString` in **BankAccount**

Esempio: toString in BankAccount

- 
- Stile adottato nella libreria standard:
 - **toString** crea una stringa contenente il **nome** della classe seguito dai **valori dei campi** di esemplare racchiusi fra parentesi quadre.

```
public class BankAccount
{
    ...
    public String toString()
    {
        return "BankAccount[balance=" + balance + "];"
    }
}
```

- **Problema:** devo sovrascrivere **toString** anche per le **sottoclassi**, per vedere stampato il nome di classe corretto

Esempio: toString in BankAccount

- È possibile evitare di scrivere esplicitamente il nome della classe all'interno del metodo
 - Si può usare il metodo **getClass**, che restituisce un **oggetto di tipo classe**
 - E poi invocare il metodo **getName** sull'oggetto di tipo classe, per ottenere il **nome della classe**



```
public String toString()  
{  
    return getClass().getName() +  
           "[balance=" + balance + "];"  
}
```

- **toString** visualizza il nome corretto della classe anche quando viene invocato su un oggetto di una sottoclasse

Esempio: toString in SavingsAccount

- Ovviamente, se si vogliono visualizzare i valori di nuovi campi di esemplare di una sottoclasse bisogna **sovrascrivere toString**
 - Bisogna invocare **super.toString** per ottenere i valori dei campi della superclasse

```
public class SavingsAccount extends BankAccount
{
    ...
    public String toString()
    {
        return super.toString() +
               "[interestRate=" + interestRate + "]";
    }
    ...
}
```

Il metodo equals

- Il metodo **equals** della classe **Object** ha la firma

```
public boolean equals(Object otherObject)
```

- L'invocazione di questo metodo restituisce
 - true** se l'oggetto su cui viene invocato coincide con l'oggetto passato come parametro esplicito
 - Ovvero se **this == otherObject**
 - false** altrimenti
- In particolare equals restituisce un valore **true** se e solo se gli **hashcode** dell'oggetto **this** e dell'oggetto **otherObject** sono uguali
- In generale questo comportamento non è molto utile
 - È più utile ottenere una informazione booleana che dica se **gli stati** degli oggetti in esame coincidono



Sovrascrivere il metodo equals

- Sovrascrivere il metodo **equals** nelle classi che si scrivono è considerato un buon stile di programmazione
- **equals** dovrebbe sempre dare informazione sulla coincidenza degli **stati dei due oggetti**
 - Ovvero restituire true se e solo se i valori di tutte le variabili di esemplare dei due oggetti coincidono
- Questo stile di programmazione è molto utile per il confronto di oggetti ed è usato nella libreria standard
 - Esempio: confronto di uguaglianza tra **BankAccount**

```
BankAccount a1 = new BankAccount(1000);  
BankAccount a2 = new BankAccount(a1.getBalance());  
System.out.println("Confronto uguaglianza: " + a1.equals(a2));
```

Confronto uguaglianza: true

- Bisogna **sovrascrivere equals** in **BankAccount**

Esempio: equals in BankAccount

- Confronto tra i campi di esemplare dei due oggetti

```
public class BankAccount
{
    ...
    public boolean equals(Object otherObject)
    {
        BankAccount otherAcct = (BankAccount) otherObject;
        return balance == otherAcct.balance;
    }
}
```

- **Problema:** nella firma di **equals** il parametro esplicito è di tipo **Object**, quindi è necessario **eseguire un cast** per accedere ai campi di esemplare di **otherObject**
- Se si scrive un metodo
public boolean equals(BankAccount acct)
non si sovrascrive il metodo equals di **Object**!

Ereditarietà e controllo di accesso

Controllo di accesso

- Java fornisce **quattro livelli** per il controllo di accesso a metodi, campi, e classi
 - **public**
 - **private**
 - **protected**
 - **package**
- Finora noi abbiamo sempre usato solo i primi due
 - Ora siamo in grado di comprendere anche gli altri due
 - In particolare lo specificatore di accesso **protected**

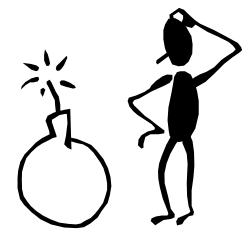
Accesso *package*

- Un membro di classe (o una classe) senza specificatore di accesso ha **di default** una impostazione di accesso **package** (accesso **di pacchetto**)
 - I metodi di classi nello **stesso pacchetto** vi hanno accesso
 - Può essere una buona impostazione per le classi, **non lo è** per le **variabili**, perché si viola l'incapsulamento
- **Errore comune**: dimenticare lo specificatore di accesso per una variabile di esemplare

```
public class Window extends Container
{   String warningString;
    ...
}
```

- È un **rischio** per la sicurezza: un altro programmatore può realizzare una classe nello stesso pacchetto e **ottenere l'accesso** al campo di esemplare

Accesso *protected*



```
public class BankAccount
{
    ...
    protected double balance;
}
```

- Abbiamo visto che una sottoclasse non può accedere a campi private ereditati dalla propria superclasse
- Il progettista della superclasse può rendere accessibile in modo **protected** un campo di esemplare
 - **Tutte** le sottoclassi vi hanno accesso
- Attenzione: è una violazione dell'incapsulamento
 - Il progettista della superclasse **non ha controllo** sui progettisti di eventuali sottoclassi
 - Diventa **difficile modificare** variabili protected perché potrebbe esserci una sottoclasse che ne fa uso
- Anche i metodi possono essere definiti **protected**
 - possono essere invocati solo nella classe in cui sono definiti e nelle classi da essa derivate

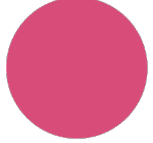
Ereditarietà e il modificatore di accesso final

Classi e metodi final

- Sappiamo che la parola chiave **final** viene usata nella definizione di una variabile per impedire successive assegnazioni di valori
- Anche **metodi** possono essere dichiarati **final**
 - Un metodo **final** **non può essere sovrascritto** da sottoclassi
- **Esempio:** un metodo per verificare una password

```
public class SecureAccount extends BankAccount
{
    ...
    public final boolean checkPassword(String password)
    {
        ...
    }
}
```

- In questo modo la sicurezza è garantita
 - **nessuno** può sovrascriverlo con un metodo che restituisca sempre **true**



Classi e metodi final

- Anche **classi** possono essere dichiarati **final**
 - Una classe dichiarata **final** **non può essere estesa**
- Esempio: la classe **String** è una classe immutabile
 - Gli oggetti di tipo **String** **non** possono essere modificati da nessuno dei loro metodi
- La firma della classe **String** nella libreria standard è:

```
public final class String
{
    ...
}
```

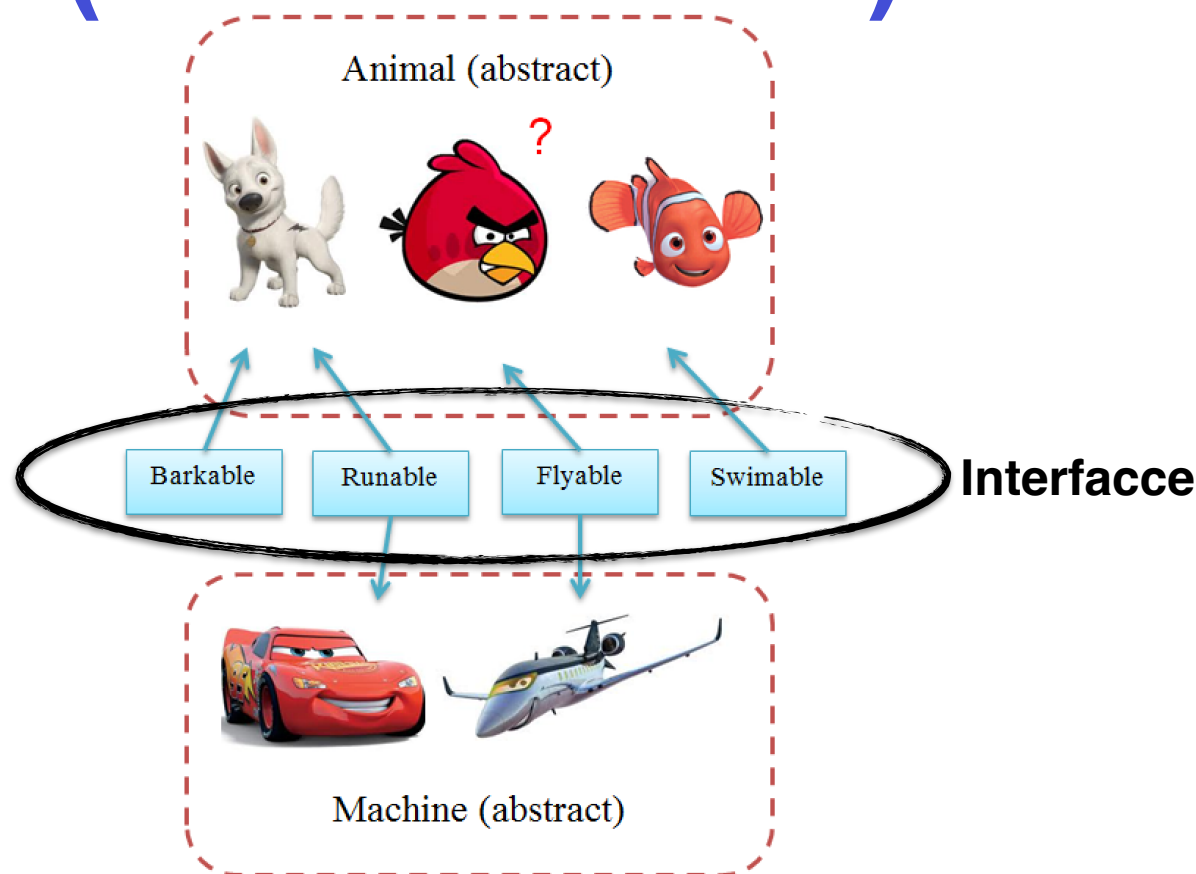
- **Non** si possono creare sottoclassi di **String**
 - Di conseguenza una variabile di tipo **String** deve contenere il riferimento a un oggetto di tipo **String**, non di una sua sottoclasse (che non esiste)



È tutto chiaro? ...

1. Quale è una comune motivazione che porta a definire campi di esemplare con accesso di pacchetto?
2. Se una classe dotata di costruttore pubblico ha accesso di pacchetto, chi ne può costruire esemplari?

Interfacce (capitolo 10) e l'interfaccia Comparable (sezione 10.3)



Realizzare una proprietà astratta

- Torniamo al problema già enunciato: ordinare oggetti generici
 - Affinché i suoi esemplari possano essere ordinati, una classe deve **definire un metodo** adatto a confrontare esemplari della classe, con lo stesso comportamento di **compareTo**
- Gli oggetti della classe diventano **confrontabili**
 - gli algoritmi di ordinamento/ricerca non hanno bisogno di conoscere alcun particolare del criterio di confronto tra gli oggetti
 - basta che gli oggetti siano tra loro **confrontabili**

Scrivere una interfaccia

- Sintassi:

```
public interface NomeInterfaccia
{
    firme di metodi
}
```
- La definizione di una **interfaccia** in Java serve per **definire un comportamento astratto**
 - Una interfaccia deve poi essere **realizzata** (implementata) da una classe che dichiara di avere tale comportamento
- Definire un'interfaccia è simile a definire una classe
 - si usa la parola chiave **interface** al posto di **class**
- Un'interfaccia può essere vista come una classe **ridotta**
 - **non** può avere costruttori
 - **non** può avere variabili di esempio
 - **contiene le firme** di uno o più metodi non statici (definiti implicitamente **public**), ma **non può definirne il codice**

Implementare una interfaccia

- Esempio:

- L'interfaccia **Comparable**

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

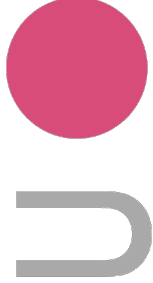
- Se una classe intende realizzare concretamente un comportamento astratto definito in un'interfaccia, deve
 - Dichiarare di **implementare** l'interfaccia
 - **Definire i metodi** dell'interfaccia, con la stessa firma
 - Una classe può implementare più di una interfaccia: la relazione implements non deve rispettare la regola dell'ereditarietà singola!!!

```
public class BankAccount implements Comparable
{
    ...
    public int compareTo(Object other)
    { //other è un Object perché questa è la firma in
      //Comparable. Però facciamo un cast a BankAccount...
      BankAccount acct = (BankAccount) other;
      if (balance < acct.balance) return -1;
      if (balance > acct.balance) return 1;
      return 0;
    }
    ...
}
```

Implementare una interfaccia

- Una interfaccia può estendere una o più interfacce (non classi), indicate dopo la parola chiave `extends`
- Per le interfacce non vale la restrizione di "ereditarietà singola" che vale per le classi.

```
public interface <IntName> [extends <IntName1>, <IntName2>, ...] {  
    <tipo1> <nome-var1> = <val1>;  
    ...  
    <tipoN> <nome-varN> = <valN>;  
    <tipo-res1> <metodo1> ( <arg1> );  
    ...  
    <tipo-resM> <metodoM> ( <argM> );  
}
```



Implementare una interfaccia

- Ogni classe estende (extends) una sola altra classe (Object se non specificata);
- Una interfaccia può estendere (extends) una o più interfacce:

```
public interface <Int> extends <Int1>,<Int2>, ...{  
    ...  
}
```

- La gerarchia di ereditarietà singola delle classi e la gerarchia di ereditarietà multipla delle interfacce sono completamente disgiunte.
- Una classe può implementare (implements) una o più interfacce:

```
public class <nomeClasse> extends <nomeSuperClasse>  
    implements <Int1>, <Int2>, ..., <Intn> {  
    ...  
}
```

- In questo caso <nomeClasse> deve fornire una realizzazione per tutti i metodi delle interfacce <Int1>, <Int2>, ... che implementa, nonché per i metodi di eventuali super-interfacce da cui queste ereditano

Facciamo evolvere un'interfaccia

- Sino a Java8 tutti i metodi delle interfacce dovevano essere astratti
- Java 8 consente di definire nelle interfacce metodi statici che funzionano esattamente come i metodi statici definiti nelle classi
- Un metodo statico definito in un'interfaccia non opera su un oggetto e il suo scopo dovrebbe essere correlato a quello dell'interfaccia in cui è contenuto

```
public interface Measurable {  
    // un metodo astratto:  
    double getMeasure();  
    // un metodo statico:  
    static double average(Measurable[] objects)  
    {  
        . . . // implementazione  
    }
```

- Per invocare tale metodo si scrive il nome dell'interfaccia che lo contiene:

```
double meanArea = Measurable.average(countries);
```


Facciamo evolvere un'interfaccia

- Un metodo predefinito o di default (default o defender method) in un'interfaccia è un metodo **non** statico di cui viene definita anche l'implementazione
- Una classe che implementa l'interfaccia erediterà il comportamento predefinito del metodo oppure lo potrà sovrascrivere: il fatto che in un'interfaccia venga predefinita un'implementazione per un metodo alleggerisce il lavoro necessario per realizzare una classe che la implementi
- Per esempio, l'interfaccia Measurable potrebbe dichiarare un metodo smallerThan per verificare se un oggetto ha una misura inferiore a quella di un altro, azione utile per disporre oggetti in ordine di misura crescente, come predefinito:

```
public interface Measurable
{ double getMeasure(); // un metodo astratto
  default boolean smallerThan(Measurable other)
  { return getMeasure() < other.getMeasure();
  }
}
```

- Una classe che voglia implementare l'interfaccia Measurable deve soltanto definire il metodo getMeasure, ereditando automaticamente il metodo smallerThan: un meccanismo che può essere molto utile.

Conversione di tipo tra classi e interfacce



Usare una interfaccia

- In generale, valgono le regole di conversione viste nell'ambito dell'ereditarietà
 - Una **interfaccia** ha lo stesso ruolo di una **superclasse**
 - Una classe che **implementa** un'interfaccia ha lo stesso ruolo di una sottoclasse che **estende** la superclasse
- È lecito definire una variabile il cui tipo è un'interfaccia

```
Comparable c; // corretto
```
- Ma **non è possibile** costruire oggetti da un'interfaccia

```
new Comparable(); // ERRORE IN COMPILAZIONE
```

 - Le interfacce **non** hanno campi di esemplare nè costruttori
 - Allora a che serve avere una variabile oggetto il cui tipo è quello di una interfaccia?

Conversione da classe a interfaccia

- Una variabile oggetto di tipo **Comparable** può puntare a un oggetto di una classe che realizza **Comparable**

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

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

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

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

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

Conversione da interfaccia a classe

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

```
Comparable c = new BankAccount(10);  
double saldo = c.getBalance(); //ERRORE! Come faccio?
```

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

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

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

Polimorfismo e interfacce

Polimorfismo e interfacce

```
Comparable x;  
...  
if (...) x = new BankAccount(1000);  
else x = new String("");  
...  
if (x.compareTo(y) > 0) ...
```

- Possiamo invocare il metodo **compareTo** perché **x** è di tipo **Comparable**
 - Ma **quale** metodo **compareTo**?
 - Quello scritto in **BankAccount** o quello di **String**?
- Il compilatore non ha questa informazione
 - Il tipo di oggetto a cui **x** si riferisce dipende dal flusso di esecuzione
 - In fase di **esecuzione** la **JVM** determina il tipo di oggetto a cui **x** si riferisce e applica il metodo corrispondente

Polimorfismo e interfacce

```
Comparable x;
...
if (...) x = new BankAccount(1000);
else x = new String("");
...
if (x.compareTo(y) > 0) ...
```

- Se la condizione dell'if è vera, **x** contiene un riferimento a un oggetto che, **in quel momento dell'esecuzione** è di tipo **BankAccount**!
- E l'interprete Java lo sa (il compilatore no)
 - secondo la semantica di Java, viene invocato il metodo **compareTo** scritto in **BankAccount**



Polimorfismo e interfacce

- Il tipo di una variabile **non determina in modo completo** il tipo dell'oggetto a cui essa si riferisce
- Come per l'ereditarietà, questa semantica si chiama **polimorfismo**
 - La stessa operazione (**compareTo**) può essere svolta in modi diversi, a seconda dell'oggetto a cui ci si riferisce
- L'invocazione di un metodo **è sempre determinata dal tipo dell'oggetto** effettivamente usato come parametro implicito, e **NON** dal tipo della variabile oggetto
 - La variabile oggetto (interfaccia) ci dice **cosa si può fare** con quell'oggetto (cioè quali metodi si possono utilizzare)
 - L'oggetto (appartenente a una classe) ci dice **come farlo**

Ordinamento di oggetti e l'interfaccia Comparable

L'interfaccia Comparable

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

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

Ordinamento di oggetti

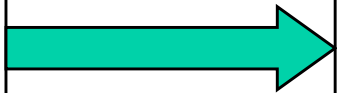
- Tutti i metodi di ordinamento e ricerca che abbiamo visto per array di numeri interi possono essere riscritti per array di oggetti **Comparable**
 - Basta usare le seguenti “traduzioni”

```
if (a < b)
```

```
if (a > b)
```

```
if (a == b)
```

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



```
if (a.compareTo(b) < 0)
```

```
if (a.compareTo(b) > 0)
```

```
if (a.compareTo(b) == 0)
```

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

SelectionSort per oggetti Comparable

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

Ordinamento di oggetti

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

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

- Se** **BankAccount** realizza l'interfaccia **Comparable**, l'array di riferimenti viene convertito in maniera automatica

Scrivere metodi compareTo

Il metodo `compareTo`

- Il metodo **`compareTo`** deve definire una **relazione di ordine totale**, ovvero deve avere queste proprietà
 - Antisimmetrica**: $a.compareTo(b) \leq 0 \Leftrightarrow b.compareTo(a) \geq 0$
 - Riflessiva**: $a.compareTo(a) = 0$
 - Transitiva**: $a.compareTo(b) \leq 0$ e $b.compareTo(c) \leq 0$
 - Implica $a.compareTo(c) \leq 0$
- `a.compareTo(b)`** può restituire qualsiasi valore negativo per segnalare che **a** precede **b**

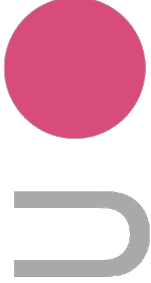
```
if (a.compareTo(b) == -1) //errore logico!  
if (a.compareTo(b) < 0)   // giusto!
```


I metodi compareTo ed equals

- Una classe che implementa **Comparable** ha i metodi
 - **compareTo** (per implementare **Comparable**)
 - **equals** (**ereditato** da **Object**)
- Il metodo **compareTo** definito in una classe dovrebbe sempre essere **consistente** con il metodo **equals**:
 - (**e1.compareTo(e2) == 0**)
 - dovrebbe sempre avere lo **stesso valore booleano** di
 - **e1.equals(e2)** per ogni **e1** e **e2** della classe
- Il metodo **equals** in **Object** è:

```
public boolean equals(Object obj)
{   return (this == obj); }
```

 - Quindi una classe che implementa **Comparable** **dovrebbe sempre** sovrascrivere **equals**...
 - Non sempre noi lo faremo



Riassunto: la classe ArrayAlgs

- Ecco la classe **ArrayAlgorithms** con tutti i metodi statici da noi sviluppati
 - Tutti i metodi per l'ordinamento e ricerca agiscono su array di tipo **Comparable[]** e usano il metodo **compareTo** per effettuare confronti tra oggetti
 - **binarySearch** potrebbe usare il metodo **equals** per verificare l'uguaglianza
 - Ma per avere comportamenti **consistenti** le classi che implementano **Comparable** devono sovrascrivere **equals**
- Per ordinare (o cercare valori in) array **numerici** possiamo usare **classi involucro** (**Integer**, **Double**,...)
- Compilando questa classe vengono generati alcuni messaggi di **warning**, che ignoreremo

La classe ArrayAlgs

```
public class ArrayAlgs
{ // riportiamo solo i metodi per array di oggetti Comparable
  // i metodi per array di interi sono obsoleti

  //----- selectionSort per oggetti Comparable -----
  public static void selectionSort(Comparable[] v, int vSize)
  {
    for (int i = 0; i < vSize - 1; i++)
    {
      int minPos = findMinPos(v, i, vSize-1);
      if (minPos != i) swap(v, minPos, i);
    }
  }
  private static void swap(Comparable[] v, int i, int j)
  {
    Comparable temp = v[i];
    v[i] = v[j];
    v[j] = temp;
  }
  private static int findMinPos(Comparable[] v, int from, int to)
  {
    int pos = from;
    Comparable min = v[from];
    for (int i = from + 1; i <= to; i++)
      if (v[i].compareTo(min) < 0)
      {
        pos = i;
        min = v[i];
      }
    return pos;
  }
}
```

//continua

La classe ArrayAlgs

```

//-----mergeSort per oggetti Comparable -----//continua
public static void mergeSort(Comparable[] v, int vSize)
{
    if (vSize < 2) return; // caso base
    int mid = vSize / 2; //dividiamo circa a meta'
    Comparable[] left = new Comparable[mid];
    Comparable[] right = new Comparable[vSize - mid];
    System.arraycopy(v, 0, left, 0, mid);
    System.arraycopy(v, mid, right, 0, vSize-mid);
    mergeSort(left, mid); // passi ricorsivi
    mergeSort(right, vSize-mid);
    merge(v, left, right);
}

private static void merge(Comparable[] v, Comparable[] v1,
                           Comparable[] v2)
{
    int i = 0, i1 = 0, i2 = 0;
    while (i1 < v1.length && i2 < v2.length)
        if (v1[i1].compareTo(v2[i2]) < 0) v[i++] = v1[i1++];
        else v[i++] = v2[i2++];
    while (i1 < v1.length)
        v[i++] = v1[i1++];
    while (i2 < v2.length)
        v[i++] = v2[i2++];
}
//continua

```

La classe ArrayAlgs

```
// ---- insertionSort per oggetti Comparable //continua
public static void insertionSort(Comparable[] v, int vSize)
{
    for (int i = 1; i < vSize; i++)
    {
        Comparable temp = v[i]; //elemento da inserire
        int j;
        for (j = i; j > 0 && temp.compareTo(v[j-1]) < 0; j--)
            v[j] = v[j-1];
        v[j] = temp;      } // inserisci temp in posizione
    }
//----- ricerca (binaria) per oggetti Comparable
public static int binarySearch(Comparable[] v, int vSize,
                               Comparable value)
{
    return binSearch(v, 0, vSize-1, value);
}
private static int binSearch(Comparable[] v, int from, int to,
                              Comparable value)
{
    if (from > to) return -1; // el. non trovato
    int mid = (from + to) / 2; // circa in mezzo
    Comparable middle = v[mid];
    if (middle.compareTo(value) == 0) return mid; //trovato
    else if (middle.compareTo(value) < 0) //cerca a destra
        return binSearch(v, mid + 1, to, value);
    else // cerca a sinistra
        return binSearch(v, from, mid - 1, value);
    }
}
```



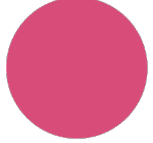
Approfondimenti

Ordinamento e ricerca “di libreria”

- La libreria standard fornisce, per l'ordinamento e la ricerca, alcuni metodi statici in **java.util.Arrays**
 - un metodo **sort** che ordina array di tutti i tipi fondamentali e array di **Comparable**

```
String[] ar = new String[10];
...
Arrays.sort(ar);
```

- un metodo **binarySearch** che cerca un elemento in array di tutti i tipi fondamentali e in array di **Comparable**
 - restituisce la posizione come numero intero



Comparable e Java >= 5.0



- **IMPORTANTE** per gli esercizi di programmazione
- In seguito all'introduzione in Java 5.0 dei **tipi generici**, l'utilizzo di **Comparable** induce il compilatore all'emissione di particolari "segnalazioni"

Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

- il compilatore avvisa che nell'invocazione di **compareTo** non è stato verificato che gli oggetti **Comparable** da confrontare siano davvero esemplari della stessa classe.
- Possiamo ignorare questi "warning": non sono errori, il compilatore produce comunque i/il file di bytecode

```
javac -nowarn MyClass.java
```

- Con l'opzione **-nowarn** si eliminano le segnalazioni

Interfaccia Comparable parametrica

- Java >=5.0 fornisce un utile strumento: l'interfaccia **Comparable parametrica**, che permette di definire il tipo di oggetto nel parametro esplicito del metodo **compareTo**

```
public interface Comparable<T>
{
    int compareTo(T o);
}
```

```
public class BankAccount implements Comparable<BankAccount>
{
    public int compareTo(BankAccount obj)
    {
        if (balance < obj.balance) return -1; //non serve il cast
        if (balance > obj.balance) return 1;
        return 0;
    }
}
```

- T rappresenta una **classe generica**

Interfaccia Comparable parametrica

- In questo modo, errori dovuti a confronti tra oggetti di classi diverse (e quindi non confrontabili) vengono intercettati dal compilatore

```
BankAccount acct1 = new BankAccount(10);  
String s = "ciao";  
...  
Object acct2 = s; // errore del programmatore  
if (acct1.compareTo(acct2) < 0)
```

- In compilazione:

`compareTo(BankAccount) cannot be applied to
<java.lang.Object>`

- Se non si usa l'interfaccia **Comparable** parametrica, questo errore non viene rilevato in compilazione e si genera invece una **ClassCastException** in esecuzione

Interfaccia Comparable parametrica

```
public class Mela implements Comparable<Mela>{
    private int grammi;
    private String tipo;

    public Mela(String tipo, int grammi){
        this.tipo = tipo;
        this.grammi = grammi;
    }

    public int compareTo(Mela m){
        return grammi - m.grammi;
    }
}
```

```
public class Pera implements Comparable<Pera>{
    private int grammi;
    private String tipo;

    public Pera(String tipo, int grammi){
        this.tipo = tipo;
        this.grammi = grammi;
    }

    public int compareTo(Pera p){
        return grammi - p.grammi;
    }
}
```



Interfaccia Comparable parametrica

```
public class TestMelaPera{  
  
    public static void main (String [] args){  
        Mela mela = new Mela("golden",135);  
        Pera pera = new Pera("williams",120);  
        int comp = pera.compareTo(mela); // errore di compilazione  
    }  
}
```

Classi e metodi astratti

- Un **metodo astratto** è un metodo che non ha implementazione nella classe in cui è definito
- Si definiscono l'intestazione del metodo e della classe a cui appartiene con la parola chiave **abstract**

```
public abstract class Poligono implements Comparable
{
    ...
    public abstract double area();

    public int compareTo(Object obj)
    {
        Poligono p = (Poligono) obj;
        if ( area() > p.area() ) return 1;
        if ( area() < p.area() ) return -1;
        return 0;
    }
}
```

- Il metodo deve essere realizzato nelle classi che estendono la classe astratta