
Capitolo 9

# Tipi enumerativi, tipi generici e interfacce

# Sommario: Tipi enumerativi, tipi generici e interfacce

# Estensione di una classe

- Vengono definiti come le classi mediante la parola riservata `enum`

# Estensione di una classe

- Vengono definiti come le classi mediante la parola riservata `enum`
- Valori del tipo enumerativo
  - sono definiti all'inizio del corpo del tipo enumerativo
  - separati da una "virgola" (,)
  - la dichiarazione è conclusa dal carattere "punto e virgola"

## Esempio

```
public enum MeseDellAnno {
  // COSTANTI ENUMERATIVE
  GENNAIO, FEBBRAIO, MARZO, APRILE, MAGGIO, GIUGNO, LUGLIO,
  AGOSTO, SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBRE;
}
```

# Estensione di una classe

- Vengono definiti come le classi mediante la parola riservata `enum`
- Valori del tipo enumerativo
  - sono definiti all'inizio del corpo del tipo enumerativo
  - separati da una "virgola" (,)
  - la dichiarazione è conclusa dal carattere "punto e virgola"

**Esempio**

```
public enum MeseDellAnno {
  // COSTANTI ENUMERATIVE
  GENNAIO, FEBBRAIO, MARZO, APRILE, MAGGIO, GIUGNO, LUGLIO,
  AGOSTO, SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBRE;
}
```

- L'ordine di dichiarazione è rilevante, determina:
  - l'ordinamento fra i valori del tipo enumerativo (`compareTo`)
  - l'ordine in cui compaiono nell'array restituito da `values`

# MeseDellAnno

## Esempio

```
public enum MeseDellAnno {
  // COSTANTI ENUMERATIVE
  GENNAIO, FEBBRAIO, MARZO, APRILE, MAGGIO, GIUGNO, LUGLIO,
  AGOSTO, SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBRE;
}
```

# MeseDellAnno

## Esempio

```
public enum MeseDellAnno {
  // COSTANTI ENUMERATIVE
  GENNAIO, FEBBRAIO, MARZO, APRILE, MAGGIO, GIUGNO, LUGLIO,
  AGOSTO, SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBRE;
}
```

## Fase di esecuzione

La JVM crea un oggetto per ogni identificatore indicato e ne memorizza il riferimento nella costante corrispondente

# La classe Enum

Ogni tipo enumerativo è una classe che estende la classe generica `Enum<E extends Enum<E>>` (`java.lang`)

# La classe Enum

Ogni tipo enumerativo è una classe che estende la classe generica `Enum<E extends Enum<E>>` (`java.lang`)

## Metodi definiti in Enum

- `public String name()`
- `public int ordinal()`
- `public int compareTo(E o)`
  Confronta l'oggetto di tipo enumerativo che esegue il metodo con quello fornito come argomento. Restituisce un valore negativo, zero o un valore positivo a seconda che l'oggetto che esegue il metodo preceda, sia uguale o segua quello fornito come argomento, sulla base dell'ordine in cui sono dichiarate le costanti nel tipo enumerativo.

# La classe Enum

Ogni tipo enumerativo è una classe che estende la classe generica `Enum<E extends Enum<E>>` (`java.lang`)

## Metodi definiti in Enum

- `public String name()`
- `public int ordinal()`
- `public int compareTo(E o)`
  Confronta l'oggetto di tipo enumerativo che esegue il metodo con quello fornito come argomento. Restituisce un valore negativo, zero o un valore positivo a seconda che l'oggetto che esegue il metodo preceda, sia uguale o segua quello fornito come argomento, sulla base dell'ordine in cui sono dichiarate le costanti nel tipo enumerativo.

  Il metodo `compareTo` è `final` e dunque non può essere sovrascritto

# Il metodo statico values di Enum¡E extends Enum¡E¿¿

- `public static E[] values()`
  Restituisce l'array array contenente le costanti del tipo enumerativo nell'ordine in cui sono dichiarate.

# Il metodo statico values di Enum¡E extends Enum¡E¿¿

- `public static E[] values()`
  Restituisce l'array array contenente le costanti del tipo enumerativo nell'ordine in cui sono dichiarate.

## Esempio

Il tipo enumerativo `MeseDellAnno` dispone automaticamente del metodo statico

```
MeseDellAnno.values()
```

che restituisce un array di 12 posizioni contenente i riferimenti ai 12 valori del tipo.

# Metodi: ridefinizione di `toString`

```
public String toString() {
  switch (this) {
  case GENNAIO:
    return "Gennaio";
  case FEBBRAIO:
    return "Febbraio";
  case MARZO:
    return "Marzo";
  case APRILE:
    return "Aprile";
  case MAGGIO:
    return "Maggio";
  ...
  default:
    return "";
  }
}
```

# Nuovi metodi

- `public MeseDellAnno precedente()`
  Restituisce il riferimento all'oggetto che rappresenta il mese precedente a quello che esegue il metodo. L'oggetto precedente a `GENNAIO` è quello corrispondente a `DICEMBRE`.

# Nuovi metodi

- `public MeseDellAnno precedente()`
  Restituisce il riferimento all'oggetto che rappresenta il mese precedente a quello che esegue il metodo. L'oggetto precedente a `GENNAIO` è quello corrispondente a `DICEMBRE`.

- `public MeseDellAnno successivo()`
  Restituisce il riferimento all'oggetto che rappresenta il mese successivo a quello che esegue il metodo. L'oggetto successivo a `DICEMBRE` è quello corrispondente a `GENNAIO`.

# Implementazione

```
public MeseDellAnno successivo() {
  return MeseDellAnno.values()[(this.ordinal() + 1) % 12];
}
```

# Implementazione

```
public MeseDellAnno successivo() {
  return MeseDellAnno.values()[(this.ordinal() + 1) % 12];
}



public MeseDellAnno precedente() {
  int prec = this.ordinal() - 1 < 0 ? 11 : this.ordinal() - 1;
  return MeseDellAnno.values()[prec];
}
```

# I metodi `numeroGiorni`

- `public int numeroGiorni()`
  Restituisce il numero dei giorni del mese (28 se l'oggetto che esegue il metodo è `FEBBRAIO`).

# I metodi `numeroGiorni`

- `public int numeroGiorni()`
  Restituisce il numero dei giorni del mese (28 se l'oggetto che esegue il metodo è `FEBBRAIO`).

- `public int numeroGiorni(boolean bisestile)`
  Restituisce il numero dei giorni del mese; nel caso di febbraio, se l'argomento è `true` restituisce 29, se è `false` restituisce 28.

# I metodi `numeroGiorni`

- `public int numeroGiorni()`
  Restituisce il numero dei giorni del mese (28 se l'oggetto che esegue il metodo è `FEBBRAIO`).

- `public int numeroGiorni(boolean bisestile)`
  Restituisce il numero dei giorni del mese; nel caso di febbraio, se l'argomento è `true` restituisce 29, se è `false` restituisce 28.

- `public int numeroGiorni(int anno)`
  Restituisce il numero dei giorni del mese, relativamente all'anno specificato come argomento. Pertanto, nel caso degli anni bisestili, per il `FEBBRAIO` restituisce 29.

# Constant-specific methods

```
public enum MeseDellAnno {
  GENNAIO {
    public int numeroGiorni() {
      return 31;
    }
  },
  FEBBRAIO {
    public int numeroGiorni() {
      return 28;
    }
  },
  MARZO {
    public int numeroGiorni() {
      return 31;
    }
  },
  ...
  public abstract int numeroGiorni();
```

# Constant-specific methods

Tali metodi devono essere dichiarati (eventualmente astratti) nel tipo enumerativo

# Constant-specific methods

Tali metodi devono essere dichiarati (eventualmente astratti) nel tipo enumerativo

## Esempio

`numeroGiorni` è stato dichiarato astratto nel corpo del tipo enumerativo `MeseDellAnno`.

Senza tale dichiarazione

```
MeseDellAnno m;
...
m.numeroGiorni()
```

non verrebbe compilato.

# Tipi enumerativi: campi e costruttori

- È possibile associare ad ogni oggetto del tipo enumerativo dei campi

# Tipi enumerativi: campi e costruttori

- È possibile associare ad ogni oggetto del tipo enumerativo dei campi
- È possibile definire dei costruttori:
  - non sono utilizzabili dal programmatore per costruire istanze

# Tipi enumerativi: campi e costruttori

- È possibile associare ad ogni oggetto del tipo enumerativo dei campi
- È possibile definire dei costruttori:
  - non sono utilizzabili dal programmatore per costruire istanze
  - vengono utilizzate dalla JVM per costruire le istanze corrispondenti alle costanti

# MeseDellAnno: un implementazione più elegante

```
public enum MeseDellAnno {
  // COSTANTI ENUMERATIVE
  ...
  ...

  // CAMPI
  private String nome;
  private int numGiorni;

  // COSTRUTTORI
  private MeseDellAnno(String nome, int numGiorni) {
    this.nome = nome;
    this.numGiorni = numGiorni;
  }
  ...
}
```

# MeseDellAnno: un'implementazione più elegante

```
public enum MeseDellAnno {
  GENNAIO("Gennaio", 31), FEBBRAIO("Febbraio", 28),
  MARZO("Marzo", 31), APRILE("Aprile", 30),
  MAGGIO("Maggio", 31), GIUGNO("Giugno", 30),
  LUGLIO("Luglio", 31), AGOSTO("Agosto", 31),
  SETTEMBRE("Settembre", 30), OTTOBRE("Ottobre", 31),
  NOVEMBRE("Novembre", 30), DICEMBRE("Dicembre",31);

  // CAMPI
  private String nome;
  private int numGiorni;

  // COSTRUTTORI
  private MeseDellAnno(String nome, int numGiorni) {
    this.nome = nome;
    this.numGiorni = numGiorni;
  }
  ...
}
```

# I metodi numeroGiorni e toString

```
public enum MeseDellAnno {
  GENNAIO("Gennaio", 31), FEBBRAIO("Febbraio", 28),
  MARZO("Marzo", 31), APRILE("Aprile", 30),
  ...

  // CAMPI
  private String nome;
  private int numGiorni;
  ...

  public int numeroGiorni() {
    return this.numGiorni;
  }

  public String toString() {
    return this.nome;
  }
}
```

# I metodi numeroGiorni

```
public enum MeseDellAnno {
  GENNAIO("Gennaio", 31),
  FEBBRAIO("Febbraio", 28){
    public int numeroGiorni(boolean bisestile) {
      return bisestile ? 29 : 28;
    }
  }, MARZO("Marzo", 31),
  ...
  public int numeroGiorni() {
    return this.numGiorni;
  }

  public int numeroGiorni(boolean bisestile) {
    return this.numeroGiorni();
  }
  ...
}
```

# I metodi numeroGiorni

```
public enum MeseDellAnno {
  GENNAIO("Gennaio", 31),
  FEBBRAIO("Febbraio", 28){
    public int numeroGiorni(boolean bisestile) {
      return bisestile ? 29 : 28;
    }
  }, MARZO("Marzo", 31),
  ...
  public int numeroGiorni(boolean bisestile) {
    return this.numeroGiorni();
  }

  public int numeroGiorni(int anno) {
    Data d = new Data(1, 1, anno);
    return this.numeroGiorni(d.isInAnnoBisestile());
  }
```

# La classe generica Coppia<E, F>

**Contratto**

Le istanze di `Coppia<E, F>` sono coppie di oggetti, il primo di tipo `E` ed il secondo di tipo `F`.

# La classe generica Coppia<E, F>

## Contratto

Le istanze di `Coppia<E, F>` sono coppie di oggetti, il primo di tipo `E` ed il secondo di tipo `F`.

## Esempio

```
Data d = new Data(22, 1, 2005);
Orario o = new Orario(23, 33);
Coppia<Data, Orario> tempo = new Coppia<Data, Orario>(d, o);

Coppia<Data, Orario> adesso =
              new Coppia<Data, Orario>(new Data(), new Orario());
```

# I metodi

- `public E getSinistro()`
  Restituisce il primo elemento della coppia, cioè quello di sinistra.

# I metodi

- `public E getSinistro()`
  Restituisce il primo elemento della coppia, cioè quello di sinistra.
- `public F getDestro()`
  Restituisce il secondo elemento della coppia, cioè quello di destra.

# I metodi

- `public E getSinistro()`
  Restituisce il primo elemento della coppia, cioè quello di sinistra.
- `public F getDestro()`
  Restituisce il secondo elemento della coppia, cioè quello di destra.
- `public String toString()`
  Restituisce una stringa contenente le stringhe associate ai due oggetti che costituiscono la coppia.

# I metodi

- `public E getSinistro()`
  Restituisce il primo elemento della coppia, cioè quello di sinistra.
- `public F getDestro()`
  Restituisce il secondo elemento della coppia, cioè quello di destra.
- `public String toString()`
  Restituisce una stringa contenente le stringhe associate ai due oggetti che costituiscono la coppia.
- `public static int numeroCoppie()`
  Restituisce il numero totale di istanze della classe costruite.

# Implementazione

```
public class Coppia<E, F> {
  //CAMPI
  private E sinistro;
  private F destro;
  private static int nCoppie = 0;

  ...
}
```

# Implementazione

```
public class Coppia<E, F> {
  //CAMPI
  private E sinistro;
  private F destro;
  private static int nCoppie = 0;

  //COSTRUTTORI
  public Coppia(E e, F f) {
    sinistro = e;
    destro = f;
    nCoppie++;
  }
  ...
}
```

# Implementazione

```
public class Coppia<E, F> {
  //CAMPI
  private E sinistro;
  private F destro;
  private static int nCoppie = 0;
  ...
  //METODI
  public E getSinistro() {
    return sinistro;
  }

  public F getDestro() {
    return destro;
  }

  public String toString() {
    return "(" + sinistro + ", " + destro + ")";
  }
```

# Implementazione

```
public class Coppia<E, F> {
  //CAMPI
  private E sinistro;
  private F destro;
  private static int nCoppie = 0;

  ...

  //METODI STATICI
  public static int numeroCoppie() {
    return nCoppie;
  }
}
```

# Definizione di un'interfaccia

- Le interfacce sono definite mediante la parola chiave `interface`

# Definizione di un'interfaccia

- Le interfacce sono definite mediante la parola chiave `interface`
- Nel corpo è possibile specificare solo:
  - prototipi di metodi
  - costanti statiche (cioè dei campi definiti `static` e `final`)

# Definizione di un'interfaccia

- Le interfacce sono definite mediante la parola chiave `interface`
- Nel corpo è possibile specificare solo:
  - prototipi di metodi
  - costanti statiche (cioè dei campi definiti `static` e `final`)
- Tutti i metodi di un'interfaccia sono automaticamente `public` anche se non viene indicato

# Definizione di un'interfaccia

- Le interfacce sono definite mediante la parola chiave `interface`
- Nel corpo è possibile specificare solo:
  - prototipi di metodi
  - costanti statiche (cioè dei campi definiti `static` e `final`)
- Tutti i metodi di un'interfaccia sono automaticamente `public` anche se non viene indicato

**Esempio**

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

# Estensione di un'interfaccia

- Si può definire un'interfaccia estendendo interfacce già definite

# Estensione di un'interfaccia

- Si può definire un'interfaccia estendendo interfacce già definite
- È possibile estendere più interfacce

# Estensione di un'interfaccia

- Si può definire un'interfaccia estendendo interfacce già definite
- È possibile estendere più interfacce

## Esempio

```
public interface I extends I1, I2 {
   ...
}
```

# Estensione di un'interfaccia

- Si può definire un'interfaccia estendendo interfacce già definite
- È possibile estendere più interfacce

## Esempio

```
public interface I extends I1, I2 {
   ...
}
```

Un oggetto di una classe che implementa `I` può essere visto:

- come un oggetto di tipo `I`
- come un oggetto di tipo `I1`
- come un oggetto di tipo `I2`

# Ordinare un array di interi: bubblesort

Iterativamente:

- confrontiamo coppie di elementi adiacenti

# Ordinare un array di interi: bubblesort

Iterativamente:

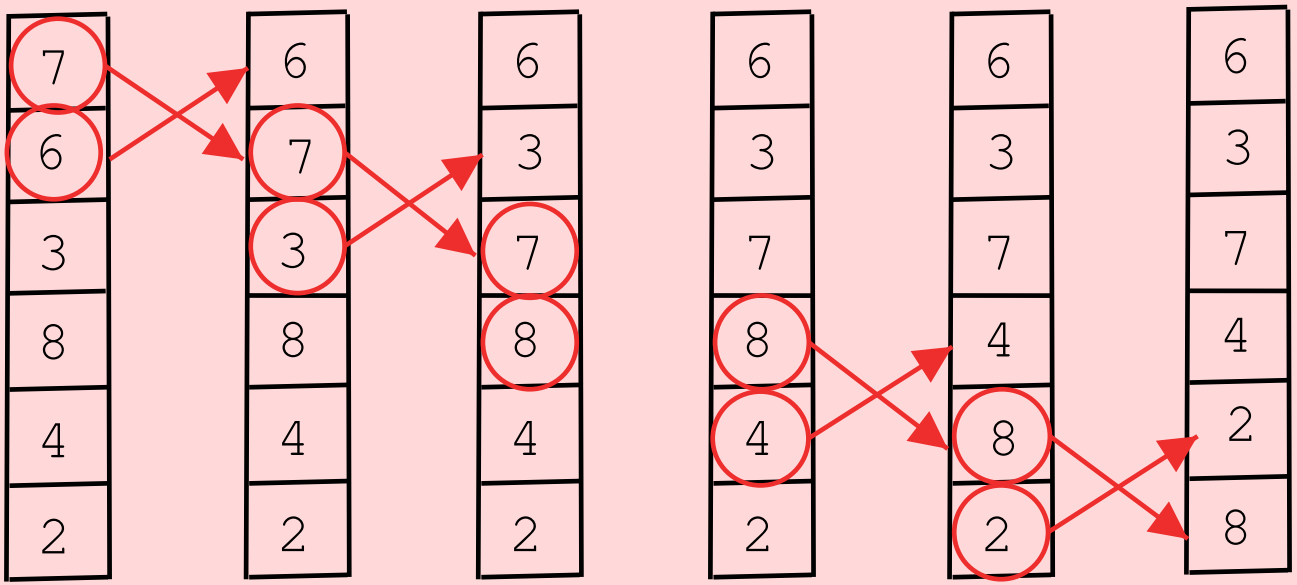
- confrontiamo coppie di elementi adiacenti
- se non sono nell'ordine giusto scambiamo gli elementi della coppia.

# Ordinare un array di interi: bubblesort

Iterativamente:

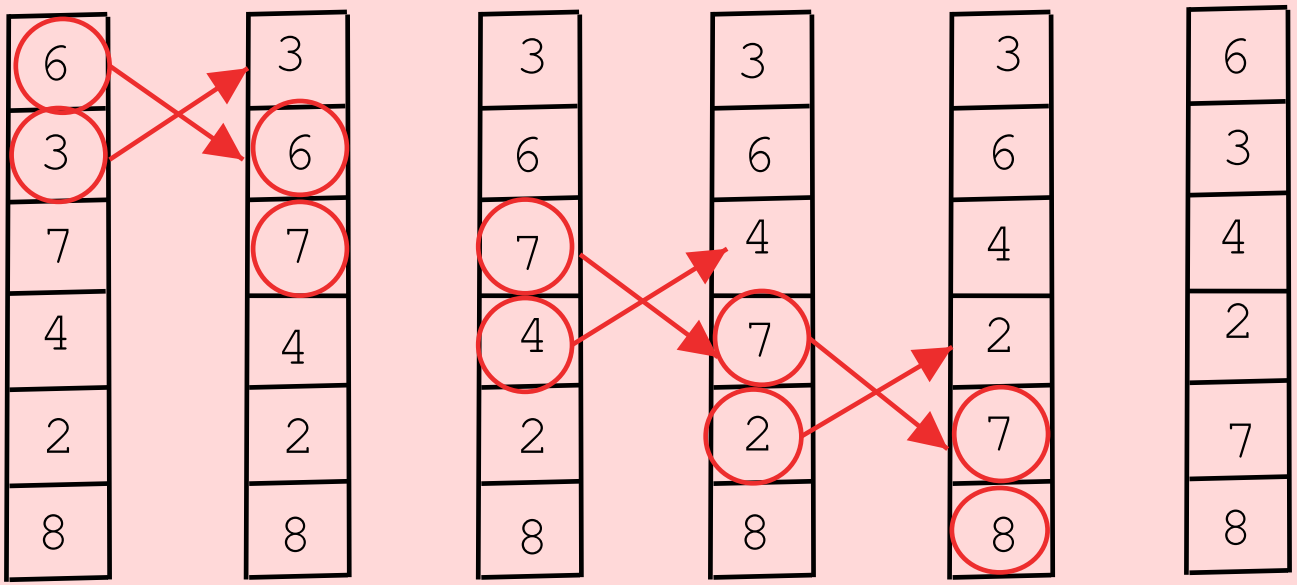
- confrontiamo coppie di elementi adiacenti
- se non sono nell'ordine giusto scambiamo gli elementi della coppia.

## Prima iterazione



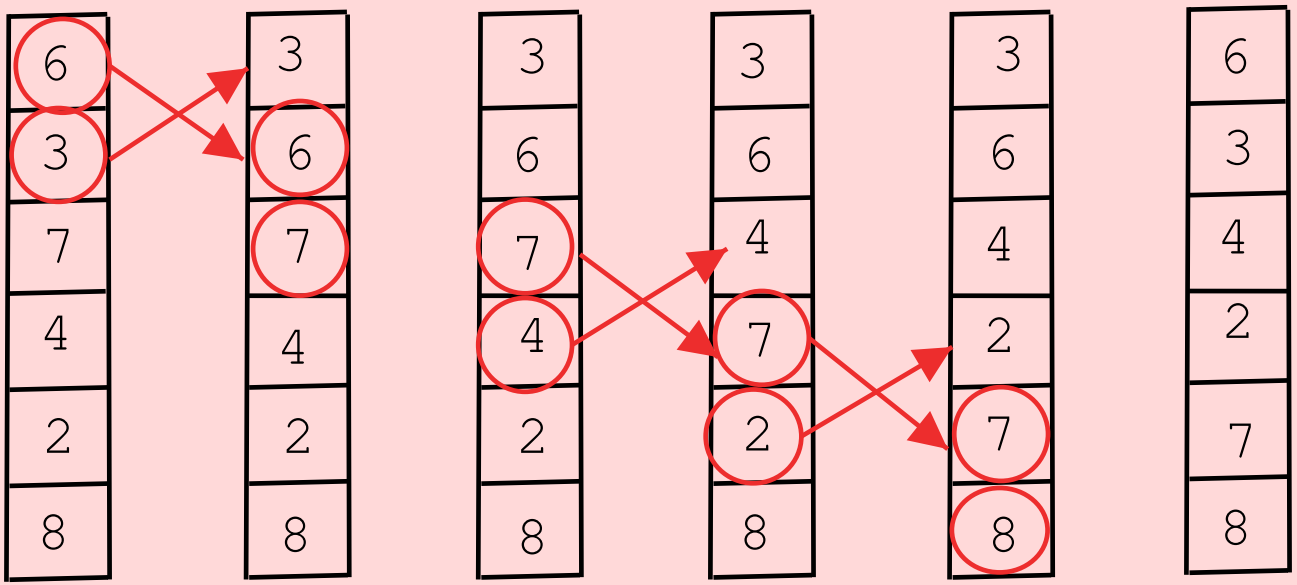
# Bubblesort

## Seconda iterazione



# Bubblesort

## Seconda iterazione




Se l'array è ordinato non vengono effettuati scambi: questa è la condizione per concludere le iterazioni.

# Bubblesort: lo schema

```
do {
 scambiato = false;
 for ( ... )  //   effettua una scansione
  if (gli elementi di posto i - 1 e i NON sono in ordine) {
   scambiali;
   scambiato = true;
 }
} while (scambiato);
```

# Bubblesort: implementazione

```
public static void ordina(int[] a) {
  int temp;
  boolean scambiato;
  do {
    scambiato = false;
    for (int i = 1; i < a.length; i++)
      if (a[i - 1] > a[i]) {
        temp = a[i - 1];
        a[i - 1] = a[i];
        a[i] = temp;
        scambiato = true;
      }
  } while (scambiato);
}
```

# Ordinare un array di `String`: bubblesort

```
public static void ordina(String[] a) {
  String temp;
  boolean scambiato;
  do {
    scambiato = false;
    for (int i = 1; i < a.length; i++)
      if (a[i - 1].compareTo(a[i]) > 0) {
        temp = a[i - 1];
        a[i - 1] = a[i];
        a[i] = temp;
        scambiato = true;
      }
  } while (scambiato);
}
```

# Ordinare un array di Comparable: bubblesort

```
public static <T extends Comparable<? super T>>
                                  void ordina(T[] a) {
  T temp;
  boolean scambiato;
  do {
    scambiato = false;
    for (int i = 1; i < a.length; i++)
      if (a[i - 1].compareTo(a[i]) > 0) {
        temp = a[i - 1];
        a[i - 1] = a[i];
        a[i] = temp;
        scambiato = true;
      }
  } while (scambiato);
}
```

# Ricerca in un array di interi

```
public static int cerca(int[] a, int chiave) {
  boolean trovato = false;
  int i;
  for (i = 0; i < a.length && !trovato; i++)
    if (chiave == a[i])
      trovato = true;

  if (trovato)
    return i - 1;
  else return -1;
  }
```

# Osservazioni

- Il tempo impiegato dal ciclo è proporzionale al numero di confronti (`chiave == a[i]`) effettuati

# Osservazioni

- Il tempo impiegato dal ciclo è proporzionale al numero di confronti (`chiave == a[i]`) effettuati
- Caso peggiore
  Se l'elemento cercato non appartiene all'array: il tempo impiegato è proporzionale alla lunghezza dell'array

# Osservazioni

- Il tempo impiegato dal ciclo è proporzionale al numero di confronti (`chiave == a[i]`) effettuati
- Caso peggiore
  Se l'elemento cercato non appartiene all'array: il tempo impiegato è proporzionale alla lunghezza dell'array
- Se l'array è ordinato possiamo fare meglio

  Un array, privo di ripetizioni

  ```
  int[] a = new int[N];
  ```

  è ordinato se, per ogni $i = 0, \ldots, N-2$, `a[i]<a[i+1]`

# Ricerca dicotomica

Consideriamo l'array ordinato a =

| 3 | 7 | 8 | 9 | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

# Ricerca dicotomica

Consideriamo l'array ordinato a =

| 3 | 7 | 8 | 9 | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

Supponiamo di cercare 8

# Ricerca dicotomica

Consideriamo l'array ordinato a =

| 3 | 7 | 8 | 9 | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

Supponiamo di cercare 8

- Possiamo pensare a come composto da 3 parti

| 3 | 7 | 8 | **9** | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

- Confrontiamo l'elemento da cercare (8) con l'elemento di mezzo (9)

# Ricerca dicotomica

Consideriamo l'array ordinato a =

| 3 | 7 | 8 | 9 | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

Supponiamo di cercare 8

- Possiamo pensare a come composto da 3 parti

| 3 | 7 | 8 | **9** | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

- Confrontiamo l'elemento da cercare (8) con l'elemento di mezzo (9)

Con due confronti (8==9 ? e 8<9 ?) siamo in grado di stabilire che l'elemento da cercare, se è nell'array, deve stare nella parte a sinistra

# Ricerca dicotomica

Consideriamo l'array ordinato a =

| 3 | 7 | 8 | 9 | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

Supponiamo di cercare 8

- Possiamo pensare a come composto da 3 parti

| 3 | 7 | 8 | **9** | 15 | 21 | 27 |
|---|---|---|---|---|---|---|

- Confrontiamo l'elemento da cercare (8) con l'elemento di mezzo (9)

Con due confronti (8==9 ? e 8<9 ?) siamo in grado di stabilire che l'elemento da cercare, se è nell'array, deve stare nella parte a sinistra

Possiamo tralasciare di considerare gli elementi nella parte destra

# Ricerca dicotomica

- Iteriamo il procedimento:

# Ricerca dicotomica

- Iteriamo il procedimento:


- Confrontiamo (8) con l'elemento di mezzo (7)

# Ricerca dicotomica

- Iteriamo il procedimento:


- Confrontiamo (8) con l'elemento di mezzo (7)

l'elemento da cercare, se appartiene all'array, deve stare nella parte destra

# Ricerca dicotomica

- Iteriamo il procedimento:


- Confrontiamo (8) con l'elemento di mezzo (7)

  l'elemento da cercare, se appartiene all'array, deve stare nella parte destra

- Iteriamo di nuovo:

# Ricerca dicotomica

- Iteriamo il procedimento:


- Confrontiamo (8) con l'elemento di mezzo (7)

  l'elemento da cercare, se appartiene all'array, deve stare nella parte destra

- Iteriamo di nuovo:

  abbiamo un unico elemento, con un confronto stabiliamo che l'elemento sta nell'array

# Quanti confronti?

- Ad ogni confronto dimezziamo la dimensione dell'array da analizzare

# Quanti confronti?

- Ad ogni confronto dimezziamo la dimensione dell'array da analizzare
- Il caso peggiore si ha quando l'elemento da cercare non sta nell'array

# Quanti confronti?

- Ad ogni confronto dimezziamo la dimensione dell'array da analizzare
- Il caso peggiore si ha quando l'elemento da cercare non sta nell'array

## Esempio

Array con 100 000 elementi

- $1^\circ$ confronto $\Rightarrow$ 50.000 elementi
- $2^\circ$ confronto $\Rightarrow$ 25.000 elementi
- ...

# Quanti confronti?

- Ad ogni confronto dimezziamo la dimensione dell'array da analizzare
- Il caso peggiore si ha quando l'elemento da cercare non sta nell'array

## Esempio

Array con 100 000 elementi

- $1^\circ$ confronto $\Rightarrow$ 50.000 elementi
- $2^\circ$ confronto $\Rightarrow$ 25.000 elementi
- ...

In questo caso il massimo numero di confronti che possiamo fare è dato dal

$$\text{minimo } k \text{ tale che } \frac{100000}{2^k} < 1$$

# Quanti confronti?

Per un array di $N$ elementi il massimo numero di confronti necessari per decidere se un numero $x$ appartiene all'array e per determinarne la posizione è dato dal minimo $k$ tale che:

$$\frac{N}{2^k} \leq 1$$

# Quanti confronti?

Per un array di $N$ elementi il massimo numero di confronti necessari per decidere se un numero $x$ appartiene all'array e per determinarne la posizione è dato dal minimo $k$ tale che:

$$\frac{N}{2^k} \leq 1$$

Da cui:

$$N \leq 2^k$$

# Quanti confronti?

Per un array di $N$ elementi il massimo numero di confronti necessari per decidere se un numero $x$ appartiene all'array e per determinarne la posizione è dato dal minimo $k$ tale che:

$$\frac{N}{2^k} \leq 1$$

Da cui:

$$N \leq 2^k$$

Quindi il numero di confronti necessari é dato dal piú piccolo numero intero $k$ tale che

$$k \geq \log_2 N$$

# Ricerca binaria

```
public static int cercaBinaria(int[] a, int chiave) {
  boolean trovato = false;

  // a array ordinato
  int i = 0, j = a.length - 1, m = -1;
  while(i <= j  && !trovato) {
    m = (i + j) / 2;
    if (chiave < a[m])        // prosegui nella prima meta'
      j = m - 1;
    else if (chiave > a[m])  // prosegui nella seconda meta'
      i = m + 1;
    else
      trovato = true;   // TROVATO
   }

   return trovato ? m : -1;
}
```

# Ricerca dicotomica su un array di Comparable

```
public static <T extends Comparable<? super T>>
                        int cerca(T[] a, T chiave) {
  int sx = 0, dx = a.length - 1, m;

  while (sx < dx) {
    m = (sx + dx) / 2;
    if (a[m].compareTo(chiave) < 0)
      sx = m + 1;
    else if (a[m].compareTo(chiave) > 0)
      dx = m - 1;
    else
      sx = dx = m;
  }
  if (a[sx].compareTo(chiave) == 0)
    return sx;
  else return -1;
}
```