Interfacce

Interfacce

- **Dichiarazioni di tipi riferimento che descrivono oggetti in modo astratto**
- **Specificano solo le firme dei metodi tralasciando tutti gli aspetti di implementazione**

Interfacce

```
interface Polynomial
{
   public void somma (Polynomial p);
   public void derivata();
   public int grado();
   public double coeff(int grado);
}
```

Interfacce

```
interface Collection<E>
{
   public void add(E element);
   public void remove(E element);
   public boolean find(E element);
   . . .
}
```

Classi implementano Interfacce

- La keyword `implements` indica che una classe implementa una interfaccia

```
class ArrayPoly implements Polynomial
{
   ...
}
```

```
class TreeSet<E> implements Collection<E>
{
   ...
}
```

Classi implementano Interfacce

- **Una classe che implementa una interfaccia deve implementare tutti i metodi dell'interfaccia**
- **Tutti I metodi dell'interfaccia devono essere dichiarati public nella classe**
- **Nulla vieta che la classe dichiari altri metodi oltre quelli dell'interfaccia che implementa**

Classi implementano Interfacce

- **Una interfaccia può essere implementata da più classi**

```
class ArrayPoly implements Polynomial
{
   ...
}
```

```
class ListPoly implements Polynomial
{
   ...
}
```

Classi implementano Interfacce

- **Una interfaccia può essere implementata da più classi**

```
class HashMap<E> implements Collection<E>
{
   ...
}
```

```
class TreeSet<E> implements Collection<E>
{
   ...
}
```

Conversioni di tipo

- **Una interfaccia “rappresenta” tutte le classi che la implementano**
- **Il tipo interfaccia si può associare a tutti gli oggetti delle classi che la implementano**

Continua…

Conversioni di tipo

- **Possiamo assegnare un riferimento di tipo classe ad una variabile di tipo interfaccia purchè la classe implementi l'interfaccia**

```
Collection<Double> c = new TreeSet<Double>();

Collection<Double> d = new HashMap<Double>();
```

```
Polynomial p = new ArrayPoly();
Polynomial q = new ListPoly();
```

Continua…

Conversioni di tipo

- La conversione è lecita solo in determinate situazioni

```
Polynomial x = new TreeSet<Double>(); // ERRORE
```

- Problema: `TreeSet<Double>()` non implementa `Polynomial`

Conversioni di Tipo

- **Sottotipo (<:)**
  - La relazione che permette di decidere quando è lecito convertire un tipo riferimento in un altro
  - Per il momento diciamo

> S <: T sse
> - S è una classe, T è una interfaccia
> - S implementa T.

- **Principio di sostituibilità**
  - Ovunque ci si aspetti un riferimento di un tipo è lecito utilizzare un rifetimento di un sottotipo

Continua…

Conversioni di Tipo

- **Assumiamo S <: T**
- **Regola di assegnamento**
  - Un riferimento di tipo S si puo sempre assegnare ad una variabile di tipo T
  - Un riferimento di tipo classe può sempre essere assegnato ad una variabile di tipo interfaccia che la classe implementa
- **Regola di passaggio di parametri**
  - Un riferimento di tipo S si puo sempre passare per un parametro di tipo T
  - Un riferimento di tipo classe può sempre essere passato per un parametro di tipo interfaccia che la classe implementa

Conversioni di Tipo

- **Le conversioni abilitate dalla regola**

  `C` <: `I` sse `C` implementa `I`

  sono corrette

- **Dimostrazione intuitiva:**
  - Se **`C`** implementa **`I`** , **`C`** deve definire tutti i metodi dichiarati da **`I`** (e dichiararli public)
  - Quindi tutti le invocazioni di metodo ammesse da **`I`** trovano effettivamente un metodo definito in **`C`**

Continua…

Polimorfismo – *dynamic dispatch*

- **Le regole del linguaggio garantiscono che una variabile di tipo interfaccia ha sempre come valore un riferimento di una classe che implementa l'interfaccia**
- **Se una interfaccia è implementata da più classi, il tipo dei valori legati alla variabile può cambiare**

```
Polynomial p;
p = new ListPoly(...);
p = new ArrayPoly(...);
```

Continua…

Polimorfismo – *dynamic dispatch*

- **Possiamo invocare ognuno dei metodi dell'interfaccia:**

```
int g = p.grado();
```

- **Quale metodo invoca?**

Polimorfismo – *dynamic dispatch*

```
int g = p.grado();
```

- **Se `p` riferisce un `ListPoly`, invoca il metodo `ListPoly.grado()`**
- **Se `p` riferisce un `ArrayPoly`, invoca il metodo `ArrayPoly.grado();`**
- **Polimorfismo (molte forme):**
  - il metodo invocato dipende dipende dal tipo del riferimento legato alla variabile

Continua…

Esempio

- **Costruiamo una applicazione per disegnare un insieme di forme geometriche contenute in una componente grafico:**
  - definiamo **`GWin,`** una classe che descrive un contenitore di forme geometriche disegnate mediante una invocazione del metodo **`paint()`**
  - per esemplificare, consideriamo due tipi di forme**:** **`Quadrato`** e **`Cerchio`**

Forme grafiche

```
class Quadrato
   {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
   }
```

```
class Cerchio
   {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
   }
```

GWin

- Un contenitore di `Quadrati` e `Cerchi`

```
/**
 Una finestra che contiene Quadrati e Cerchi
*/
class GWin
{
  /**
    Disegna tutte le forme di questo component
   */
   public void paint(){ /* disegna su g  */ }
   /**
    Componente grafica su cui disegnare
   */
   private Graphics2D g;
}
```

Domanda

- Che struttura utilizziamo per memorizzare le forme contenute nella `GWin`?
- Come definiamo il metodo `paint()` in modo che disegni tutte le forme della componente?

Risposte

- definiamo una nuova interfaccia: **`Shape`**

```
interface Shape { void draw(Graphics2D g); }
```

- Ridefiniamo le classi **`Quadrato`** e **`Cerchio`** in modo che implementino **`Shape`**
- Memorizziamo gli oggetti della componente in una **`ArrayList<Shape>`**

Shapes

```
class Quadrato implements Shape
   {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
   }
```

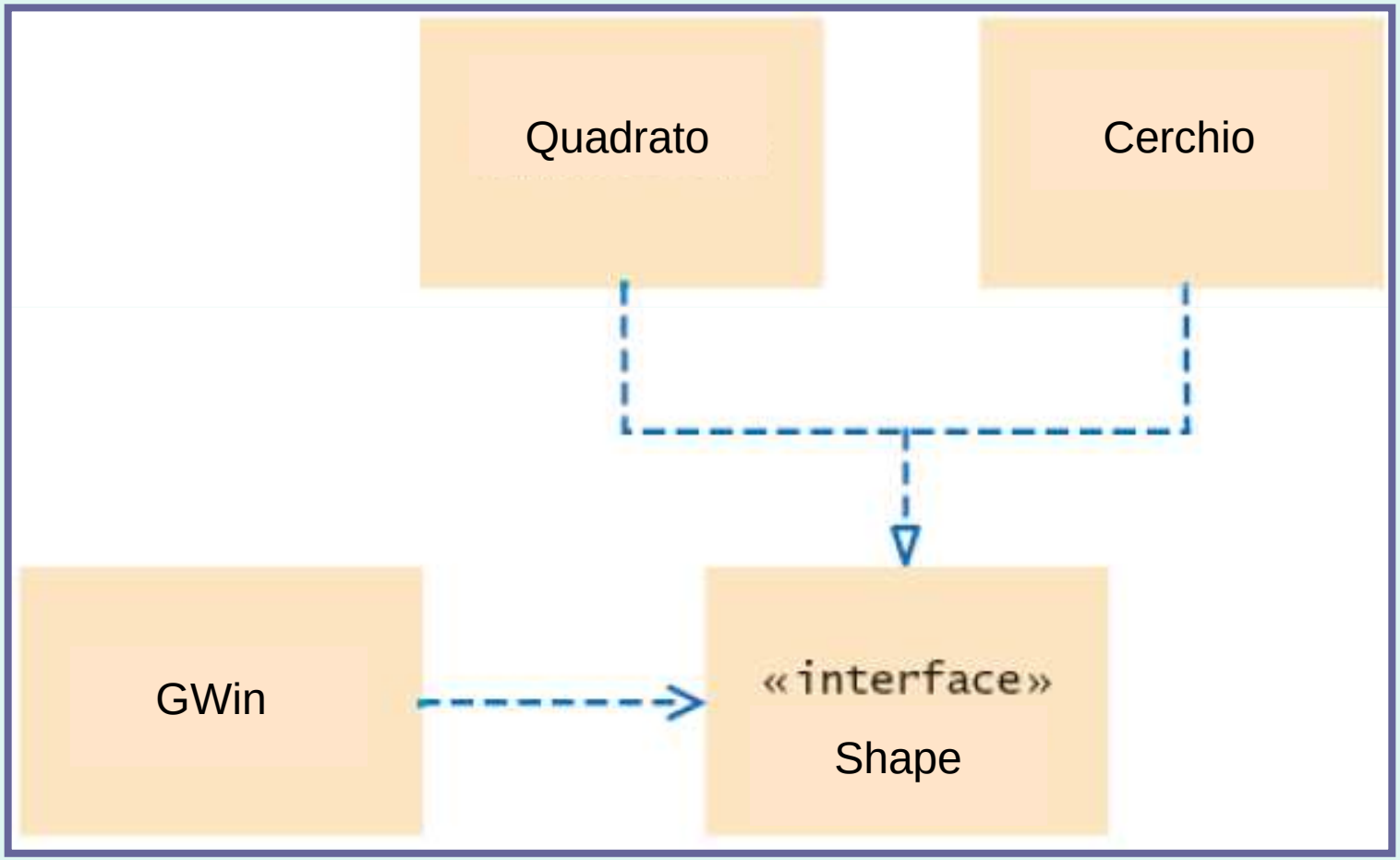
```
class Cerchio implements Shape
 {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
  }
```

GWin

Mantiene una `ArrayList<Shape>`

```
class GWin
{
    private Graphics2D g;
    private ArrayList<Shape> shapes;
    // crea una GWin con un insieme di forme
    public GWin(Shape... shapes)
    {
        Graphics2D g = new Graphics2D();
        this.shapes = new ArrayList<Shape>();
        for (Shape s:shapes) this.shapes.add(s);
    }
    // disegna tutte le componenti della GWin
    public void paint()
    {
        for (Shape s:shapes) s.draw(g);
    }
}
```

Diagramma delle Classi



Ancora Interfacce

Esempio

- **Definiamo una classe `DataSet` che permette di condurre alcune semplici analisi su un insieme di conti bancari**
  - calcolo della media degli importi del saldo
  - calcolo del conto con il valore massimo tra i saldi

Continua…

DataSet

```
public class DataSet
{
  public void add(BankAccount x)
   {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() < x.getBalance())
         maximum = x;
      count++;
   }

   public BankAccount getMaximum() { return maximum; }
   public double average()
   {return (count>0)? sum/count : Double.NaN; }

   private double sum;
   private BankAccount maximum;
   private int count;
}
```

Esempio

- **Ora ripetiamo l'esempio per definire una versione della classe `DataSet` che permette di condurre alcune semplici analisi su un insieme di corsi universitari:**
  - calcolo della media del numero di studenti per corso
  - calcolo del corso con massimo numero di studenti

```
class Corso
{
  // . . .
  public double iscritti()
  {
    //  ...
  }
}
```

DataSet – versione per Corso

```
public class DataSet
{
   public void add(Corso x)
   {
      sum = sum + x.iscritti();
      if (count == 0 || maximum.iscritti() < x.iscritti())
         maximum = x;
      count++;
   }

   public Corso getMaximum() { return maximum; }
   public double average()
   {return (count>0)? sum/count : Double.NaN; }

   private double sum;
   private Corso maximum;
   private int count;
}
```

Interfacce ⇨ riuso di codice

- **Il meccanismo di analisi dei dati è sempre lo stesso;**
  - si basa sull'estrazione di una misura
  - la differenza è solo nell'implementazione del metodo che fornisce la misura
- **Possiamo uniformare:**
  - è sufficiente stabilire una modalità uniforme per estrarre la misura.

```
interface Measurable
{
   double getMeasure();
}
```

Measurable BankAccounts

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return getBalance();
   }
   // ...
}
```

Continua...

Measurable Corsi

```
class Corso implements Measurable
{
   public double getMeasure()
   {
      return iscritti();
   }
   // . . .
}
```

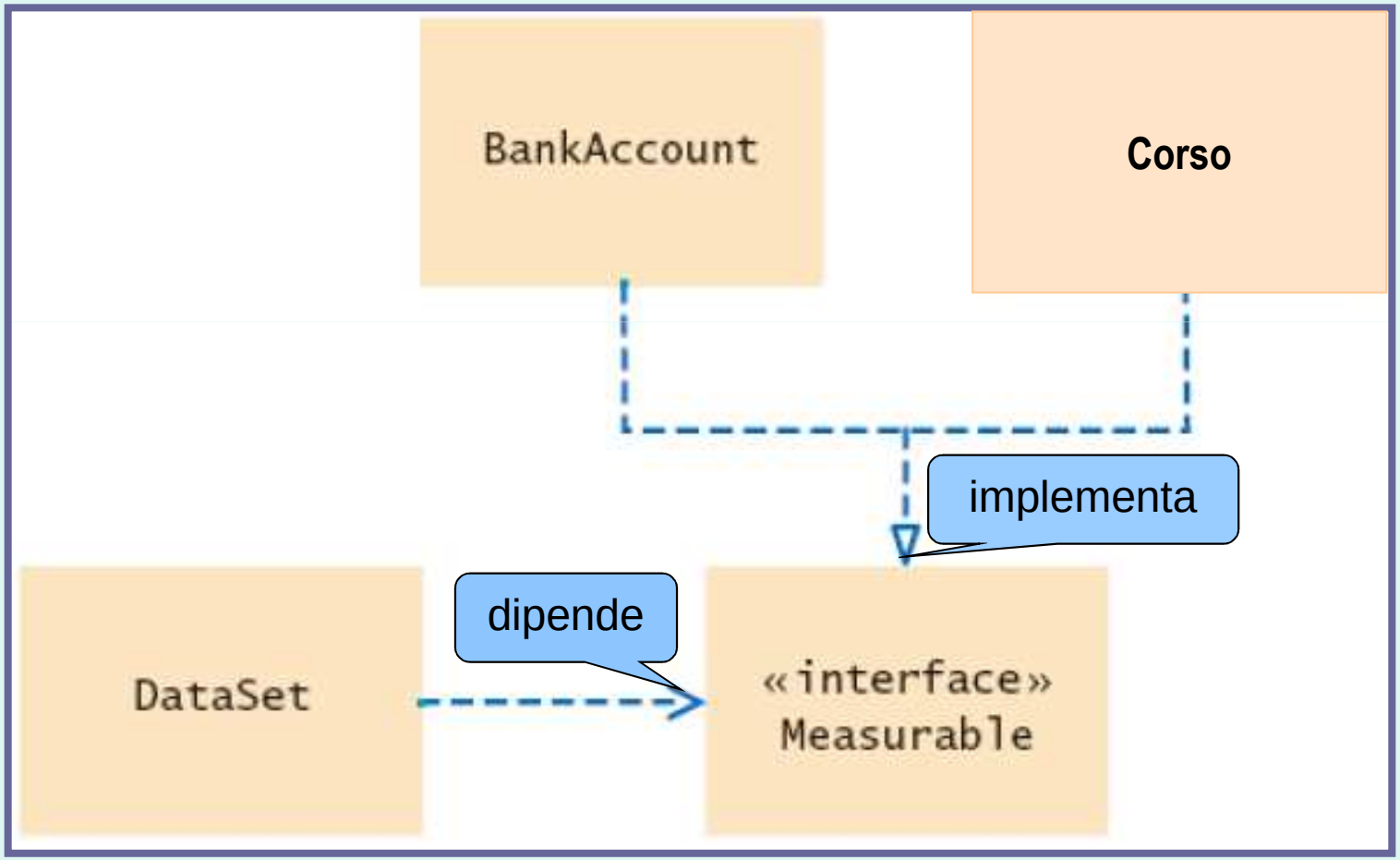
DataSet – versione generica

```
public class DataSet
{
  public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 || maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public Measurable getMaximum() { return maximum; }
   public double average() { return sum/count; }

   private double sum;
   private Measurable maximum;
   private int count;
}
```

DataSet Diagramma UML



Interfacce standard

- **Le librerie Java forniscono molte interfacce standard, che possono essere utilizzare nella programmazione**
- **Un esempio:**
  - **`ActionListener`**

ActionListener

- **Descrive oggetti che svolgono il ruolo di gestori di eventi:**
  - rimane attesa (*in ascolto*) di un evento, per poi rispondere con una azione al momento in cui l'evento si manifesta

```
interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

Esempio: eventi di un timer

- **Eventi notificati ad un `ActionListener` associato al timer**
- **Il gestore viene attivato ad ogni tick del timer,**

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

  - **`actionPerformed()`** invocato ad ogni tick
  - **`event`**: contiene informazione sull'evento

Continua…

Esempio: eventi di un timer

- La gestione dell'evento avviene nel metodo **`actionPerformed()`**
- Gestioni diverse realizzate da classi diverse che implementano **`ActionListener`**

```
class TimerListener implements ActionListener
{
     public void actionPerformed(ActionEvent event)
     {
      // Eseguito ad ogni tick.
     }
}
```

Continua…

Esempio: eventi di un timer

- **Per associare un particolare listener al timer è necessario registrare il listener sul timer**

```
TimerListener listener = new TimerListener();
Timer t = new Timer(interval, listener);
```

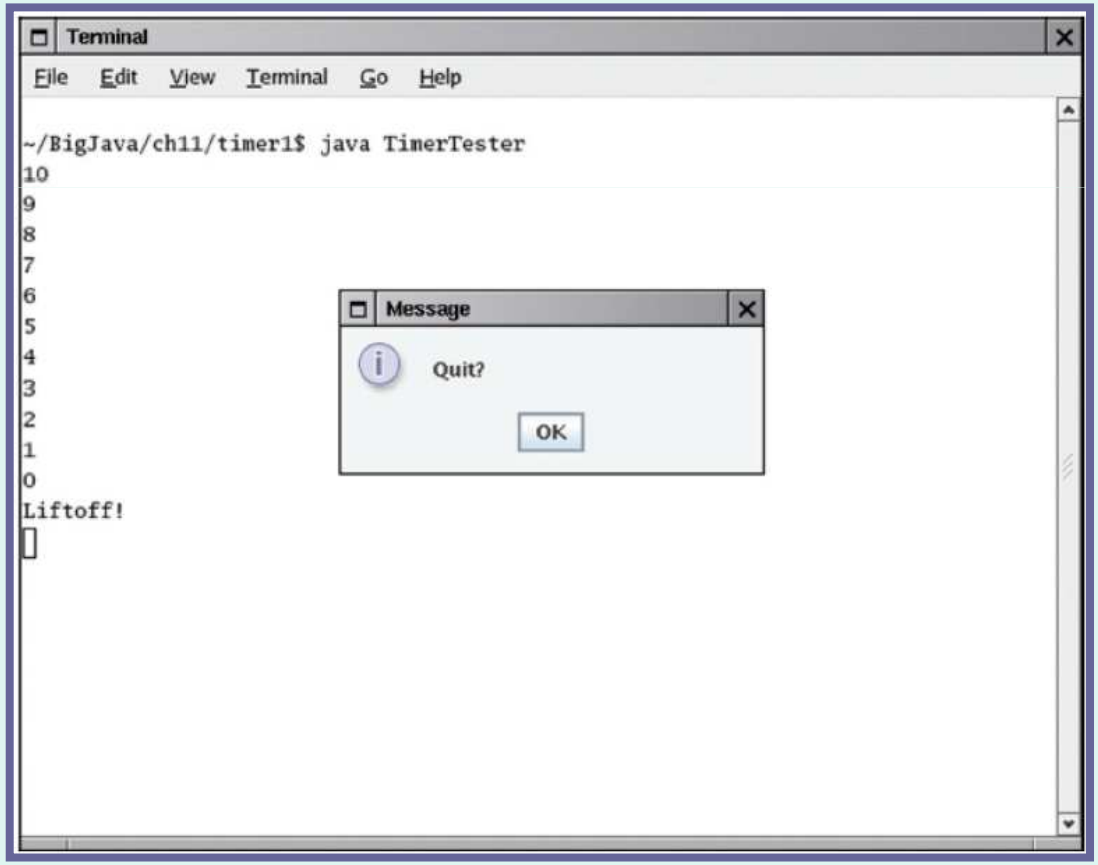
tra due tick

- **Ora possiamo far partire il timer**

```
t.start(); // Esegue in un thread separato
```

Esempio: *countdown*

- **Un timer che esegue il *countdown***



CountDownListener

```
public class CountDownApp
{
   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener(10);
     // Millisecondi tra due tick
     final int DELAY = 1000;
     Timer t = new Timer(DELAY, listener);
     t.start();

     JOptionPane.showMessageDialog(null, "Quit?");
     System.exit(0);
   }
}
```

`CountDownListener`

- **Inizializza un contatore al valore passato nel costruttore**
- **Ad ogni invocazione del metodo `actionPerformed()`**
  - controlla il valore del contatore e dà in output il messaggio corrispondente
  - decrementa il contatore

CountDownListener

```
class CountDownListener implements ActionListener
{
   public CountDownListener(int initialCount)
   {
      count = initialCount;
   }

   public void actionPerformed(ActionEvent event)
   {
      if (count >= 0) System.out.println(count);
      if (count == 0) System.out.println("Liftoff!");
      count--;
   }
   private int count;
}
```

Interfacce multiple

- **Una classe può implementare più di una interfaccia**
- **In quel caso deve definire, public, tutti i metodi delle interfacce che implementa**
- **E' automaticamente sottotipo di ciascuna delle interfacce che implementa**

Measurable Shapes

```
class Quadrato implements Shape, Measurable
   {
      . . .
      public void draw(Graphics2D g) { ... }
      public double getMeasure() { ... }
   }
```

```
class Cerchio implements Shape, Measurable
 {
      . . .
      public void draw(Graphics2D g){ ... }
      public void getMeasure() { ... }
  }
```

Measurable Shapes

- Ora possiamo passare `Quadrati` e `Cerchi` sia all'interno della classe `Gwin` per essere disegnati, sia all'interno della classe `DataSet` per essere misurati

Conversioni di tipo

- **Una interfaccia “rappresenta” tutte le classi che la implementano**
- **Più in generale: un supertipo rappresenta tutti i suoi sottotipi**

Continua…

Conversioni di tipo

- **Principio di sostituibilità**
  - Un riferimento di un sottotipo può essere usato ovunque ci si aspetti un riferimento di un supertipo
- **Può causare perdita di informazione**
  - nel contesto in cui ci aspettiamo il supertipo, non possiamo usare solo I metodi del supertipo
  - perdiamo la possibilità di utilizzare gli eventuali metodi aggiuntivi del sottotipo

Continua…

Ancora Shapes

```
class Car implements Shape
{
      . . .
   public void draw(Graphics2D g){ . . . }
   public String brand() {. . . }
}
```

```
class Smiley implements Shape
{
      . . .
   public void draw(Graphics2D g){ . . . }
   public String mood() {. . . }
}
```

Sottotipi e perdita di informazione

- **Consideriamo**

```
public static void printBrand(List<Shape> l)
{
     for (Shape s : l)
        // stampa la marca di tutte le macchine di l
        // ???
}
```

Continua…

Sottotipi e perdita di informazione

- **Certamente non possiamo fare così …**

```
public static void printBrand(List<Shape> l)
{
     for (Shape s : l)
        // stampa la marca di tutte le macchine di l
        System.out.println( s.brand() ); // TYPE ERROR!
}
```

Continua…

Cast

- **Permette di modificare il tipo associato ad una espressione**

```
((Car)s).brand()
```

- **Un cast è permesso dal compilatore solo se applica conversioni tra tipi compatibili**
- **Compatibili = sottotipi (per il momento)**
- **Anche quando permesso dal compilatore, un cast può causare errore a run time**
- **Se `s` non è un `Car` errore a run time**

Continua…

Tipo statico e Dinamico

- **Tipo statico e tipo dinamico di una variabile**
  - tipo statico: quello dichiarato
  - tipo dinamico: il tipo del riferimento assegnato alla variabile
- **Il tipo dinamico può cambiare durante l'esecuzione, ma le regole garantiscono che**
  - Il tipo dinamico di una variabile è sempre un sottotipo del tipo statico della variabile

Continua…

Cast

- **`(T)var` causa errore**
  - **in compilazione**
    se **`T`** non è compatibile con il tipo statico di **`var`**
  - **in esecuzione** (**`ClassCastException`**)
    se **`T`** non è compatibile con il tipo dinamico di **`var`**

Continua…

Cast

```
Shape s =  new Car();
```

- **OK: `Car` sottotipo di `Shape`**

```
Car c = (Car) s
```

- **Compila correttamente**
  - il tipo dichiarato di **`s`** è **`Shape`**
  - **`Car`** e **`Shape`** sono compatibili
- **Esegue correttamente**
  - s è un **`Car`** (il tipo dinamico di **`s`** è **`Car`**)

Continua…

Cast

```
Shape s =  new Car();
```

- **OK: `Car` sottotipo di `Shape`**

```
Smiley c = (Smiley) s
```

- **Compila correttamente**
  - il tipo dichiarato di **`s`** è **`Shape`**
  - **`Smiley`** e **`Shape`** sono compatibili
- **Errore a run time**
  - **`s`** non è uno **`Smiley`**

Continua…

Cast

- **Attenzione anche qui …**

```
public static void printBrand(List<Shape> l)
{
     for (Shape s : l)
        // ClassCastException se s instance of Smiley
        System.out.println( ((Car)s).brand() );
}
```

Continua…

`instanceof`

- **Permette di determinare il tipo dinamico di una variabile**

> `x istanceof T` **è true solo se** `x` **ha tipo dinamico** `T`

- **Quindi permette di evitare errori in esecuzione**

```
if (x instanceof T)  return (T) x
```

- **Esegue correttamente, perchè** `x` **è sicuramente un** `T`

Cast

- **Questo, finalmente, è corretto**

```
public static void printBrand(List<Shape> l)
{
     for (Shape s : l)
        if (s instanceof Car)
             System.out.println( ((Car)s).brand() );
}
```

Domanda

- **Come disegnare solo le `Shapes` che sono `Cars`?**

Risposta

```
// disegna tutte le Cars della GWin

public void paint(){
    for (Shape c:shapes)
            if (c instanceof Car) c.draw(g);
}
```