



Programmazione III

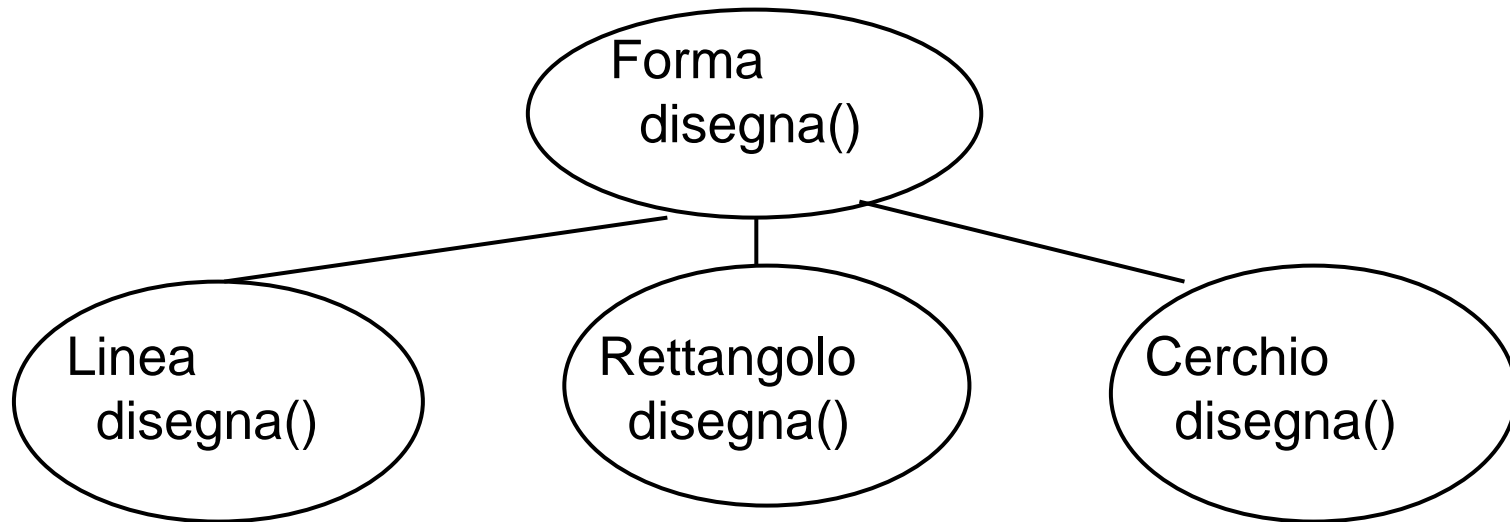
Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

**RunTime Type Identification (RTTI) e
Java Reflection
Controllo di tipo per downcast**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

Programmare con l'ereditarietà - I



```
interface Forma {  
    void disegna();  
}
```

```
class Linea implements Forma {  
    void disegna() { ..... }  
}
```

...

```
Forma f = new Linea();  
f.disegna();
```

Programmare con l'ereditarietà - II



Si vuole eseguire un'operazione **op** particolare su oggetti di tipo Cerchio → downcast da Forma a Cerchio

```
Forma f;
```

```
...
```

```
(Cerchio)f.op()
```

Se f non è un cerchio, viene sollevata una eccezione a run-time. Per prevenirla si può verificare il tipo di un oggetto a run-time:

```
Forma f;
```

```
...
```

```
if (f instanceof Cerchio) (Cerchio)f.op()
```

La notazione instanceof è statica. Deve essere specificato il nome del tipo (Cerchio, Triangolo, ecc.). Non abusare.

La classe **Class** (metaclass)



In Java esiste la classe **Class**. Per ogni classe **C** usata in un programma, c'è un unico oggetto a run-time di tipo **Class** che rappresenta quella classe.

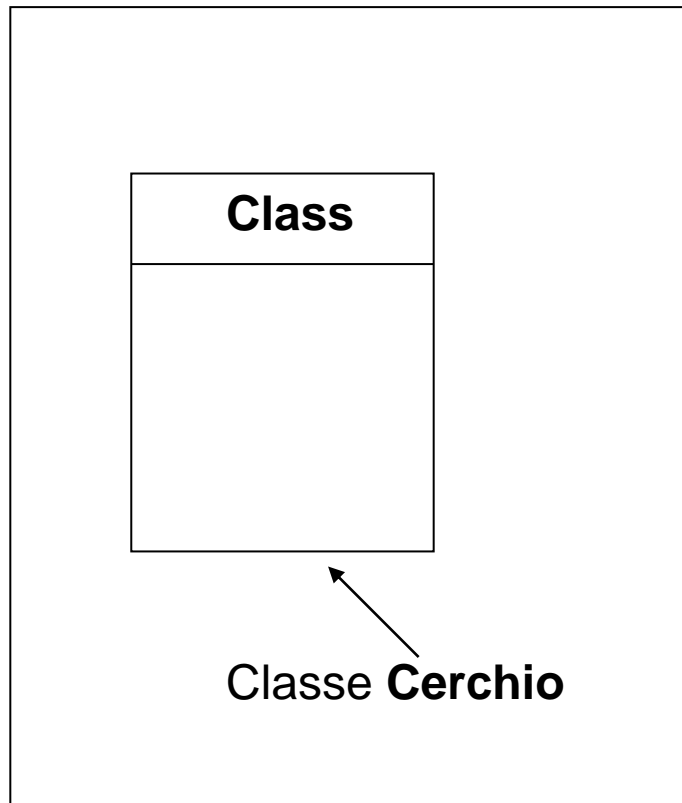
Esiste un oggetto **Class** per ogni tipo di dato: classi, tipi enumerativi, interfacce, annotazioni, array e tipi primitivi. L'oggetto **Class** serve per analizzare la classe (nome, membri, ...).

Quando un programma è in esecuzione, il sistema runtime di Java conserva la **RunTime Type Identification (RTTI)** di ogni oggetto. Per ogni oggetto *o* si mantiene il riferimento all'oggetto **Class** che rappresenta la classe di *o*.

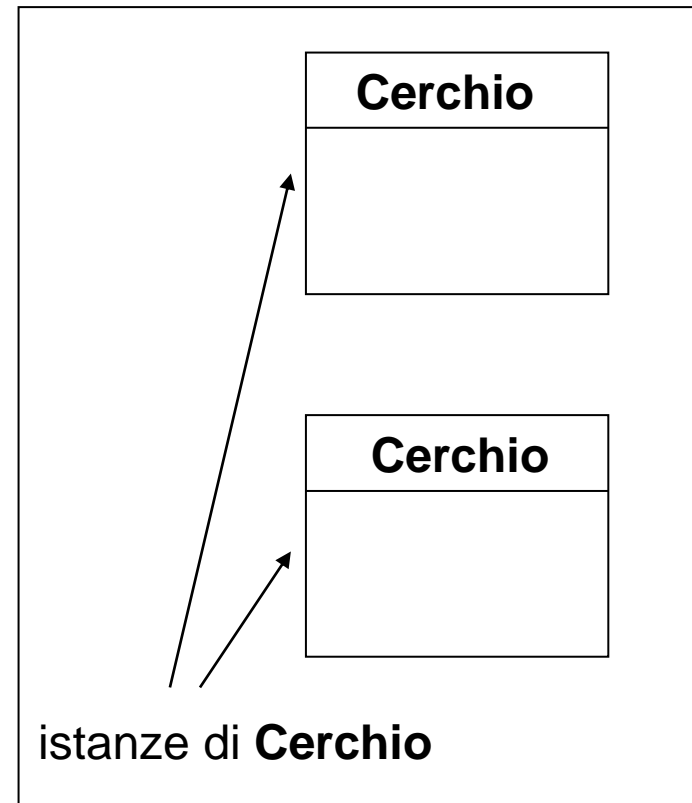
Classi e istanze



Area delle classi (memoria statica)



HEAP



Come rilevare dinamicamente il tipo di un oggetto? - I



Object offre il metodo **getClass()** che restituisce la classe dell'oggetto che lo esegue:

Forma f;

...

Class c = f.getClass();

System.out.println(c.getName());

Class offre i seguenti metodi:

getName() restituisce il nome della classe come stringa

isInstance() - versione dinamica di **instanceof**:

Class c;

...

c.isInstance(f)

Come rilevare dinamicamente il tipo di un oggetto?



Altri metodi di **Class**

```
Class c = Class.forName("Cerchio");
```

```
Class c = Cerchio.class;
```

due modi per ottenere l'oggetto associato alla classe Cerchio;
o anche attraverso gli oggetti della classe:
`Cerchio c = new Cerchio(); c.getClass();`

`c.getSuperclass();` restituisce la sopraclasse

`c.newInstance();` crea un nuovo oggetto della classe **c**
(di tipo **Object**)

Come rilevare dinamicamente il tipo di un oggetto? Esempio

```
public class Esempio0 {  
    public static void main(String[] args) {  
        String f = "Ciao";  
        Class c = f.getClass();  
        System.out.println("Ciao e' istanza di " + c.getName() +  
                           "? " + c.isInstance(f));  
        ArrayList ar = new ArrayList();  
        System.out.println("L'ArrayList e' istanza di " +  
                           c.getName() + "? " + c.isInstance(ar));  
        System.out.println("La classe e' " + ar.getClass().getName());  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
Ciao e' istanza di java.lang.String? true  
L'ArrayList e' istanza di java.lang.String? false  
La classe e' java.util.ArrayList  
Press any key to continue . . .
```


Gestione dell'oggetto Class



Così come un oggetto **Point** descrive le proprietà di un determinato punto del piano cartesiano, un oggetto **Class** descrive le proprietà di una determinata classe.

L'oggetto di tipo **Class** che rappresenta la classe **C** viene creato (**caricato**) dall'interprete, a partire dal file **C.class**, nel momento in cui la classe **C** è usata.

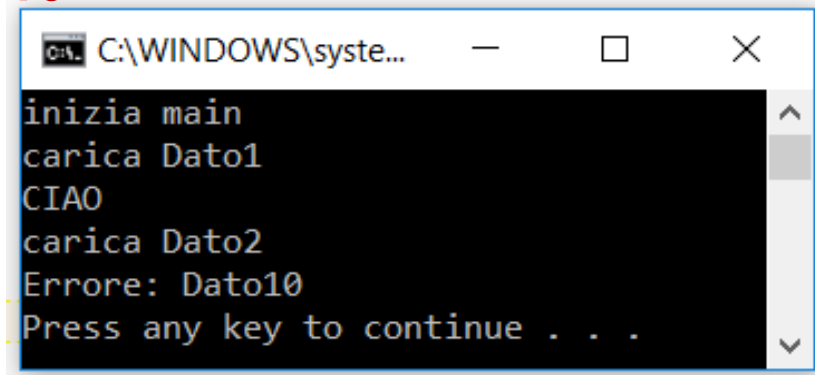
Se il file **C.class** non c'è, l'interprete lancia un'eccezione.

Vedere il prossimo esempio.

Class e errori a runtime – esempio – I

```
class Dato1 {  
    static {  
        System.out.println("carica Dato1");  
    }  
    public static void saluta() {System.out.println("CIAO");}  
}  
  
class Dato2 {  
    static {  
        System.out.println("carica Dato2");  
    }  
}
```

NB: non abbiamo definito la classe Dato10



```
C:\WINDOWS\system32\cmd.exe  
inizia main  
carica Dato1  
CIAO  
carica Dato2  
Errore: Dato10  
Press any key to continue . . .
```

Class e errori a runtime – esempio - II



```
public class Esempio1 {  
  
    public static void main(String[] args) {  
        System.out.println("inizia main");  
        Dato1.saluta();  
        Dato1 d1 = new Dato1();  
        try {  
            Class c = Class.forName("Dato2");  
        } catch (ClassNotFoundException e) {System.out.println("Errore: " +  
                                                                    e.getMessage());}  
  
        try {  
            Class c = Class.forName("Dato10");  
        } catch (ClassNotFoundException e) {System.out.println("Errore: " +  
                                                                    e.getMessage());}  
    }  
}
```

Class e errori a runtime – esempio - III



Nell'esempio precedente, se la classe **Dato1** non è definita, si ottiene un errore di **compilazione** (type checking statico) alla linea

```
Dato1 d1 = new Dato1();
```

Viceversa, se **Dato10** non c'è, si verifica un errore a runtime e viene lanciata **l'eccezione** **ClassNotFoundException** quando si esegue

```
Class c = Class.forName("Dato10");
```

Java Reflection (Riflessione) - I



La Reflection è un meccanismo molto potente fornito da Java per analizzare le funzionalità delle classi, ad esempio per ottenere a run-time informazioni su campi, metodi, costruttori, ...

Il package **java.lang.reflect** contiene le classi **Field**, **Method**, **Constructor**

La classe **Class** contiene metodi come: **getFields**, **getMethods**, **getConstructors**

La classe **Method** contiene i metodi: **getParameterTypes**, **invoke**

Java Reflection (Riflessione) - II



Ad esempio la riflessione permette di leggere il nome di una classe e estrarre **dinamicamente** le informazioni su campi e metodi della classe.

La riflessione è usata in *JavaBeans*, l'architettura a componenti di Java, per analizzare dinamicamente le proprietà di nuovi componenti.

Vedere l'applicazione di esempio: ReflectionTest

Java Reflection (Riflessione) - III

```
public class ReflectionTest {  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
        System.out.println("Enter class name (e.g. java.util.Date): ");  
        String name = in.next();  
        try {           // print class name and superclass name (if != Object)  
            Class cl = Class.forName(name);  
            Class supercl = cl.getSuperclass();  
            System.out.print("class " + name);  
            if (supercl != null && supercl != Object.class) {  
                System.out.print(" extends " + supercl.getName());  
            }  
            System.out.print("\n{\n");  
            printConstructors(cl);  
            System.out.println();  
            printMethods(cl);  
            System.out.println();  
            printFields(cl);  
            System.out.println("}");  
        } catch (ClassNotFoundException e) {e.printStackTrace(); }  
        System.exit(0);  
    }  
}
```



Java Reflection (Riflessione) - IV



```
public static void printConstructors(Class cl) {
    Constructor[] constructors = cl.getDeclaredConstructors();
    System.out.println("CONSTRUCTORS:");
    for (Constructor c : constructors) {
        String name = c.getName();
        System.out.print("  " + Modifier.toString(c.getModifiers()));
        System.out.print(" " + name + "(");

        // print parameter types
        Class[] paramTypes = c.getParameterTypes();
        for (int j = 0; j < paramTypes.length; j++) {
            if (j > 0) {
                System.out.print(", ");
            }
            System.out.print(paramTypes[j].getName());
        }
        System.out.println(");");
    }
}
```


Java Reflection (Riflessione) - V



```
public static void printMethods(Class cl) {
    Method[] methods = cl.getDeclaredMethods();
    System.out.println("METHODS:");
    for (Method m : methods) {
        Class retType = m.getReturnType();
        String name = m.getName();

        // print modifiers, return type and method name
        System.out.print("  " + Modifier.toString(m.getModifiers()));
        System.out.print(" " + retType.getName() + " " + name + "(");
        // print parameter types
        Class[] paramTypes = m.getParameterTypes();
        for (int j = 0; j < paramTypes.length; j++) {
            if (j > 0) {
                System.out.print(", ");
            }
            System.out.print(paramTypes[j].getName());
        }
        System.out.println(");");
    }
}
```

ReflectionTest – esecuzione (con java.util.Date)



```
C:\WINDOWS\system32\cmd.exe
Enter class name (e.g. java.util.Date):
java.util.Date
class java.util.Date
{
CONSTRUCTORS:
    public java.util.Date(java.lang.String);
    public java.util.Date(int, int, int, int, int, int, int);
    public java.util.Date(int, int, int, int, int);
    public java.util.Date();
    public java.util.Date(long);
    public java.util.Date(int, int, int);

METHODS:
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public java.lang.Object clone();
    public int compareTo(java.util.Date);
    public volatile int compareTo(java.lang.Object);
    private void readObject(java.io.ObjectInputStream);
    private void writeObject(java.io.ObjectOutputStream);
    private final sun.util.calendar.BaseCalendar$Date normalize(sun.util.calendar.BaseCalendar$Date);
    private final sun.util.calendar.BaseCalendar$Date normalize();
    public static long parse(java.lang.String);
    public boolean after(java.util.Date);
    public boolean before(java.util.Date);
    public int getDate();
    public static java.util.Date from(java.time.Instant);
    public long getTime();
    public static long UTC(int, int, int, int, int, int);
    private static final java.lang.StringBuilder convertToAbbr(java.lang.StringBuilder, java.lang.String);
    private final sun.util.calendar.BaseCalendar$Date getCalendarDate();
    private static final sun.util.calendar.BaseCalendar getCalendarSystem(int);
    private static final sun.util.calendar.BaseCalendar getCalendarSystem(sun.util.calendar.BaseCalendar$Date);
    private static final sun.util.calendar.BaseCalendar getCalendarSystem(long);
    public int getDay();
    public int getHours();
    private static final synchronized sun.util.calendar.BaseCalendar getJulianCalendar();
    static final long getMillisOf(java.util.Date);
    public int getMinutes();
    public int getMonth();
    public int getSeconds();
    private final long getTimeImpl();
```

Verifica tipi per sviluppare metodi robusti - I

Nei metodi che utilizzano il downcast, la reflection permette di verificare il tipo degli oggetti prima di fare dei cast → prevenire eccezioni. Io consiglio di utilizzare tali controlli. Esempio:



```
public class Animale {  
    ...  
    public int compareTo(Object o) {  
        int ris = Integer.MIN_VALUE;  
        if (o != null && o.getClass() == this.getClass()) {  
            // versione robusta: controlla null e tipo  
            Animale a = (Animale) o;  
            ris = nome.compareTo(a.nome);  
        }  
        return ris;  
    } //... continua  
}
```

Verifica tipi per sviluppare metodi robusti - II



```
// ...  
  
public boolean equals(Object o) {  
    boolean ris = false;  
    if (o != null && o.getClass() == this.getClass()) {  
        Animale a = (Animale) o;  
        ris = nome.equals(a.nome);  
    }  
    return ris;  
}  
} // fine classe Animale
```



Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.