Programmazione Java

Ereditarietà, Classe Object, Argomenti a riga di comando, Numeri come oggetti

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

diruscio@di.univaq.it

Sommario

- Ereditarietà
  - `super`
  - Binding dinaminco
  - Overriding metodi
  - Keyword `final`
  - Keyword `abstract`
  - Casting & `instanceof`
- Classe Object
  - `equals()`
  - `toString()`
- Argomenti a riga di comando
- Numeri come oggetti

Ereditarietà

- Idea alla base: E' possibile creare una classe a partire da classi esistenti
- E' una relazione tra una cosa più generale (detta superclasse o padre) ed una più specifica (detta sottoclasse o figlia)
- Viene detta anche relazione "*is-a-kind-of*"
- Oggetti figlio possono essere utilizzati al posto di oggetti padre (principio di sostituibilità di Liskov) **ma non il viceversa**, cioè il padre non è un sostituto per il figlio
- Meccanismo di riuso *white-box*

Ereditarietà

» Java usa `extends` per esprimere che una classe (*derivata*) è una sottoclasse di un'altra (*classe base*)

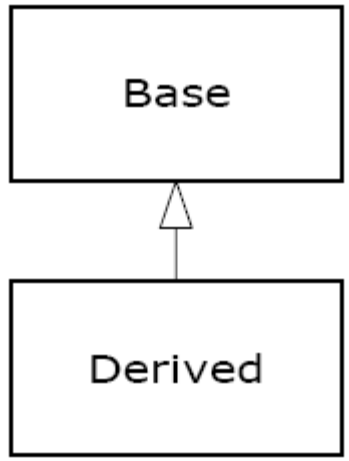
» E' possibile specificare soltanto una superclasse (ereditarietà singola)

» Sottoclasse eredita variabili e metodi dalla sua superclasse e da tutti i suoi predecessori ovvero

- Eredita i membri che sono dichiarati `public` o `protected`
- Eredita i membri dichiarati package se le classi appartengono allo stesso package

» Sottoclasse può utilizzare tali membri così come sono oppure può

- Nascondere le variabili
- O può effettuare l'override dei metodi



Ereditarietà > Esempio (1)


```
class Cleanser {

    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public String toString() { return s; }

    public static void main(String[] args) {
      Cleanser x = new Cleanser();
      x.dilute(); x.apply(); x.scrub();
      System.out.println(x);

   }

}
```

Ereditarietà > Esempio (1)


```
public class Detergent extends Cleanser {

  // Change a method:
    public void scrub() {
     append(" Detergent.scrub()");
     super.scrub(); // Call base-class version
    }

  // Add methods to the interface:
    public void foam() { append(" foam()"); }

  // Test the new class:
    public static void main(String[] args) {
     Detergent x = new Detergent();
      x.dilute();
      x.apply();
      x.scrub();
      x.foam();
      System.out.println(x);
      System.out.println("Testing base class:");
      Cleanser.main(args);
    }
}
```

- `Cleanser` definisce un insieme di metodi che automaticamente vengono derivati da `Detergent` che estende `Cleanser` (…`Detergent` **`extends`** `Cleanser`…)
- `Detergent` ridefinisce il metodo `scrub`. E' possibile invocare il metodo scrub della superclasse `Cleanser` usando la parola chiave `super`
- La classe derivata può aggiungere nuovi metodi (es. Il metodo `foam()` in `Detergent`)

(Vedi Detergent.java)

Ereditarietà > Esempio (2)


```
class Employee {
   private String name;
   private double salary;
   private Date hireDay;

   public Employee(String n, double s, int year,
                   int month, int day) {
      name = n;
      salary = s;
      GregorianCalendar calendar =
                        new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName() {
      return name;
   }

……..
```

Ereditarietà > Esempio (2)


```
.....

    public double getSalary() {
        return salary;
    }

    public Date getHireDay() {
        return hireDay;
    }

    public void raiseSalary(double byPercent) {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

}
```

Ereditarietà > Esempio (2)


```
class Manager extends Employee {

  private double bonus;

  public Manager(String n, double s,
                 int year, int month, int day) {
      super(n, s, year, month, day);
      bonus = 0;
  }

  public double getSalary() {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
  }

  public void setBonus(double b) {
      bonus = b;
  }

}
```

- La classe `Manager` estende `Employee`
- Il costruttore (non di default) di `Manager` invoca il costruttore della classe estesa (`Employee`) mediante

  `super(n, s, year, month, day);`

- `Manager` ridefinisce il metodo `getSalary()` che invova il metodo `getSalary()` della classe `Employee`
- Il metodo `setBonus()`, non presente nell'interfaccia di `Employee` viene creato in `Manager`

Ereditarietà > Esempio (2)


```
public class ManagerTest {
   public static void main(String[] args) {
      // construct a Manager object
      Manager boss = new Manager("Carl Cracker",
                                 80000, 1987, 12, 15);
      boss.setBonus(5000);

      Employee[] staff = new Employee[3];

      // fill the staff array with Manager and Employee objects

      staff[0] = boss;
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);

      // print out information about all Employee objects
      for (int i=0; i < staff.length; i++ )
         System.out.println("name=" + staff[i].getName()
            + ",salary=" + staff[i].getSalary());
   }
}
```

Liskov

Ereditarietà > Esempio (3)

» Da notare

- `boss.setBonus(5000)` `//OK`
- `staff[ 0 ].setBonus(5000)` **`//ERRORE`**
- `Manager m = staff[ i ];` **`//ERRORE`**

Ereditarietà

- L’ereditarietà non si limita a copiare l’interfaccia della classe base nella classe derivata
- Quando si crea un oggetto della classe derivata, questo contiene al suo interno un *sotto-oggetto* della classe base
- E’ essenziale che il *sotto-oggetto* della classe base sia inizializzato correttamente
- Java inserisce automaticamente nel costruttore della classe derivata le chiamate del costruttore della classe base

Ereditarietà > Esempio(4)


```
class Art {
     Art() {
     System.out.println("Art constructor");
   }

}

class Drawing extends Art {
     Drawing() {
     System.out.println("Drawing constructor");
     }
   }

public class Cartoon extends Drawing {

  public Cartoon() {
       System.out.println("Cartoon constructor");
     }

  public static void main(String[] args) {
       Cartoon x = new Cartoon();
     }

}
```

- La classe base viene inizializzata prima che i costruttori della classe derivata possano accedere ad essa
- Verrà eseguito prima il costruttore di `Art` poi quello di `Drawig` ed infine quello di `Cartoon`

(Vedere Cartoon.java)

Ereditarietà

- Se la classe base non ha il costruttore di default, ma costruttori con argomenti allora questi **devono** essere esplicitamente invocati usando la parola chiave `super`
- In particolar, `super`
  - viene utilizzato all'interno di una classe per riferirsi alla superclasse
  - si può utilizzare per riferirsi ad attributi o a metodi
    - `super.x = 100;`
    - `super.getX();`
  - può essere utilizzato all'interno di un costruttore di una classe per invocarne uno della superclasse
    - Se non definito esplicitamente il compilatore invoca automaticamente quello di default

Ereditarietà > Esempio(5)


```
class Game {
     Game(int i) {
     System.out.println("Game constructor");
     }
   }

class BoardGame extends Game {
     BoardGame(int i) {
       super(i);
       System.out.println("BoardGame constructor");
     }
   }

public class Chess extends BoardGame {
   Chess() {
       super(11);
       System.out.println("Chess constructor");
     }
   public static void main(String[] args) {
       Chess x = new Chess();
     }
   }
```

- L'invocazione esplicita del costruttore di `BoardGame` è obbligatoria altrimenti si ha un errore in fase di compilazione in quanto il compilatore non trova il costruttore di default

Ereditarietà > Esempio(5)


```
class Game {
     Game(int i) {
     System.out.println("Game constructor");
     }
   }

class BoardGame extends Game {
     BoardGame(int i) {
       super(i);
       System.out.println("BoardGame constructor");
     }
   }

public class Chess extends BoardGame {
   Chess() {
       super(11);
       System.out.println("Chess constructor");
     }
   public static void main(String[] args) {
       Chess x = new Chess();
     }
   }
```

- L'invocazione esplicita del costruttore di `Game` è obbligatoria altrimenti si ha un errore in fase di compilazione in quanto il compilatore non trova il costruttore di default

Ereditarietà > Esempio(5)


```
class Game {
     Game(int i) {
     System.out.println("Game constructor");
     }
   }

class BoardGame extends Game {
     BoardGame(int i) {
       super(i);
       System.out.println("BoardGame constructor");
     }
   }

public class Chess extends BoardGame {
   Chess() {
       super(11);
       System.out.println("Chess constructor");
     }
   public static void main(String[] args) {
       Chess x = new Chess();
     }
   }
```

La chiamata ai costruttori deve essere la prima cosa da fare all'interno del costruttore della classe derivata

(Vedere Chess.java)

Ereditarietà > Esempio(6)


```
public class Point {
  private int x,y;
  public Point(int x, int y) {
       this.x=x;
       this.y=y;
  }
}
public class Point3D extends Point {
  private int z;
  public Point3D(int x, int y, int z) {
       super(x,y);
       this.z = z;
  }
}
```

(Vedere Point3D.java)

Ereditarietà

» Il compilatore non controlla se gli oggetti membro sono stati inizializzati mentre obbliga a inizializzare le classi base e chiede di farlo proprio all'inizio del costruttore di ogni classe

```
public class PlaceSetting extends Custom {

  private Spoon sp;
  private Fork frk;
  private Knife kn;
  private DinnerPlate pl;

  public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println("PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);

  }
}
```

Queste inizializzazioni non sono obbligatorie, la loro eliminazione non comporterebbe alcun errore di compilazione

Ereditarietà

» Il compilatore non controlla se gli oggetti membro sono stati inizializzati mentre obbliga a inizializzare le classi base e chiede di farlo proprio all'inizio del costruttore di ogni classe

```
public class PlaceSetting extends Custom {

  private Spoon sp;
  private Fork frk;
  private Knife kn;
  private DinnerPlate pl;

  public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2
    frk = new Fork(i + 3
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println("PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);

  }
}
```

L'inizializzazione della classe base è obbligatoria e va fatta all'inizio del corpo del costruttore di `PlaceSetting`

(Vedere PlaceSetting.java)

Ereditarietà: final (1)

- A volte si vuole evitare che si possa formare una sottoclasse
- Tali classi vengono dette *finali*
- Si usa la keywork `final` nella definizione di una classe
- Esempio
  - ```
    final class Executive extends Manager {
      }
    ```
  - Classe `String` è dichiarata final per ragioni di sicurezza
    ```

Ereditarietà: final (2)

» E' possibile dichiarare `final` anche un metodo

- Non è possibile effettuare l'override di tale metodo
- Esempio
  - ```
    class Employee {
       public final String getName() {return name;}

    }
    ```
  - Metodi `getTime` e `setTime` della classe `Calendar` sono dichiarati `final`
    ```

Ereditarietà: abstract (1)

- In alcuni casi esistono classi che hanno senso soltanto da un punto di vista concettuale e non concretamente (ovvero oggetti di tale classe)
- E' necessario uno strumento che permetta di definire tali classi generiche e astratte
- Si usa la keyword `abstract` nella definizione di una classe
- In tal caso non è più possibile creare oggetti di tale classe
- E' possibile dichiarare `abstract` anche i metodi
  - Non viene fornita l'implementazione del metodo ma viene demandata a sottoclassi
- Ovviamente è sempre possibile dichiarare variabili di una classe astratta
- Una classe che ha almeno un metodo `abstract` deve essere dichiarata `abstract`
- E' utile creare classi e metodi astratti, perché rendono esplicita l'astrattezza di una classe e dicono sia all'utente sia al compilatore come si intende utilizzarla

Ereditarietà: abstract (2)

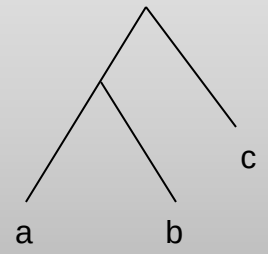
» Esempio 1: Si può posporre la definizione d'un metodo. Ad esempio, gli alberi binari le cui foglie sono etichettate con una stringa possono essere definiti nel modo seguente:

```
class Tree { }

class Node extends Tree {

  Tree right;
  Tree left;
  Node (Tree left, Tree right) {
     this.left = left;
     this.right = right;
  }

}

class Leaf extends Tree {
  String s;
  Leaf (String s) {
     this.s = s;
  }

}
```

Con

```
new Node(new Node(new Leaf("a"), new Leaf("b")), new Leaf("c"))
```

viene creato il seguente albero



(Vedere Abstract.java)

Ereditarietà: abstract (3)

» Per calcolare il numero delle foglie di un albero, si potrebbe pensare di usare il codice seguente che però da errore

```
class Node extends Tree {
  ...
  int getSize() {
     return left.getSize() + right.getSize();
  }
}

class Leaf extends Tree {
  ...
  int getSize() {
     return 1;
  }
}
```

perché `left` è di tipo `Tree` che non definisce il metodo `getSize`

Ereditarietà: abstract (4)

» Il problema si risolve definendo abstract la classe `Tree` ed il metodo `getSize` come segue:

```
abstract class Tree {
  abstract int getSize();
}
class Node extends Tree {
  ...
  int getSize() {
     return left.getSize() + right.getSize();
  }
}
class Leaf extends Tree {
  ...
  int getSize() {
     return 1;
  }
}
```

(Vedere Abstract2.java)

Ereditarietà: abstract (5)

» Esempio 2:

```
abstract class Person {
    private String name;

    public Person(String n) {
        name = n;
    }

    public abstract String getDescription();

    public String getName() {
        return name;
    }
}
```

(Vedere PersonTest.java)

Ereditarietà: abstract (6)


```
class Employee extends Person {
   private double salary;
   private Date hireDay;
   public Employee(String n, double s, int year, int month, int day) {
      super(n);
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }
   public double getSalary() {
      return salary;
   }
   public Date getHireDay() {
      return hireDay;
   }

   public String getDescription() {
      return "an employee with a salary of " + salary;
   }

   public void raiseSalary(double byPercent) {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

}
```

(Vedere PersonTest.java)

Ereditarietà: abstract (7)


```
class Student extends Person {
   private String major;

   public Student(String n, String m) {
      super(n);
      major = m;
   }
   public String getDescription() {
      return "a student majoring in " + major;
   }
}
public class PersonTest {
   public static void main(String[] args) {
      Person[] people = new Person[2];

      people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      people[1] = new Student("Maria Morris", "computer science");

      for (int i=0; i < people.length; i++)
         System.out.println(people[ i ].getName() + ", " +
                            people[ i ].getDescription());
   }
}
```

(Vedere PersonTest.java)

Ereditarietà: Overriding (1)

» Meccanismo che permette di ridefinire nella sottoclasse l'implementazione di metodi definiti nella superclasse

```
class A {
  A(){ }
  int method() {
    return 1;
  }
}

class B extends A {
  B() { }
  int method() {
    return 2;
  }
}
```

In questo esempio il comportamento di `method` è stato cambiato in `B`. Infatti:

`new A().method()` vale 1

ma

`new B().method()` vale 2.

Ereditarietà: Overriding (2)

- Meccanismo che permette di ridefinire nella sottoclasse l'implementazione di metodi definiti nella superclasse
- **Regola**
  - Metodo di istanza `m1` dichiarato all'interno della classe `C` override un altro metodo con la **stessa segnatura** `m2` di una classe `A` se e solo se
    - `C` è una sottoclasse di `A`
    - `m2` è un metodo non privato ed accessibile da `C` oppure `m1` ovverride `m3` (distinto da `m1` e da `m2`) tale che `m3` ovverride `m2`
- Se m1 non è **astratto** si dice che m1 implementa tutti i metodi astratti di cui effettua l'override

Ereditarietà: Overriding (3)

» Inoltre

- Un metodo di istanza non può effettuare l'override di metodi statici
- Tipo di ritorno deve essere lo stesso
- Modificatore di accesso deve essere almeno lo stesso accesso del metodo di cui viene effettuato l'override ovvero
  - Se è `public` deve rimanere tale
  - Se `protected` può essere `protected` o `public`
  - Se è *package* allora non può essere `private`

(Vedi SlowPoint.java)

Ereditarietà: Overriding (4)


```
public class Point {
   int x = 0, y = 0;
    void move(int dx,int dy) {
          x += dx;
          y += dy;
    }
}

class SlowPoint extends Point {
   int xLimit, yLimit;
   void move(int dx, int dy) {
          super.move(limit(dx,xLimit),
                     limit(dy,yLimit) );
    }

   static int limit( int d, int limit) {
          return (d > limit) ? limit : ( (d < - limit) ? limit : d );
   }

}
```

(Vedi SlowPoint.java)

Ereditarietà: Overriding (5)


```
public class Point {
  int x = 0, y = 0, color;
  void move(int dx,int dy) {
        x += dx;
        y += dy;
  }
  int getX() { return x; }
  int getY() { return y; }
}

class RealPoint extends Point {
  float x = 0.0f, y = 0.0f;
  void move(int dx, int dy) {
        move( (float)dx, (float)dy );
  }
  void move(float dx, float dy) {
        x += dx; y+=dy;
  }

  float getX() { return x; }            //ERRORE il tipo di ritorno è diverso dal
                                          metodo getX in Point
  float getY() { return y; }            //ERRORE il tipo di ritorno è diverso dal
                                          metodo getY in Point
}
```

(Vedi RealPoint.java)

Ereditarietà: Overriding (6)

» Nel caso di metodi **statici** non si parla di ovverride di metodi ma di *hide* di metodi
» Inoltre, un metodo statico non può effettuare l'hide di metodi di istanza

Esempio

```
class Base {
  static String greeting() { return "Goodnight";}
  String name() { return "Richard";}
}

class Sub extends Base {
  static String greeting() { return "Hello";}
  //! static String name() { return "Dick";} //Errore
  String name() { return "Dick";}
}

public class OverridingTest {
  public static void main(String[] args) {
        Base b = new Sub();
        System.out.println(b.greeting() + ", " + b.name());
   }
}
```

Output

Goodnight, Dick

Ereditarietà: Overriding (7)

» Nel caso di metodi **statici** non si parla di ovverride di metodi ma di *hide* di metodi
» Inoltre, un metodo statico non può effettuare l'hide di metodi di istanza

Esempio

```
class Base {
  String greeting() { return "Goodnight";}
  String name() { return "Richard";}
}

class Sub extends Base {
  String greeting() { return "Hello";}
  //! static String name() { return "Dick";} //Errore
  String name() { return "Dick";}
}

public class OverridingTest {
  public static void main(String[] args) {
        Base b = new Sub();
        System.out.println(b.greeting() + ", " + b.name());
   }
}
```

Output

```
Hello, Dick
```

Ereditarietà: Overriding (8)


```
public class PrivateOverride {

    private void f() {
        System.out.println("private f()");
      }

  public static void main(String[] args) {
      PrivateOverride po = new Derived();
      po.f();
    }
  }

class Derived extends PrivateOverride {

  public void f() {
      System.out.println("public f()");
  }

}
```

- Attenzione alla ridefinizione di metodi private
- Ci si potrebbe aspettare che l'output dell'esempio sia "`public f()`"
- Il metodo privato `f()` in `PrivateOverride` è nascosto alla classe derivata
- Il metodo `f()` in `Derived` è un metodo completamente nuovo, non è una redifinizione del metodo della classe base perchè quella versione di `f()` non è visibile in `Derived`

Ereditarietà > Invocazione di metodi

» L'invocazione di un metodo di istanza di un oggetto è composto di due fasi
  – Compilazione
    • Viene stabilito **se** un metodo può essere invocato
    • Viene risolto utilizzando il tipo della variabile dell'oggetto
  – Esecuzione
    • Viene stabilito **quale** metodo invocare
    • Viene risolto utilizzando il tipo dell'oggetto e il nome del metodo

» Java determina *quasi* sempre quale metodo invocare a tempo di esecuzione (*binding dinamico* o *late-binding*)
  – Metodi statici, privati e final hanno binding statico
  – Costruttori anch'essi hanno binding statico

» Altri linguaggi hanno sia binding dinamico che statico (C++)

Ereditarietà > Invocazione di metodi

» Fase compilazione

- `x.f(param)` dove `x` è una variabile di tipo `C`
- Se f è un metodo con overloading (ovvero più metodi con parametri diversi) si sceglie quello appropriato (*risoluzione dell'overloading*)
  - Esempio: `x.f("Hello")` seleziona `f(String)` e non `f(int)`
- E' possibile applicare anche conversioni di tipi

Ereditarietà > Invocazione di metodi

» Fase esecuzione

- `x.f(param)` durante esecuzione x contiene oggetto di tipo `D` (sottoclasse di `C`)
  - Se `D` contiene `f` (ovvero **ovverride** di f) invocazione di tale metodo
  - Altrimenti si sale nella gerarchia fino ad arrivare a `C`

Ereditarietà > Invocazione di metodi


```
public class ManagerTest1 {
   public static void main(String[] args) {
    Employee employee;
    if( (args.length==1) && (args[0].equals("Employee"))) {
       employee = new Employee("Harry Hacker",50000, 1989, 10, 1);
    } else {
       employee = new Manager("Carl Cracker", 80000, 1987, 12, 15);
    }

      employee.setBonus(5000); //ERRORE perché setBonus non e' un metodo
                               //dell'interfaccia di Employee

    System.out.println("name=" + employee.getName() +
                       ",salary=" + employee.getSalary());

   }
}
```

(Vedi ManagerTest1.java)

Casting & instanceof (1)

» Operatore di cast `()` può essere applicato anche ai tipi reference

» Se si assegna un riferimento a una sottoclasse in una variabile della superclasse no problem (Liskov)

```
Employee employee = new Employee(……);
Person p = employee;
```

» Il contrario non è possibile farlo a meno che non si utilizzi il cast

- `Employee employee = (Employee) people[ 0 ];`

» Non è possibile effettuare il casting tra fratelli o più in generale tra elementi non facente parte della gerarchia

- `Student student = new Student(……);`
- `Employee emp = (Student) student;  //ERRORE`

Casting & instanceof (2)

» Se `people[ 0 ]` contiene un oggetto che non è di tipo `Employee` a run-time viene lanciata un'eccezione `ClassCastException`

» Per ovviare a questo problema si utilizza l'operatore `instanceof`

```
Employee employee = null;
if (people[ 0 ] instanceof Employee)
  employee = (Employee) people[ 0 ];
```

Classe Object (1)

» Ogni classe in Java estende da `Object` anche se non si è obbligati a farlo ovvero

- `public class Employee extends Object`

» E' possibile

- Dichiarare variabili di tipo `Object`
  - `Object o = new Employee("Harry Hacker", 35000);`
- Creare oggetti di tipo `Object`
  - `Object o = new Object();`

» Contiene una serie di metodi che possono essere utilizzati e/o sovrascritti (override)

Classe Object (2)

- Metodi che possono essere sovrascritti
  - `clone`
  - `equals`
  - `hashCode`
  - `finalize`
  - `toString`
- Metodi final
  - `getClass`
  - `notify`
  - `notifyAll`
  - `wait`

Classe Object: equals (3)

» `public boolean` **`equals`**`(Object obj)`

- Verifica se un oggetto può essere considerato *uguale* ad un altro
- Operatore `==` applicato a tipi reference verifica se le due variabili *puntano* allo stesso oggetto
- Metodo classe `Object` verifica l'uguaglianza tra i due riferimenti (equivale a `==`)
- Per modificare il comportamento è necessario effettuare l'override del metodo
- Nota: è necessario dichiarare il metodo dove il tipo del parametro formale è lo stesso (`Object`)
- Generalmente se si effettua l'override di `equals` lo si fa anche di `hashCode`

Classe Object: equals (4)

» Regole per implementare `equals`

- Riflessiva: `x.equals(x)` deve restituire `true`
- Simmetrica: `x.equals(y)` è `true` se e solo se `y.equals(x)` è `true`
- Transitiva: se `x.equals(y)` è `true` e `y.equals(z)` è `true` allora `x.equals(z)` è `true`
- Coerente: se gli oggetti non cambiano allora `x.equals(y)` restituisce sempre lo stesso valore
- `x.equals(null)` restituisce sempre `false`

Classe Object: equals (5)

» Per rispettare tali regole è sufficiente implementare il metodo nel seguente modo

- Utilizzare l'operatore == per verificare se l'argomento è un riferimento all'oggetto stesso
  - `if ( this == obj) return true;`
- Utilizzare l'operatore `instanceof` per verificare se l'argomento è del tipo corretto. Se non lo è restituire `false`
- Effettuare il casting al tipo corretto
- Per ogni campo significativo della classe controllare se corrisponde a quello dell'argomento passato
  - Se il test va a buon fine restituire `true` altrimenti `false`
- Come ultimo passo vedere se sono valide le regole precedenti!!! ☺

Classe Object: equals (6)


```
public final class PhoneNumber {
   private final short areaCode;
   private final short exchange;
   private final short extension;
   public PhoneNumber(int areaCode, int exchange,int extension) {
     rangeCheck(areaCode, 999, "area code");
     rangeCheck(exchange, 999, "exchange");
     rangeCheck(extension, 9999, "extension");
     this.areaCode = (short) areaCode;
     this.exchange = (short) exchange;
     this.extension = (short) extension;
   }
   private static void rangeCheck(int arg, int max, String name) {
  if (arg < 0 || arg > max)
        throw new IllegalArgumentException(name +": " + arg);
   }
   public boolean equals(Object o) {
     if (o == this) return true;
     if (!(o instanceof PhoneNumber)) return false;
     PhoneNumber pn = (PhoneNumber)o;
     return pn.extension == extension && pn.exchange == exchange &&
            pn.areaCode == areaCode;
   }
}
```

(Vedi PhoneNumber.java)

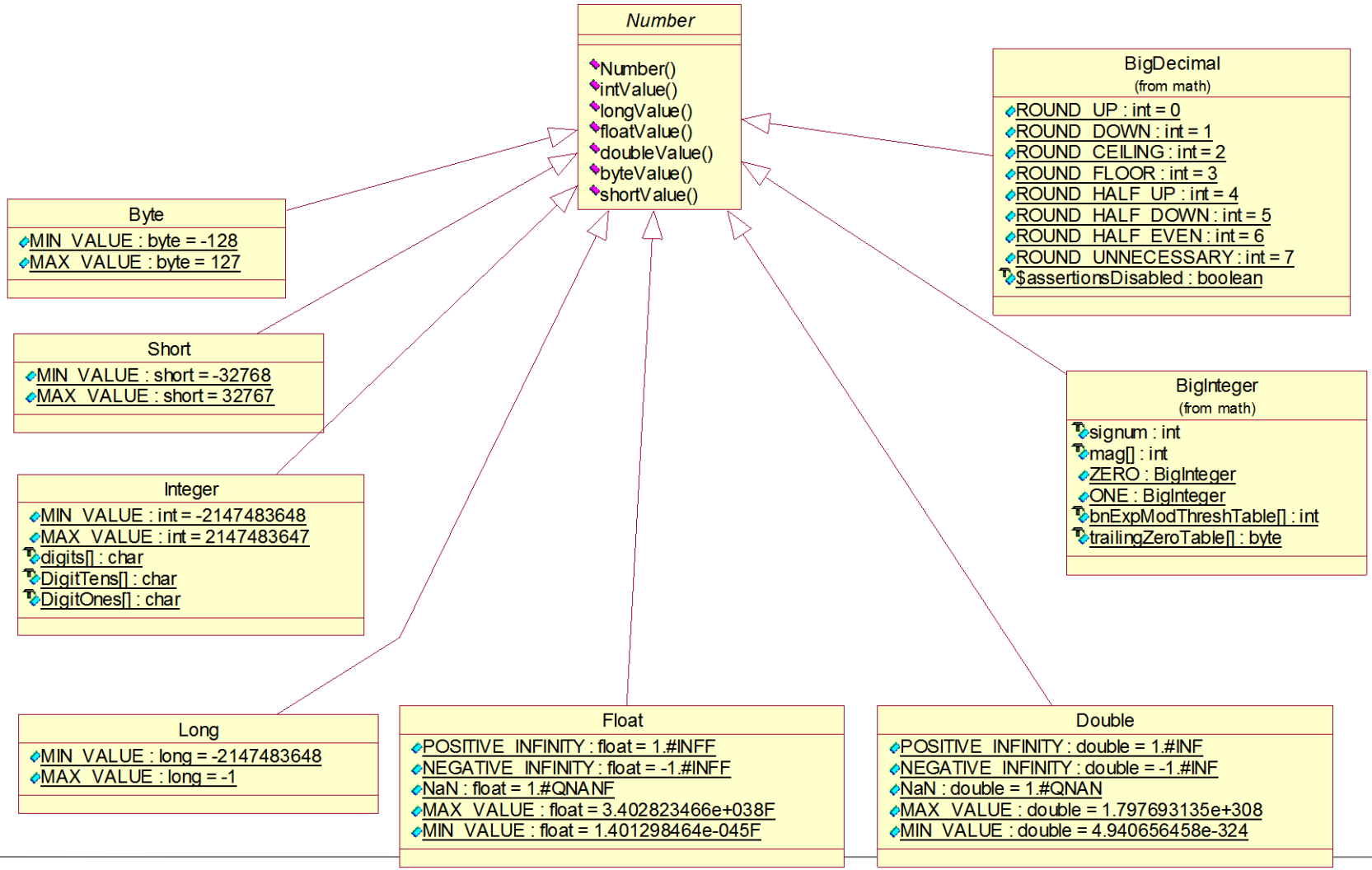
Classe Object: toString (7)

» `public String toString()`

- E’ buona norma effettuare l’override di tale metodo
- Implementazione di default della classe `Object` restituisce il nome della classe seguito da @ e dalla rappresentazione esadecimale senza segno di ciò che restituisce hashCode
- Nel caso precedente ha senso che il metodo restituisce il numero di telefono nella forma (xxx) xxx-xxxx

(Vedi PhoneNumber.java)

Numeri come Oggetti (1)



Numeri come Oggetti (2)

» Integer

- Wrappa un valore di un tipo primitivo `int`
- Valore è **immutabile** ovvero una volta creato l'oggetto non è più possibile modificarlo
- Particolarmente utile per le collezioni (`ArrayList`, `Vector`, `Hashtable`) in quanto è possibile inserire soltanto oggetti
- Sono presenti una serie di metodi per convertire `String` in `int` e viceversa
- Esempi

```
Integer i = new Integer(10);
int i = Integer.parseInt("10");
float f = Float.parseFloat("9.4");
```