Tipi numerici di base - Costanti

Interi e decimali

Definizione di costanti

Una classe Borsellino

```
/** Realizza un borsellino per le monete. Registra il
   numero di monete e calcola il valore totale
*/
public class Purse  {
   /** Costruisce un borsellino vuoto
   */
   public Purse() {
   }

   /** Aggiunge monete di tipo "nickel" al borsellino
   @param count il numero di nickel da aggiungere
   */
   public void addNickels(int count) {
   }
```

Classe Borsellino cont'd

```
  /** Aggiunge monete di tipo "dime" al borsellino
  @param count il numero di dime da aggiungere
  */
  public void addDimes(int count) {
  }
  /** Aggiunge monete di tipo "quarter" al borsellino
  @param count il numero di quarter da aggiungere
  */
  public void addQuarters(int count) {
  }
  /** Ispeziona il valore totale delle monete nel borsellino
    @returns la somma dei valori di tutte le monete attualmente
   presenti
    */
  public double getTotal() {
  }
}
```

Esempio d'uso

```
Purse myPurse = new Purse();
myPurse.addNickels(3);
myPurse.addDimes(1);
myPurse.addQuarters(2);
double totalValue  =
                    myPurse.getTotal();
```

- **totalValue** conterrà 0,75 cioè il valore in dollari delle monete contenute nel borsellino

Numeri interi e numeri decimali

- Per rappresentare quantità che si contano naturalmente con multipli di 1 usiamo variabili di tipo intero (**`int`**)
- Nell'esempio del borsellino **`int`** è il tipo dei parametri dei metodi **`addXXX`**
- Questo perché è naturale considerare le monete come quantità indivisibili e quindi quantificate con un numero intero
- Il metodo **`getTotal()`** invece restituisce un **`double`**
- Questo perché è naturale per un valore che rappresenta una quantità di dollari avere dei decimali

Primi passi per l'implementazione di Purse

- Ogni oggetto di tipo Purse può essere descritto dalla quantità di nickels, dimes e quarters che contiente
- Inseriamo quindi tre variabili istanza di tipo int per rappresentare queste quantità

```
...
private int nickels;
private int dimes;
private int quarters;
```

Primi passi per l'implementazione di Purse

```
public double getTotal() {
    return nickels * 0.05 + dimes * 0.01
                + quarters * 0.25;
}
```

- L' $*$ indica la moltiplicazione (perché • o $\times$ non si trovano generalmente nelle tastiere)
- L'espressione dopo **`return`** segue le regole di associatività e precedenza tipiche dell'aritmetica (la grammatica di Java per le espressioni segue le regole che abbiamo visto a Programmazione)
- Il valore ottenuto è un numero in virgola mobile poiché moltiplicando un intero per un numero decimale si ottiene un numero decimale, in generale

Costanti numeriche

- Nelle costanti numeriche che si possono scrivere nel codice la virgola deve essere indicata come punto decimale
- Si può usare anche la notazione esponenziale
- Ad esempio $5{,}0 \times 10^{-3}$ si scrive come **`5.0E-3`**
- Cioè si usa il punto decimale e si scrive **`E`** seguito dall'esponente di $\times 10$

Numeri interi

- Un numero intero è un numero senza decimali che può essere sia positivo che negativo
- Il tipo base Java corrispondente ai numeri interi è **`int`**
- Una variabile **`int`** può contenere i numeri interi da –2147483648 a +2147483647
- 31 bit + 1 bit per il segno = 32 bit di memoria allocati per ogni variabile **int**

Numeri interi

- Esistono altri tipi base di interi che possono rappresentare meno o più numeri di **`int`**
- **`short`**: 16 bit - Range: da -2^{15} a $2^{15}-1$
- **`byte`**: 8 bit - Range da -2^{7} a $2^{7}-1$
- **`long`**: 64 bit - Range da -2^{63} a $2^{63}-1$

Numeri in virgola mobile

- Possono contenere cifre decimali
- Contengono un certo numero di cifre significative e la posizione della virgola
- Es. 250 25 0.25 0.025 hanno tutti le stesse cifre significative (25), ciò che cambia è la posizione della virgola (da qui “virgola mobile”)
- Naturalmente la rappresentazione in realtà è in base 2
- In java la virgola è rappresentata dal punto decimale come nella notazione anglo-sassone (come tutte le calcolatrici del resto)

Numeri in virgola mobile

- **`double`** può rappresentare circa 15 cifre significative: è il tipo con maggiore precisione (“doppia precisione”)
- **`float`** può rappresentare circa 7 cifre significative: precisione spesso insufficiente, ma calcoli più veloci
- Per precisione si intende la grandezza degli errori dovuti all’ arrotondamento che inevitabilmente si commettono con l’uso di questi numeri

Precisione

```
public class ProvaPrecisione {
  public static void main(String [] argv) {
    double originalPrice = 3E14;
    double discountedPrice =
          originalPrice - 0.05;
    double discount = originalPrice -
                      discountedPrice;
    // dovrebbe essere 0.05
    System.out.println(discount);
    // stampa 0.0625 - Errore dovuto
  //all'arrotondamento da rappresentazione
  }
}
```

Numeri rappresentabili

- Il tipo float può rappresentare il range dei numeri, positivi o negativi, con valore assoluto che va da 2^{-149} a $(2-2^{-23})\bullet 2^{127}$
- Il tipo double può rappresentare il range dei numeri, positivi o negativi, con valore assoluto che va da 2^{-1074} a $(2-2^{-52})\bullet 2^{1023}$

Numeri a lunghezza e precisione arbitraria

- Il pacchetto **`java.math`** contiene una classe **`BigInteger`** i cui oggetti possono rappresentare numeri interi di lunghezza arbitraria
- Lo stesso pacchetto contiene una classe **`BigDecimal`** i cui oggetti possono rappresentare numeri a virgola mobile con precisione arbitraria

Numeri a lunghezza e precisione arbitraria

- Per questi numeri non si possono usare i normali operatori + * - / =
- I corrispondenti metodi add, multiply, subtract, divide ed equals sono forniti dalle relative classi:

```
BigInteger a = new BigInteger("123456789");
BigInteger b = new BigInteger("987654321");
BigInteger c = a.multiply(b);
System.out.println(c);
// stampa  121932631112635269
```

- Naturalmente i calcoli con questi numeri sono più lenti di quelli fatti con i numeri dei tipi base

Costruttore di Purse

```
Public Purse() {
   nickels = 0;
   dimes = 0;
   quarters = 0;
}
```

- Il costruttore di default farebbe esattamente la stessa cosa, ma lo specifichiamo per chiarezza

Implementazione di addNickels

```
public void addNickels(int count) {
   nickels = nickels + count;
}
```

- È la tipica istruzione di **incremento** di un valore
- L'assegnamento prima valuta la parte a sinistra dell'= e quindi considera il **valore corrente** della variabile istanza intera **`nickels`** al quale aggiunge il valore di **`count`**
- Il valore così ottenuto sarà assegnato alla variabile istanza **`nickels`**, cioè sarà il suo **valore dopo l'esecuzione** dell'istruzione (si può pensare che sia il valore che **`nickels`** ha dopo il ; finale)

Altre istruzioni di incremento

- In Java, come in C, esiste una forma abbreviata per l'incremento

```
nickels = nickels + count;
```

può essere scritta equivalentemente come

```
nickels += count;
```

- La stessa abbreviazione si può usare anche per gli altri operatori (*, /, -):

```
P *= q; // equivalente a p = p * q;
```

Incremento di 1

- Un'istruzione che ricorre molto frequentemente nei programmi è l'incremento o il decremento di una variabile intera di una unità
- In Java, come in C, esistono abbreviazioni speciali per questi casi:

`i++;` // equivalente a `i = i + 1;` e a `i +=1;`

`i--;` // equivalente a `i = i - 1;` e a `i -=1;`

Costanti

```
public double getTotal() {
    return nickels * 0.05 +
           dimes *  0.01 +
           quarters * 0.25;
}
```

- La maggior parte del codice si documenta da sé, ma in questo caso i numeri 0.05, 0.01 e 0.25 compaiono senza nessuna spiegazione
- È buona regola evitare di inserire nel codice questi “numeri magici”

Costanti

- Le costanti possono essere pensate come dei nomi a cui è associato un valore
- Tale associazione rimane valida per tutta la vita della costante e non cambia mai
- In genere, per convenzione, le costanti sono scritte tutte maiuscole e si usa l'underscore come separatore nel caso che il nome sia composto da più parole
- In Java una costante si dichiara come una variabile (di frame) usando la parola chiave **`final`** e inizializzando il valore nella dichiarazione

Costanti

```
public double getTotal() {
   final double NICKEL_VALUE = 0.05;
   final double DIME_VALUE = 0.01;
   final double QUARTER_VALUE = 0.25;
   return nickels * NICKEL_VALUE +
          dimes * DIME_VALUE +
          quarters * QUARTER_VALUE;
}
```

final

- In generale in Java la parola chiave final indica un qualcosa che non può essere più modificato (vedremo che esistono, oltre alle variabili, anche classi final, cioè che non possono essere estese)
- Il compilatore dà errore se si cerca di modificare una variabile final.

Costanti

- La dichiarazione di una variabile con lo specificatore final ha lo stesso effetto di una qualsiasi altra definizione di variabile
- Viene cioè allocata la nuova variabile sul frame corrente dell'attivazione corrente
- Al momento della chiusura del blocco del frame in cui è stata definita essa scompare
- E' il controllo del compilatore che la rende non modificabile

Costanti

- In genere però le costanti sono utili in diversi metodi della stessa classe
- Per evitare di dover ridefinire le variabili final in ogni metodo che le usa (ed evitare anche errori se il valore viene modificato solo in alcuni metodi, ad esempio in un'altra versione del programma) possiamo pensare di associare le costanti direttamente alla classe
- Per definire costanti che si riferiscono ad una classe si può usare lo specificatore **`static`**

static

- Lo specificatore static deriva dal C e il suo nome può risultare fuorviante
- In Java se nella definizione di una classe viene inserita una variabile istanza con lo specificatore static quella variabile istanza va considerata come elemento della classe
- In genere le variabili istanza formano lo stato di ogni oggetto della classe che viene creato
- Le variabili istanza static invece non vanno a far parte dello stato degli oggetti della classe

static

- Esiste una sola copia di una variabile istanza static di una classe e si riferisce all'intera classe
- Per riferirla e/o modificarla si usa la notazione NomeClasse.nomeVariabileIstanzaStatic
- Alle variabili istanza static possono essere associati tutti i possibili specificatori di accesso: **`public`**, **`private`** e **`protected`**
- Inoltre possono essere **`final`**, cioè costanti di classe

Esempio

- Vedremo più avanti degli esempi in cui sono utili variabili static di classe
- Per adesso useremo questa possibilità solo per specificare costanti
- Nel nostro caso le costanti che indicano il valore di ogni moneta sembrano essere utili solo nel contesto della classe Purse e quindi le dichiariamo private
- Molte classi delle API hanno invece delle costanti pubbliche che si possono usare

Esempio

```
public class Purse {
...
private static final double
                NICKEL_VALUE = 0.05;
private static final double
                DIME_VALUE = 0.01;
private static final double
                QUARTER_VALUE = 0.25;
...
}
```

Esempio

- All'interno dei metodi della classe si possono riferire le variabili static (sia final che non) semplicemente con il loro nome (nel caso di conflitto va invece specificato il nome completo (cioè NomeClasse.nomeVariabile), ad esempio Purse.NICKEL_VALUE

```
public double getTotal() {
   return nickels * NICKEL_VALUE +
          dimes * DIME_VALUE +
          quarters * QUARTER_VALUE;
}
```