Codifica degli Interi

Insieme degli interi Z

Si può definire l'insieme Z a partire dall'insieme N dei naturali definendo un insieme di coppie costituite da un naturale e dal suo opposto (il naturale con il segno meno).

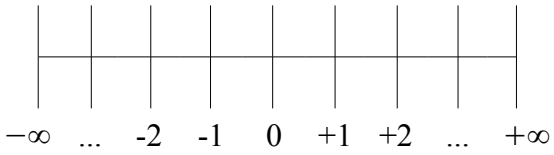
Sono quindi definite le coppie (-1,1),(-2,2),(-3,3) ...

La proprietà di ogni coppia è che la loro somma produce l'elemento neutro della somma (1-1=0, 2-2=0, ...) .

L'elemento 0 fa parte dell'insieme Z ma, essendo l'elemento neutro, si trova in una situazione particolare perché non ha un elemento di coppia.

Per questo motivo lo zero in matematica non viene considerato né positivo né negativo ma vedremo che nella codifica binaria dei sistemi di elaborazione lo zero assume significato di numero **positivo**.

L'insieme Z della matematica invece si potrebbe rappresentare su una retta con il seguente schema:

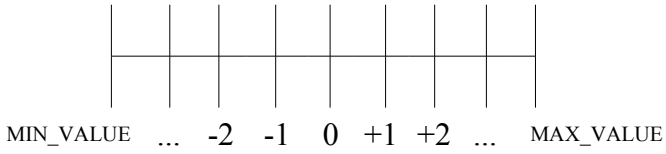


Non è possibile realizzare una codifica binaria di questo insieme che è formato da un numero infinito di elementi.

Inoltre l'alfabeto della codifica decimale ha ora due nuovi simboli "+" e "-" mentre la codifica binaria deve essere realizzata sono con due simboli "0" ed "1" sia per le cifre numeriche sia per gli altri simboli.

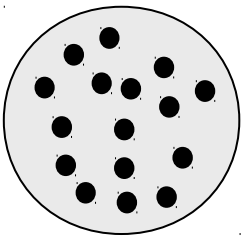
Infine in una codifica binaria si deve risolvere l'ambiguità dello zero che in matematica non è né positivo né negativo mentre in informatica deve essere definito come positivo.

Si deve quindi realizzare una rappresentazione che, usando una precisione predefinita abbia un numero finito di codici:

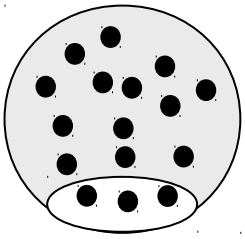


Codifica in complemento-a-2

Si potrebbe riservare un bit della codifica per il segno (codifica a modulo-e-segno) ma in genere la codifica degli interi segue una diversa regola.
In una codifica binaria a precisione finita è possibile rappresentare i numeri interi mediante una codifica in complemento-a-2.
Per comprendere la codifica in complemento-a-2 si può fare riferimento ad una rappresentazione insiemistica degli interi.
Supponiamo di voler rappresentare in binario un sotto-insieme di Z con una precisione di 4 bit cioè di voler utilizzare 4 cifre binarie.
La quantità totale di informazioni rappresentabili è $\mathbf{2^4=16}$
In un diagramma di Eulero-Venn così:



Poiché l'insieme Z è formato di coppie (+x,-x) possiamo immaginare di rappresentare l'elemento positivo di ciascuna coppia con un sottoinsieme formato da un numero di punti pari al naturale corrispondente. La parte rimanente dell'insieme si chiama "complemento" perché contiene la quantità di punti tale per arrivare al totale dell'insieme.



Ad esempio se consideriamo la coppia (+3,-3) si ottiene:
Il sotto-insieme bianco rappresenta l'elemento +3 mentre tutto il rimanente insieme esclusa la parte bianca rappresenta il "complemento di +3" cioè -3 in una rappresentazione di 16 elementi.

Espressa in forma numerica la partizione dell'insieme in due sotto-insieme diventa:

Complemento di X = B^N-X

Quindi:

Complemento di 3 = B^N - 3 = 16 - 3 = 13

Espressa in decimale la relazione appare poco significativa ma proviamo ad applicarla da una codifica a quattro bit in binario:
B^N in binario è rappresentato da 5 bit: 10000.
Il numero naturale 3 è rappresentato da 4 bit: 0011
Effettuiamo la sottrazione in colonna seguendo le normali regole della sottrazione nei sistemi di numerazione posizionale:

```
 0111
-10000 -
  0011 =
----------
  1101
```

Per effettuare la sottrazione della colonna meno significativa è necessario prendere un prestito dalla cifra successiva che però è 0 quindi si prende un ulteriore prestito dalla successiva fino ad arrivare all'ultima che scompare.

A questo punto si può fare la sottrazione della prima colonna (10-1 =1). La sottazione della seconda colonna produce 0 (1-1=0). La sottrazione della terza e quarta colonna producono 1 (1-0=1). Il risultato è 1101 che in effetti se fosse interpretato come naturale sarebbe 13 ma in una codifica "complemento-a-2" invece rappresenta -3.
Per verificare questa corrispondenza si può applicare la definizione di intero che dice che la somma dei due elementi di coppia deve produrre l'elemento neutro, cioè deve essere +3+(-3)=0: effettuiamo la somma in colonna in binario:

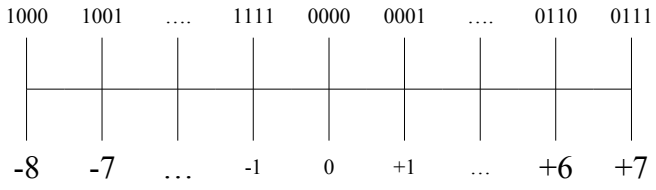
```
 +111
  0011 +
  1101 =
----------
 10000
```

La somma produce dei riporti; l'ultimo riporto avviene sulla quinta cifra che non fa parte di una rappresentazione a quattro bit; considerando il risultato nella precisione a 4 bit abbiamo in effetti ottenuto l'elemento neutro 0.

Codifichiamo ora tutte le coppie di numeri interi di una precisione a quattro bit:

Codifica	*Significato in N*	*Significato in Z*
0000	0	+0
0001	1	+1
0010	2	+2
0011	3	+3
0100	4	+4
0101	5	+5
0110	6	+6
0111	7	+7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

In una rappresentazione su un asse degli interi si ottiene:



Questa codifica, a prima vista poco intuitiva ma presenta alcuni vantaggi che vengono sfruttati nel calcolo automatico:

- Le operazioni di sottrazione non sono operazioni diverse dalle somme ma possono essere eseguite come somme del minuendo con il complemento-a-due del sottraendo:

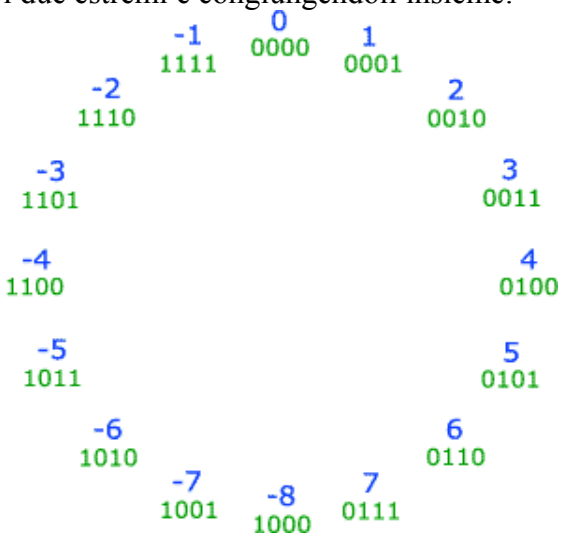
  Diff = Minuendo - Sottraendo = Minuendo + C_2(Sottraendo)

  Questa regola semplifica l'architettura interna delle CPU la cui logica non deve prevedere due distinte operazioni ma solo una.

- Esiste una unica codifica per il valore 0; se si fosse adottata una codifica a modulo-e-segno si sarebbero ottenute due codifiche (+0,-0) in contrasto con la definizione di 0 degli interi.

- Il riconoscimento del segno è comunque immediato perché tutti i numeri positivi iniziano per 0 e tutti quelli negativi per 1 come nella codifica modulo-e-segno. La conseguenza è che lo zero risulta positivo.

Per comprendere meglio il significato di questa codifica conviene anche in questo caso rappresentare l'asse in modo circolare prendendo i due estremi e congiungendoli insieme:



In questa visione i numeri negativi si trovano ordinati rispetto allo zero.
Fare operazioni di somma e sottrazione su numeri interi diventa una rotazione in senso orario (somma) o antiorario (sottrazione) Ad esempio per eseguire l'operazione +1-3 si parte dalla posizione +1 e ci si sposta in senso antiorario di 3 posizione fino a raggiungere la posizione -2.
Per effettuare invece l'operazione 2+3 si parte dalla posizione +2 e ci si sposta in senso orario di tre posizione fino a raggiungere la posizione +5.
Da notare che spostamenti che scavalchino i limiti di MAX_VALUE (+7) e MIN_VALUE (-8) producono una situazione di overflow.
Ad esempio +7+3 che determina tre spostamenti orari a partire da +7 produce -6.
La presenza dello zero come numero positivo produce una asimmetria nel valori estremi dei due campi.
Sebbene si il bit più significativo della codifica rifletta il segno (0 => positivo, 1 => negativo) non si può dire che il segno è solo nel bit più significativo.
Infatti il segno si estende per tutti i bit a partire da sinistra fino al primo bit significativo quindi ad esempio il valore +2 codificato a 4 bit con 0010 ha il segno esteso nei suoi 2 bit più significativi mentre il valore -2 codificato ad 4 bit con 1110 ha il segno esteso nei suoi tre bit più significativi ma se si passa ad una codifica ad 8 bit la situazione diventa rispettivamente 0000010 (6 bit di segno positivo) e 11111110 (7 bit di segno positivo).

Metodo pratico di calcolo delle codifiche in complemento-a-due

Data una codifica in complemento-a-due in precisione N è possibile passare da un numero positivo al corrispondente negativo con un metodo rapido che non richiede l'effettuazione della sottrazione ma solo l'uso della regola di complemento-a-uno ed una somma.

Il complemento-a-uno è la complementazione di ogni singola cifra del numero (i bit ad 1 diventano 0 ed i bit a 0 diventano 1). Al valore

complementato ad uno si aggiunge 1 ottenendo il complemento-a-due.
Esempi:

Dato il numero ad 8 bit si vuole determinare il corrispondente numero di segno opposto espresso in complemento-a-due:

```
Numero  00000001  (+1)  Numero  11111110  (-2)  Numero 00000000  (+0)
Cpl-1   11111110 +      Cpl-1   00000001 +      Cpl-1  11111111 +
               1 =                      1 =                    1 =
Cpl-2   11111111  (-1)  Cpl-2   00000010  (-2)  Cpl-2 100000000  (+0)
```

I tipi interi nei linguaggi di programmazione

Nella maggior parte dei linguaggi di programmazione sono definiti dei tipi di dato per la rappresentazione degli interi in diverse precisione.

Attraverso la definizione di un tipo di dato consente di creare variabili "tipizzate" cioè in grado di contenere solo informazioni codificate secondo le regole definite per il tipo.

Questa tecnica di programmazione consente di ridurre gli errori in fase di sviluppo.

La definizione di un tipo di dato in un linguaggio di programmazione porta alla definizione di una precisione della rappresentazione e di conseguenza al numero complessivo di codifiche disponibili per rappresentare i valori del tipo di dato e anche dei valori massimo e minimo.

Vengono inoltre definite le operazioni che si possono fare sulle variabili di un certo tipo.

I tipi interi in Java

Nell'ambiente Java tipi interi possono essere definiti in due modi: attraverso tipi "primitivi" definiti come parole chiave del linguaggio (*unboxed*) o attraverso oggetti istanziati da classi definite nel framework (*boxed*).

I due modi di definire gli interi portano ad alcune differenze nel loro utilizzo e quindi ci sono situazioni in cui è preferibile usare

variabile "unboxed" ed altre in cui conviene usare la versione "boxed".

Una importante differenza tra le due versione sta nel fatto che gli oggetti "boxed" vengono istanziati con un costruttore e quindi una variabile "boxed" è un "reference" ad un oggetto istanziato in memoria heap (la variabile contiene l'indirizzo dell'area di memoria che contiene il dato) mentre le variabili primitive (unboxed) vengono dichiarate direttamente nella memoria dei dati del programma (la variabile contiene effettivamente il valore).

Indipendentemente dalla versione unboxed o boxed sono disponibili diverse versioni di interi che si differenziano per la precisione e di conseguenza per il numero di valori rappresentabili e per i valori massimo e minimo.

Tipi interi primitivi

tipo	*prec.*	*MAX_VALUE*	*MIN_VALUE*
byte	8 bit	+127 ($+2^7-1$)	-128 (-2^7)
short	16 bit	+32.767 ($+2^{15}-1$)	-32.768 (-2^{15})
int	32 bit	+2.147.483.647 ($+2^{31}-1$)	-2.147.483.648 (-2^{31})
long	64 bit	+9.223.372.036.854.775.807 ($+2^{63}-1$)	-9.223.372.036.854.775.808 (-2^{63})

Classi per tipi interi

classe	*prec.*	*MAX_VALUE*	*MIN_VALUE*
Byte	8 bit	Come sopra	Come sopra
Short	16 bit	Come sopra	Come sopra
Integer	32 bit	Come sopra	Come sopra
Long	64 bit	Come sopra	Come sopra
BigInteger	arbitraria	Arbitraria	arbitraria

Verifica del comportamento degli interi

Attraverso una sequenza di codice Java inserita in un metodo main è possibile verificare che il comportamento degli interi in Java rispetta la codifica in complemento-a-due.

Dichiarazione di tipi primitivi:

```
byte    bVar1;
short   sVar2;
int     iVar3;
long    lVar4;
```

Essendo variabili di tipo primitivo le variabili non sono "reference" ma sono destinate a contenere effettivamente un valore del tipo definito e quindi vengono dimensionate in modo opportuno dal compilatore.
Poiché in questo esercizio vengono collocate come variabili locali all'interno di un metodo iniziano la loro esistenza solo quando vengono assegnate con un valore e terminano quando il metodo finisce.
Le altre due possibili situazioni sono:

- Attributi di una classe: le variabili primitive vengono istanziate al momento della costruzione di un oggetto e se non vengono inizializzate assumono il valore di default (0) e terminano quando l'oggetto viene dereferenziato
- Attributi **statici** di una classe: le variabili primitive vengono istanziate al momento dell'avvio del programma e se non vengono inizializzate assumono il valore di default (0) e terminano quando termina il programma.

Inizializzazione di variabili primitive

```
bVar1=1;
bVar1=-128;
bVar1=128;//errore di compilazione: eccede MAX_VALUE
sVar2=1;
sVar2=-32768;
sVar2=32768;//errore di compilazione: eccede MAX_VALUE
...
```

Dichiarazione di oggetti boxed:

```
Byte          bObj1;
Short         sObj2;
Integer       iObj3;
Long          lObj4;
BigInteger    biObj5;
```

Essendo variabili di tipo boxed le variabili sono "reference" ad oggetti istanziati nella memoria heap. Vengono inizializzati a 0 al

momento della costruzione se il costruttore non inizializza diversamente.

Istanziazione di oggetti boxed

```
iObj=new Integer(1);
iObj=new Integer(-1);
iObj1=new Integer(2147483648);//errore di compilazione:
                              //eccede MAX_VALUE
...
```

Visualizzazione di variabili intere

Possiamo visualizzare il contenuto di una variabile intera inviando sullo standard output una sua rappresentazione in caratteri numerici.
Il metodo println infatti è in grado di convertire le variabili primitive in stringhe di caratteri numerici da inviare sullo standard output.
Inoltre le classi "boxed" dispongono di molti metodi statici utili per la manipolazione di tipi interi:

```
int iV1;
int iV2;
iV1=+1;
iV2=-1;
System.out.println("iV1:"+iV1+" cod:"+Integer.toBinaryString(iV1));
System.out.println("iV2:"+iV2+" cod:"+Integer.toBinaryString(iV2));
```

Il metodo println converte il contenuto di idVx da intero in codifica binaria complemento-a-due in una stringa di testo e lo manda sullo standard output.
Il metodo statico toBinaryString della classe Integer riceve una variabile intera e restituisce una stringa di caratteri numerici che rappresenta il numero ricevuto in complemento-a-due e questa stringa viene inviata sullo standard output.
Nella stampa si vede sia il valore rappresentato sia la sua codifica:

```
iV1:1 cod:1
iV2:-1 cod:11111111111111111111111111111111
```

Nel caso del numero positivo la rappresentazione omette il segno e la codifica omette gli zeri non significativi.

Verifica dell'overflow

```
int iV1;
int iV2;
iV1=Integer.MAX_VALUE;
iV2=Integer.MIN_VALUE;
System.out.println("Prima dell'overflow");
System.out.println("iV1:"+iV1+" cod:"+Integer.toBinaryString(iV1));
System.out.println("iV2:"+iV2+" cod:"+Integer.toBinaryString(iV2));
iV1=iV1+1;
iV2=iV2-1;
System.out.println("Dopo l'overflow");
System.out.println("iV1:"+iV1+" cod:"+Integer.toBinaryString(iV1));
System.out.println("iV2:"+iV2+" cod:"+Integer.toBinaryString(iV2));
```

Le due variabili vengono inizializzate rispettivamente al valore massimo e minimo sfruttando una **costante statica pubblica** resa disponibile dalla classe Integer.
Alla variabile contenente il massimo viene aggiunto 1 mentre a quella contenente il minimo viene tolto 1. In questo modo in entrambi i casi si attraversa il limite della rappresentazione passando all'altro estremo della codifica.
Nella stampa si vede la rappresentazione sbagliata e la relativa codifica.

```
Prima dell'overflow
iV1:2147483647 cod:11111111111111111111111111111111
iV2:-2147483648 cod:10000000000000000000000000000000
Dopo l'overflow
iV1:-2147483648 cod:10000000000000000000000000000000
iV2:2147483647 cod:11111111111111111111111111111111
```

I

Operazioni sugli interi

Sugli interi primitivi sono definite le seguenti operazioni:

- Somma (operatore +): effettua la somma algebrica; può andare in overflow
- Sottrazione (operatore -): effettua la somma algebrica; può andare in overflow
- Moltiplicazione (operatore *): effettua la moltiplicazione algebrica quindi può determinare un cambio di segno con la regola del segno; può andare in oveflow

- Divisione intera (operatore /): effettua la divisione intera tra due numeri interi ottenendo il quoziente e scartando il resto; può cambiare il segno con la regola del cambio del segno; non può andare in overflow
- Resto della divisione (operatore %): effettua la divisione intera tra due numeri interi ottenendo il resto e scartando il quoziente; il resto è sempre positivo; non può andare in overflow
- Operatori relazionali (==,!=,<,>,<=,>=) : si applicano agli interi che hanno come relazione d'ordine:
  MIN_VALUE > negativi > 0 > positivi > MAX_VALUE

Conversione di tipo tra gli interi

Effettuando elaborazioni con i tipi di dato primitivi capita spesso di dover trasformare un tipo di dato in un altro tipo di dato.
Il valore dell'informazione non cambia (il valore +1 è sempre +1 sia nella precisione ad 8 bit [byte], 16 [short], 32 [int], 64 [long]) ma cambia la sua codifica (nel caso di un +1 si hanno 7 bit a zero seguiti da un 1 nel primo caso, 15 nel secondo, 31 nel terzo, 63 nel quarto caso)
Passando da un tipo ad un altro la codifica va quindi adattata.
Si possono verificare due situazioni:

- Promozione del tipo: si passa da un tipo di capacità inferiore ad un tipo di capacità superiore. Questa operazione è sempre possibile. Il compilatore non segnala alcun errore in fase di compilazione. Al tempo di esecuzione viene esteso il segno del complemento-a-due; vengono aggiunti un numero di zeri (per i numeri positivi) o di uni (per i numeri negativi) fino al riempimento della capacità del nuovo tipo.
  Esempi:
  ```
  byte bVar=1; //codice 00000001
  short sVar=bVar; //estensione  0000000000000001
  byte bVar=-1; //codice 11111111
  short sVar=bVar; //estensione  1111111111111111
  ```
- Forzatura del tipo (*casting*): si passa da un tipo di capacità superiore ad un tipo di capacità inferiore. Questa operazione non è sempre possibile perchè potrebbe provocare un overflow. In alcuni casi però è lecita e talvolta anche

indispensabile quindi viene lasciata allo sviluppatore la responsabilità di decidere se farla eseguire al runtime mediante una esplicita dichiarazione.
La compilazione produce un errore di compilazione che lo sviluppatore può tacitare "forzando" il tipo. Al tempo di esecuzione il valore viene troncato in modo da poter essere inserito nella variabile di capacità minore; se il valore è troppo grande si verifica un overflow.

Esempi:

```
short sVar=1;     //codice 0000000000000001
byte  bVar=sVar;  //errore di compilazione
byte bVar=(byte)sVar; //forzatura da short a byte
                      //troncatura  0000000000000001
                      //non produce overflow
short sVar=128;  //codice 0000000100000000
byte  bVar=sVar;  //errore di compilazione
byte bVar=(byte)sVar; //forzatura da short a byte
                      //troncatura  0000000100000000
                      //produce overflow
```