PROGETTO BIGINT

Gaia Assunta Bertolino
Mat. 209507 A.A. 2020/2021
Corso di Programmazione orientata agli oggetti

Organizzazione delle classi:

Il progetto è stato organizzato secondo quattro classi:
1) L'interfaccia BigInt che estende le classi Comparable<BigInt> e Iterable<Integer>
2) La classe astratta AbstractBigInt che implementa BigInt
3) La classe BigIntLL che estende la classe AbstractBigInt
4) Una classe Main

Interface BigInt

L'interfaccia BigInt ha la funzione di definire quei metodi comuni a tutte le implementazioni concrete del tipo BigInt. Dunque, conterrà al suo interno

1) METODI DI DEFAULT -> metodi che valgono per tutte le classi concrete che ereditano da essa. Essi sono:

- **String value() ->** metodo che costruisce una stringa formata dalle cifre del BigInt. Per fare ciò usa uno StringBuilder che permette di aggiungere una cifra alla volta tramite un iteratore anziché usare la semplice concatenazione di stringhe che, poiché non modificabili, sarebbe meno efficiente e comporterebbe la creazione di un nuovo oggetto ad ogni iterazione. Alla fine, converte lo *StringBuilder* in stringa
- **int length() ->** metodo che restituisce, sempre tramite un iteratore, il numero di cifre da cui è composto il numero
- **BigInt incr() ->** metodo che opera l'incremento del BigInt di una unità. Si appoggia al metodo *add(BigInt i)* con i=1 e richiama il metodo *factory(int i);* entrambi sono lasciati astratti e da implementare nelle classi eredi
- **BigInt decr() throws IllegalArgumentException ->** funziona come il metodo *incr()* ma si appoggia al metodo *sub(BigInt i)* con i=1, metodo lasciato astratto. Il metodo può lanciare una *IllegalArgumentException* quando this=0, in quanto il decremento genererebbe un numero negativo. L'eccezione viene gestita nella classe Main attraverso un blocco try-catch.
- **BigInt mul(BigInt m) ->** metodo che calcola il prodotto fra due BigInt attraverso il metodo *add(BigInt i)* lasciato da implementare nelle classi concrete. Il metodo opera attraverso una serie di somme successive (es. 5 * 6 = 5 + 5 + 5 + 5 + 5 + 5). Il ciclo di while si ferma nel momento in cui il moltiplicatore (che, facendo da indice, viene decrementato ad ogni ciclo) è pari a zero. Tale confronto viene operato tramite l'*equals(Object o)* (implementato nella classe astratta AbstractBigInt)
- **BigInt div (BigInt d) throws IllegalArgumentException ->** metodo che calcola la divisione fra due BigInt. Il metodo solleva una *IllegalArgumentException* nel momento in cui this è minore del BigInt passato come argomento o quando quest'ultimo è pari o minore a 0. Per operare la divisione, vengono utilizzati un indice *ris*, che sarà il risultato della divisione, e una variabile *tmp*. Ad ogni ciclo di while, viene sommato a *tmp* il divisore; se tale operazione mantiene la variabile *tmp* minore o uguale di this allora viene incrementato il valore di *ris* (es. 50/6 = 8 poiché 8 + 8 + 8 + 8 + 8 + 8, cioè sei volte, dà 48 che è minore e non maggiore di 50)

- **BigInt rem (BigInt d) throws IllegalArgumentException ->** metodo che calcola il resto della divisione fra due BigInt. Funziona come il metodo *div* (vedi sopra) senza però usare un contatore: dunque, viene calcolata la somma ripetuta del divisore nella variabile *ris* fino a quando questa è minore o uguale a this ma alla fine, per calcolare il resto, si opera la sottrazione fra this e *ris* (es. 50/6 = 8 poiché 8 + 8 + 8 + 8 + 8 + 8, cioè sei volte, dà 48 che è minore e non maggiore di 50 e dunque il resto sarà pari a 50-48=2).
- **BigInt pow(int exp) ->** metodo che calcola l'elevamento a potenza. Il metodo si appoggia al metodo *mul(BigInt n)* e usa la variabile *ris* inizializzata al BigInt=1, elemento neutro della moltiplicazione. Poiché possiamo assumere l'elevamento a potenza come una serie di moltiplicazioni fra un numero e sé stesso, ad ogni ciclo di while *ris* viene moltiplicata per this e si decrementa l'esponente *exp*, il quale assume la funzione di contatore per il ciclo (es. 5^3 = 5 * 5 * 5 cioè viene moltiplicato per sé stesso tre volte)

2) METODI ASTRATTI -> metodi che devono essere necessariamente implementati nella classe concreta. Essi sono:

- **BigInt factory(int i) throws IllegalArgumentException** -> metodo di supporto che serve a generare un BigInt a partire da un intero i
- **BigInt add(BigInt a)** -> metodo che opera l'addizione
- **BigInt sub(BigInt s) throws IllegalArgumentException** -> metodo che opera la sottrazione e solleva l'eccezione *IllegalArgumentException* nel momento in cui viene passato come argomento un numero minore di this (in questo caso, infatti, la sottrazione darebbe origine ad un numero negativo, non contemplato nella struttura di BigInt).

Abstract class AbstractBigInt implements BigInt

La classe AbstractBigInt estende l'interfaccia BigInt ma è astratta in quanto opera solamente l'override di tre metodi fondamentali quali

- **boolean equals(Object o)** -> metodo che verifica che due BigInt siano uguali o meno. Dopo i controlli di "routine", verifica se i due numeri hanno diversa lunghezza e in caso restituisce false; se però hanno lo stesso numero di cifre, opera il confronto cifra per cifra grazie a due iteratori (ciascuno generato su uno dei due BigInt). Il metodo *iterator()* da cui vengono generati va creato nella classe concreta.
- **String toString()** -> restituisce il BigInt sotto forma di stringa. Viene creato uno StringBuilder (più efficiente di un oggetto String nel caso di aggiunta di molti caratteri) e viene utilizzato un for-each nel ciclo di for per aggiungere una cifra alla volta. Ha lo stesso scopo e risultato del metodo *value()* dell'interfaccia BigInt ma, in quanto override del metodo *toString(),* ha una più ampia utilità in quanto è un metodo richiamato implicitamente da Java in varie situazioni. Dunque, avrebbe potuto anche richiamare direttamente il metodo *value()* ma si è preferito mostrare una ulteriore implementazione tramite l'iterazione implicita del for-each

- **int hashCode()** -> metodo che restituisce un valore hash basato su tutte le cifre del BigInt

class BigIntLL extends AbstractBigInt

La classe concreta BigIntLL fornisce una implementazione concreta dell'interfaccia BigInt estendendo la classe astratta AbstractBigInt e attraverso l'uso di una LinkedList, la quale permette una maggiore efficienza nell'aggiungere tanti nodi quanti sono le cifre di un numero a precisione illimitata. Dunque, conterrà al suo interno

1) METODI DI DEFAULT -> metodi che valgono per tutte le classi concrete che ereditano da essa. Essi sono:

- **BigIntLL()**-> costruttore di default che crea un BigIntLL inizializzando l'unica variabile d'istanza *lista* ad una nuova *LinkedList<>()*
- **BigIntLL(String s**) -> costruttore che crea un BigIntLL a partire da una stringa passata come argomento. Solleva una *NumberFormatException* se l'argomento passato presenta caratteri non numerici. Inoltre, alla fine viene richiamata una funzione *eliminazeri()* che elimina gli zeri nelle cifre più significative di this
- **BigIntLL(int i)** -> costruttore che crea un BigIntLL a partire da un intero passato come argomento. Solleva una *IllegalArgumentException* se l'argomento passato è un numero negativo
- **BigIntLL factory(int i)** -> metodo che fa l'override del rispettivo metodo astratto dell'interfaccia BigInt. Il metodo restituisce un BigIntLL a partire da un intero passato come argomento richiamando il costruttore *BigIntLL(int i*)
- **BigIntLL add(BigInt a)** -> metodo che fa l'override del rispettivo metodo astratto dell'interfaccia BigInt. Il metodo opera l'addizione cifra a cifra di due BigInt attraverso due relativi *listIterator* inizializzati alla fine di ciascun BigIntLL. Il cuore del metodo è il ciclo di while che continua fino a che l'indice *i* (inizializzato al numero di cifre del BigIntLL più grande) non si esaurisce; al suo interno viene valutato se bisogna sommare le cifre nella stessa posizione (col relativo riporto calcolato ad ogni addizione precedente) o se basta solo ricopiare le cifre del BigIntLL più grande nel caso in cui non abbiano lo stesso numero di cifre (sempre col relativo riporto che può propagarsi). Alla fine, viene valutato se è necessario sommare una ultima volta il riporto (ad es. 999 + 1 = 1000 deve propagare il riporto fino alla prima cifra a sinistra).
- **BigIntLL sub(BigInt s)** -> metodo che fa l'override del rispettivo metodo astratto dell'interfaccia BigInt. Il funzionamento è analogo a quello del metodo *add(BigInt a)* con la differenza che avviene una sottrazione fra le cifre corrispondenti e anche con il riporto. Inoltre, alla fine viene richiamata una funzione *eliminazeri()* che elimina gli zeri nelle cifre più significative del BigIntLL ottenuto e lo restituisce
- **BigIntLL eliminazeri()** -> metodo che elimina i nodi alle cifre più significative il cui valore è 0 per poi restituirlo
- **int compareTo(BigInt)** -> metodo che implementa il confronto fra due BigIntLL attraverso un confronto iniziale del numero di cifre dei due numeri; se dunque i due BigIntLL hanno lo stesso numero di cifre, si opera il confronto cifra per cifra attraverso due *iterator()*.
-

- **Iterator<Integer> iterator()** -> metodo che fa l'override del rispettivo metodo astratto dell'interfaccia BigInt. Restituisce un *iterator()* nella variabile d'istanza *lista* che raccoglie le cifre del BigIntLL
- **ListIterator<Integer> listIterator()** -> metodo che restituisce un *listIterator()* nella variabile d'istanza *lista* che raccoglie le cifre del BigIntLL
- **ListIterator<Integer> listIterator(int i)** -> metodo che restituisce un *listIterator()* nella variabile d'istanza *lista* che raccoglie le cifre del BigIntLL in posizione i
- **BigIntLL mul (int i)** -> metodo che calcola la moltiplicazione fra this e l'intero *i* passato come argomento. E' un metodo di supporto per operare una moltiplicazione più efficiente fra due BigIntLL nel metodo *mul(BigInt m)*
- **BigIntLL mul (BigInt m)** -> metodo che fa l'override del rispettivo metodo astratto dell'interfaccia BigInt per una migliore efficienza. Il metodo calcola la moltiplicazione fra due BigInt. Esso consiste nell'implementare il meccanismo da "carta e penna" attraverso una serie di somme (dove ciascun numero ha dunque i rispettivi zeri nelle cifre meno significative) fra la moltiplicazione di this e una cifra per volta (da sinistra verso destra) del BigInt passato come argomento.

Class Main

Classe che presenta dei metodi di supporto ad un *main*, il quale è il fulcro del programma in quanto implementa la possibilità di interagire con il programma stesso in maniera intuitiva attraverso uno scanner che legge i comandi dell'utente da tastiera. I metodi contenuti dunque sono:

- **static BigIntLL nuovo (Scanner sc)** -> metodo che, tramite uno scanner passato come argomento, legge cosa viene digitato da tastiera fin quando non è possibile generare un BigIntLL; gestisce la lettura da tastiera catturando la NumberFormatException generata nel caso in cui si cerchi di creare un BigIntLL a partire da una stringa che è solo numerica. Invece, nel caso in cui si sia digitata una stringa corretta, si esce dal ciclo e il metodo restituisce il BigIntLL creato. Inoltre, il metodo fa terminare il programma nel caso in cui venga digitato in input *STOP.*
- **static void scelte(Scanner sc, BigIntLL numero) ->** metodo che consente di eseguire delle operazioni sul BigIntLL ricevuto come argomento attraverso la scelta da un elenco. La scelta dell'operazione da eseguire è gestita con una lettura da tastiera e uno switch che incanala la scelta verso la rispettiva sequenza di istruzioni (es. digitando *A* viene richiesto un secondo BigIntLL da tastiera (gestito sempre attraverso l'invocazione della funzione *metodo(Scanner sc)* e vengono eseguite le istruzioni dell'addizione*).* Quando viene terminata un'operazione, viene chiesto all'utente se ha intenzione di continuare con le operazioni, creare un nuovo BigIntLL o terminare il programma
- **static void main(String[] args) ->** metodo che a sua volta invoca il metodo *nuovo(Scanner sc)* e il metodo *scelte(Scanner sc, BigIntLL numero)* e permette l'esecuzione del programma interattivo