

settembre 2025

4. Considerare le seguenti classi:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class FactIterable implements Iterable<Double> {
    private final byte items; // numero di elementi della sequenza

    // inizializza la sequenza con items elementi 1!, 2!, ..., items! se items > 0
    // la sequenza non ha elementi se items <= 0
    public FactIterable(byte items) { this.items = items; }

    // inizializza la sequenza con Byte.MAX_VALUE elementi
    public FactIterable() { this(Byte.MAX_VALUE); }

    // restituisce un oggetto che itera sulla sequenza
    @Override public Iterator<Double> iterator() { /* completare */ }

    public class FactIterator implements Iterator<Double> {
        /* completare con le dichiarazioni dei campi */

        public FactIterator(byte maxItems) { /* completare */ }
        public boolean hasNext() { /* completare */ }
        public Double next() { /* completare */ }
    }
}
```

Un oggetto **new** FactIterable(*n*) è immutabile e rappresenta la sequenza crescente dei numeri fattoriali 1!, 2!, ..., *n*!. Se *n* ≤ 0, la sequenza è vuota.

Un oggetto **new** FactIterator(*n*) consente di iterare sulla sequenza crescente dei numeri fattoriali 1!, 2!, ..., *n*!. Se *n* ≤ 0, l'iteratore è vuoto.

Il metodo hasNext () restituisce **true** se e solo se esiste il prossimo elemento dell'iterazione e non modifica lo stato dell'iteratore. Il metodo next () restituisce tale elemento, se esiste, e fa avanzare l'iteratore all'elemento successivo; altrimenti, lancia l'eccezione NoSuchElementException.

Esempio:

```
public class Test {
    public static void main(String[] args) {
        var it = new FactIterable((byte) 5).iterator();
        while(it.hasNext())
            System.out.print(it.next() + " ");
        for (var i : new FactIterable((byte) -1))
            System.out.print(i + " ");
    }
}
```

- (a) completare il metodo iterator() della classe FactIterator.
- (b) dichiarare i campi della classe FactIterator e completarne il costruttore.
- (c) completare il metodo hasNext () della classe FactIterator.
- (d) completare il metodo next () della classe FactIterator. Il metodo deve avere una complessità temporale costante rispetto al numero di elementi restituiti dall'iteratore.

luglio 2025

4. Considerare le seguenti interfacce e classi che modellano sequenze immutabili di interi in ordine decrescente e i relativi iteratori di tipo `java.util.Iterator<Integer>`:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public interface ReverseRangeInterface extends Iterable<Integer> {
    Iterator<Integer> iterator();
}

public class ReverseRange implements ReverseRangeInterface {
    private final int start, end;
    // inizializza una sequenza da start (incluso) a end (escluso), in ordine decrescente
    public ReverseRange(int start, int end) { this.start = start; this.end = end; }
    public Iterator<Integer> iterator() { /* completare */ }

    public class ReverseRangeIterator implements Iterator<Integer> {
        /* completare con le dichiarazioni dei campi */

        public ReverseRangeIterator(int start, int end) { /* completare */ }
        public boolean hasNext() { /* completare */ }
        public Integer next() { /* completare */ }
    }
}
```

Un oggetto **new** ReverseRange(*i*, *j*) è immutabile e rappresenta la sequenza decrescente di numeri interi contigui da *i* (incluso) a *j* (escluso). Se *i* ≤ *j*, la sequenza è vuota.

Un oggetto della classe ReverseRangeIterator consente di iterare sugli elementi di un oggetto di tipo ReverseRange.

Il metodo hasNext () restituisce **true** se e solo se esiste il prossimo elemento dell'iterazione. Il metodo next () restituisce tale elemento, se c'è; altrimenti, lancia l'eccezione NoSuchElementException.

- (a) completare il metodo iterator() della classe ReverseRange.
- (b) dichiarare i campi della classe ReverseRangeIterator e completarne il costruttore.
- (c) completare il metodo hasNext () della classe ReverseRangeIterator.
- (d) completare il metodo next () della classe ReverseRangeIterator.
- (e) completare il metodo statico sumSquaresOf(ReverseRange r) che calcola la somma dei quadrati degli elementi della sequenza *r*. La somma degli elementi di una sequenza vuota è 0.

Esempio:

```
public class ReverseRangeUtil {
    public static int sumSquaresOf(ReverseRange r) { /* completare */ }

    public static void main(String[] args) {
        assert sumSquaresOf(new ReverseRange(1,1))==0; // sequenza vuota
        assert sumSquaresOf(new ReverseRange(5,2))==50; // 5²+4²+3²=50
        assert sumSquaresOf(new ReverseRange(-1,-4))==14; // (-1)²+(-2)²+(-3)²=14
        assert sumSquaresOf(new ReverseRange(3,7))==0; // sequenza vuota
    }
}
```

```
public class FactIterable implements Iterable<Double> {
    private final byte items;
    public FactIterable(byte items) {
        this.items = items;
    }
    public FactIterable() {
        this(Byte.MAX_VALUE);
    }
    @Override
    public Iterator<Double> iterator() {
        return new FactIterator(items);
    }
}
```

```
public class FactIterator implements Iterator<Double> {
    private byte items;
    private double current = 1;
    private final byte maxItems;
    public FactIterator(byte maxItems) {
        this.maxItems = maxItems;
    }
    @Override
    public boolean hasNext() {
        return items < maxItems;
    }
    @Override
    public Double next() {
        if (!hasNext())
            throw new NoSuchElementException();
        items++;
        current *= items;
        return current;
    }
}
```

```
public class ReverseRange implements ReverseRangeInterface {
    private final int start, end;
    public ReverseRange(int start, int end) {
        this.start = start;
        this.end = end;
    }
}
```

```
@Override
public Iterator<Integer> iterator() {
    return new ReverseRangeIterator(start, end);
}
```

```
public class ReverseRangeIterator implements Iterator<Integer> {
    private int next;
    private final int end;
    public ReverseRangeIterator(int start, int end) {
        this.start = start;
        this.end = end;
    }
}
```

```
@Override
public boolean hasNext() {
    return next > end;
}
@Override
public Integer next() {
    if (!hasNext())
        throw new NoSuchElementException();
    return next--;
}
```

```
public static int sumSquaresOf(ReverseRange r) {
    int sum = 0;
    for (int x : r) {
        sum += x * x;
    }
    return sum;
}
```

giugno 2025

4. Considerare le seguenti interfacce, corrispondenti alle sequenze di interi e agli iteratori nello stile di C#, e le classi che le implementano:

```
public interface RangeInterface { IteratorInterface iterator(); }
public interface IteratorInterface { boolean moveNext(); int current(); }
public class Range implements RangeInterface {
    private final int start, end;
    // inizializza una sequenza da start (incluso) a end (escluso)
    public Range(int start, int end) { this.start = start; this.end = end; }
    public IteratorInterface iterator() { /* completare */ }
}
public class RangeIterator implements IteratorInterface {
// dichiarare i campi necessari
    public RangeIterator(int next, int end) { /* completare */ }
    public boolean moveNext() { /* completare */ }
    public int current() { /* completare */ }
}
```

Un oggetto `new Range(i, j)` è immutabile e rappresenta la sequenza crescente di numeri interi contigui che inizia con `i` e termina con `j` escluso. Se `i ≥ j`, allora la sequenza è vuota.

Un oggetto della classe `RangeIterator` permette di iterare sugli elementi di un oggetto di tipo `Range`.

Il metodo `moveNext()` permette all'iteratore di spostarsi al prossimo elemento dell'iterazione. Il metodo non lancia mai eccezioni e restituisce `true` se e solo se tale elemento esiste.

Il metodo `current()` restituisce l'elemento corrente dell'iteratore, se è definito, ossia, se `moveNext()` è stato chiamato almeno una volta e se l'ultima chiamata di `moveNext()` ha restituito `true`; altrimenti lancia l'eccezione `NoSuchElementException`. Il metodo **non modifica** lo stato dell'iteratore; quindi, finché non viene chiamato `moveNext()`, chiamate consecutive di `current()` restituiscono sempre lo stesso risultato o lanciano tutte l'eccezione `NoSuchElementException`.

- (a) completare il metodo `iterator()` della classe `Range`.
- (b) dichiarare i campi della classe `RangeIterator` e il suo costruttore.
- (c) completare il metodo `moveNext()` della classe `RangeIterator`.
- (d) completare il metodo `current()` della classe `RangeIterator`.
- (e) completare il metodo statico `prodOf(Range r)` che calcola il prodotto degli elementi di `r`.

Esempio:

```
public class RangeUtil {
    public static int prodOf(Range r) { /* completare */ }
    public static void main(String[] args) {
        assert RangeUtil.prodOf(new Range(1,1))==1; // prodotto di sequenza vuota
        assert RangeUtil.prodOf(new Range(-1,2))==0; // -1*0*1==0
        assert RangeUtil.prodOf(new Range(1,6))==120; // 5!=1*2*3*4*5==120
    }
}
```

```
public static int prodOf(Range r) {
    var res = 1;
    var it = r.iterator();
    while (it.moveNext())
        res *= it.current();
    return res;
}
```

```
public interface RangeInterface {
    IteratorInterface iterator();
}
public interface IteratorInterface {
    boolean moveNext(); int current();
}
public class Range implements RangeInterface {
    private final int start, end;
    public Range(int start, int end) {
        this.start = start;
        this.end = end;
    }
    @Override
    public IteratorInterface iterator() {
        return new RangeIterator(start, end);
    }
}
```

```
public class RangeIterator implements IteratorInterface {
    private int next;
    private final int end;
    private boolean hasCurrent;
    public RangeIterator(int next, int end) {
        this.next = next;
        this.end = end;
    }
    @Override
    public boolean moveNext() {
        hasCurrent = next < end;
        if (hasCurrent)
            next++;
        return hasCurrent;
    }
    @Override
    public int current() {
        if (!hasCurrent)
            throw new NoSuchElementException();
        return next - 1;
    }
}
```

settembre 2024

4. Considerare la seguente interfaccia

```
public interface MyIterator<E> { boolean advance(); E element(); }
```

che definisce iteratori di tipo `MyIterator<E>` con interfaccia diversa da quella predefinita `Iterator<E>`.

Il metodo `advance()` permette di modificare lo stato dell'iteratore avanzando al prossimo elemento dell'iterazione. Se tale elemento è presente, allora il metodo restituisce `true`, altrimenti `false`. In ogni caso `advance()` non solleva mai eccezioni.

Il metodo di query `element()` restituisce l'elemento corrente, ossia quello dove è arrivata l'iterazione. Se tale elemento non esiste, perché `advance()` non è stato chiamato una prima volta o ha restituito `false`, allora `element()` solleva l'eccezione `java.util.NoSuchElementException`.

Dato che `element()` è un metodo di query, esso non modifica lo stato dell'iteratore, quindi due sue chiamate consecutive si devono comportare esattamente allo stesso modo se tra di esse non è stato chiamato il metodo `advance()`.

La classe `GetMyIterator<E>` permette di creare un iteratore `i1` di tipo `MyIterator` a partire da un iteratore `i0` di tipo `java.util.Iterator<E>`. L'iteratore `i1` usa `i0` per compiere l'iterazione.

Esempio:

```
import static java.util.Arrays.asList;
/*
stampa
a
a
null
null
b
b
*/
public class MyIteratorTest {
    public static void main(String[] args) {
        var list = asList("a", null, "b"); // crea la lista ["a",null,"b"]
        var it = list.iterator(); // ottiene un Iterator di list
        var myIt = new GetMyIterator<E>(it); // ottiene un MyIterator da it
        while (myIt.advance()) {
            System.out.println(myIt.element());
            System.out.println(myIt.element());
        }
    }
}
public class GetMyIterator<E> implements MyIterator<E> { /* completare */ }
```

- (a) definire in `GetMyIterator<E>` i campi strettamente necessari per una corretta implementazione della classe, insieme ai loro possibili invarianti.
- (b) definire in `GetMyIterator<E>` il costruttore necessario per una corretta implementazione della classe.
- (c) definire in `GetMyIterator<E>` il metodo `advance()`.
- (d) definire in `GetMyIterator<E>` il metodo `element()`.

```
public class GetMyIterator<E> implements MyIterator<E> {
    private final Iterator<E> it;
    private E elem;
    private boolean hasElem;
}
```

```
public GetMyIterator(Iterator<E> it) {
    this.it = requireNonNull(it);
}
```

```
@Override
public boolean advance() {
    hasElem = it.hasNext();
    if (hasElem)
        elem = it.next();
    return hasElem;
}
```

```
@Override
public E element() {
    if (!hasElem)
        throw new NoSuchElementException();
    return elem;
}
```

luglio 2024

4. Considerare la seguente interfaccia

```
public interface SmartIterator<E> { boolean moveNext(); E current(); }
```

che definisce iteratori "smart".

Il metodo `moveNext()` permette di avanzare al prossimo elemento dell'iterazione. Se tale elemento è presente, allora il metodo restituisce `true`, altrimenti `false`. In ogni caso `moveNext()` non solleva mai eccezioni.

Il metodo `current()` restituisce l'elemento corrente, ossia quello dove è arrivata l'iterazione. Se tale elemento non esiste, perché `moveNext()` non è stato chiamato una prima volta o ha restituito `false`, allora `current()` solleva l'eccezione `java.util.NoSuchElementException`. Due chiamate di `current()` si devono comportare esattamente allo stesso modo se tra le due non è stato chiamato il metodo `moveNext()`.

La classe `GetSmartIterator<E>` permette di definire iteratori "smart" su array di tipo `E[]`. L'iterazione segue l'ordine convenzionale degli elementi degli array. Esempio:

```
public class IteratorTest {
/*
stampa
a
a
null
null
b
b
*/
public static void main(String[] args) {
    var arr = new String[] { "a", null, "b" }; // crea l'array ["a",null,"b"]
    var smartIt = new GetSmartIterator<String>(arr); // ottiene un iteratore smart per arr
    while (smartIt.moveNext()) {
        System.out.println(smartIt.current());
        System.out.println(smartIt.current());
    }
}
public class GetSmartIterator<E> implements SmartIterator<E> { /* completare */ }
```

- (a) definire in `GetSmartIterator<E>` il/i campo/i strettamente necessario/i per una corretta implementazione della classe, insieme ai loro possibili invarianti.
- (b) definire in `GetSmartIterator<E>` il/i costruttore/i strettamente necessario/i per una corretta implementazione della classe.
- (c) definire in `GetSmartIterator<E>` il metodo `moveNext()` necessario per una corretta implementazione della classe.
- (d) definire in `GetSmartIterator<E>` il metodo `current()` necessario per una corretta implementazione della classe.

```
public class GetSmartIterator<E> implements SmartIterator<E> {
    private final E[] arr;
    private int index = -1;
```

```
public GetSmartIterator(E[] arr) {
    this.arr = requireNonNull(arr);
}
```

```
@Override
public boolean moveNext() {
    if (index < arr.length)
        index++;
    return index < arr.length;
}
```

```
@Override
public E current() {
    if (index < 0 || index >= arr.length)
        throw new NoSuchElementException();
    return arr[index];
}
```

giugno 2024

4. Considerare le seguenti interfacce corrispondenti all'iterator pattern implementato nello stile di JavaScript:

```
public interface JSIterator<E> { JSIteratorResult<E> next(); }
public interface JSIteratorResult<E> { E value(); boolean done(); }
```

Un iteratore ha solo il metodo `next()` che restituisce come risultato un oggetto immutabile con due attributi il cui valore può essere letto con i metodi `value()` e `done()`.

Se `done()` restituisce `true`, allora l'iterazione è terminata e `value()` restituisce `null`. In questo caso ogni ulteriore chiamata del metodo `next()` restituisce un risultato analogo, senza sollevare eccezioni.

Se `done()` restituisce `false`, allora l'iterazione non è terminata e `value()` restituisce il suo corrispondente elemento.

La classe `ToJSIterator<E>` permette di convertire un iteratore implementato nello stile di Java in uno corrispondente nello stile di JavaScript. Esempio:

```
import static java.util.Arrays.asList;
public class IteratorTest {
    public static void main(String[] args) {
        var l = asList("a", "b", "c"); // crea la lista ["a","b","c"]
        var javIt = l.iterator(); // iteratore per la lista l in stile Java
        var jsIt = new ToJSIterator<String>(javIt); // converte in iteratore in stile JavaScript
        var res = jsIt.next();
        while (!res.done()) {
            System.out.println(res.value()); // stampa le tre linee a b c
            res = jsIt.next();
        }
    }
}
public record JSIteratorResultRecord<E> // da completare, usato da ToJSIterator<E>
{
    public class ToJSIterator<E> implements JSIterator<E> { // da completare
    }
```

- (a) completare `JSIteratorResultRecord<E>` che implementa `JSIteratorResult<E>` e che viene usato dalla classe `ToJSIterator<E>`.
- (b) definire in `ToJSIterator<E>` il/i campo/i strettamente necessario/i per una corretta implementazione della classe, insieme ai loro possibili invarianti.
- (c) definire in `ToJSIterator<E>` il/i costruttore/i strettamente necessario/i per una corretta implementazione della classe.
- (d) definire in `ToJSIterator<E>` il metodo `next()` necessario per una corretta implementazione della classe.

```
public record JSIteratorResultRecord<E>(E value, boolean done) implements JSIteratorResult<E> { }
```

```
public class ToJSIterator<E> implements JSIterator<E> {
    private final Iterator<E> it;
```

```
public ToJSIterator(Iterator<E> it) {
    this.it = requireNonNull(it);
}
```

```
@Override
public JSIteratorResult<E> next() {
    var done = !it.hasNext();
    var value = done ? null : it.next();
    return new JSIteratorResultRecord<E>(value, done);
}
```

febbraio 2024

4. Considerare la classe `Pair<E1, E2>` che implementata copie di valori di tipo generico rispettivamente `E1` e `E2`.

```
public class Pair<E1, E2> {
    private final E1 first;
    private final E2 second;
    public Pair(E1 first, E2 second) { this.first = first; this.second = second; }
    public String toString() { /* completare */ } ✘
    ...
}
```

La classe incompleta `IndexIterator`, definita sotto, permette di creare iteratori a partire da un altro iteratore `iterator`.

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import static java.util.Objects.requireNonNull;
public class IndexIterator<E> implements Iterator<Pair<Integer, E>> {
    private final Iterator<E> iterator; // invariant: iterator!=null
    private int index;
    public IndexIterator(Iterator<E> iterator) { /* completare */ }
    public boolean hasNext() { /* completare */ }
    public Pair<Integer, E> next() { /* completare */ }
}
```

Un oggetto della classe `IndexIterator` restituisce nello stesso ordine gli elementi di `iterator` accoppiati, tramite la classe `Pair<E1, E2>`, con il corrispondente indice, a partire da 0.

Esempio:

```
import static java.util.Arrays.asList;
public class IteratorTest {
    public static void main(String[] args) {
        var l = asList("a", "b", "c"); // crea la lista ["a", "b", "c"]
        var it = new IndexIterator<>(l.iterator());
        while (it.hasNext())
            System.out.println(it.next()); // stampa le tre linee (0,a) (1,b) (2,c)
    }
}
```

- (a) completare il metodo `toString()` della classe `Pair<E1, E2>`.
- (b) completare il costruttore della classe `IndexIterator<E>`.
- (c) completare il metodo `hasNext()` della classe `IndexIterator<E>`.
- (d) completare il metodo `next()` della classe `IndexIterator<E>`.

luglio 2023

4. Considerare la classe `StringArrayRevIterator` per creare oggetti che permettono l'iterazione in **ordine inverso** su array non nulli.

Esempio:

```
var it = new StringArrayRevIterator(new String[] { "a", "b", "c" });
while (it.hasNext())
    System.out.println(it.next()); // stampa le tre linee c b a
it = new StringArrayRevIterator("one", "two", "three");
while (it.hasNext())
    System.out.println(it.next()); // stampa le tre linee three two one
it = new StringArrayRevIterator(new String[0]);
while (it.hasNext())
    System.out.println(it.next()); // non stampa nulla
```

Codice della classe `StringArrayIterator`:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import static java.util.Objects.requireNonNull;

public class StringArrayRevIterator implements Iterator<String> {
    // ...
}
```

- (a) definire i campi e i relativi costruttori, anche uno solo se reputato sufficiente, strettamente necessari alla classe;
- (b) implementare il metodo `hasNext()` dell'interfaccia `Iterator`;
- (c) implementare il metodo `next()` dell'interfaccia `Iterator`.

giugno 2023

4. Completare la classe `StringArrayIterator` per creare oggetti che permettono di iterare su elementi di array non nulli.

Esempio:

```
var it = new StringArrayIterator(new String[] { "a", "b", "c" });
while (it.hasNext())
    System.out.println(it.next()); // stampa a b c
it = new StringArrayIterator(new String[0]);
while (it.hasNext())
    System.out.println(it.next()); // non stampa nulla
```

Codice della classe `StringArrayIterator`:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import static java.util.Objects.requireNonNull;

public class StringArrayIterator implements Iterator<String> {
    // completare
}
```

- (a) definire i campi (anche uno solo se reputato sufficiente) e i relativi costruttori strettamente necessari alla classe;
- (b) implementare il metodo `hasNext()` dell'interfaccia `Iterator`;
- (c) implementare il metodo `next()` dell'interfaccia `Iterator`.

```
public class IndexIterator<E> implements Iterator<Pair<Integer, E>> {
```

```
    private final Iterator<E> iterator; // invariant: iterator!=null
    private int index;
```

```
    public IndexIterator(Iterator<E> iterator) {
        this.iterator = requireNonNull(iterator);
    }
```

```
    @Override
    public boolean hasNext() {
        return iterator.hasNext();
    }
```

```
    @Override
    public Pair<Integer, E> next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return new Pair<>(index++, iterator.next());
    }
}
```

```
    ✘ @Override
    public String toString() {
        return "(" + first + "," + second + ")";
    }
```

```
public class StringArrayRevIterator implements Iterator<String> {
    public int index;
    public final String[] arr;
```

```
    public StringArrayRevIterator(String... arr) {
        this.arr = requireNonNull(arr);
        index = arr.length - 1;
    }
```

```
    @Override
    public boolean hasNext() {
        return index >= 0;
    }
```

```
    @Override
    public String next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return arr[index--];
    }
}
```

```
public class StringArrayIterator implements Iterator<String> {
    private int index;
    private final String[] arr;
```

```
    public StringArrayIterator(String... arr) {
        this.arr = requireNonNull(arr);
    }
```

```
    @Override
    public boolean hasNext() {
        return index < arr.length;
    }
```

```
    @Override
    public String next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return arr[index++];
    }
}
```

febbraio 2023

4. Considerare la classe `AltIterator<E>` che fonde assieme due iteratori `it1` e `it2` di tipo `Iterator<E>` in un nuovo iteratore `altIt` dello stesso tipo che genera i loro elementi alternandoli a partire da `it1`. Quando uno dei due iteratori non ha più elementi `altIt` genera i rimanenti elementi dell'altro.

Esempio:

```
var it1 = ... // iteratore che restituisce 1, 2, 3
var it2 = ... // iteratore che restituisce 4, 5
var it = new AltIterator<E>(it1, it2); // nuovo iteratore che alterna it1 e it2
while(it.hasNext())
    System.out.println(it.next()); // stampa 1, 4, 2, 5, 3
it1 = ... // iteratore vuoto
it2 = ... // iteratore che restituisce 4, 5
it = new AltIterator<E>(it1, it2); // nuovo iteratore che alterna it1 e it2
while(it.hasNext())
    System.out.println(it.next()); // stampa 4, 5
it1 = ... // iteratore che restituisce 1, 2, 3
it2 = ... // iteratore vuoto
it = new AltIterator<E>(it1, it2); // nuovo iteratore che alterna it1 e it2
while(it.hasNext())
    System.out.println(it.next()); // stampa 1, 2, 3
```

Codice della classe `CatIterator`:

```
import java.util.Iterator;
import static java.util.Objects.requireNonNull;

public class AltIterator<E> implements Iterator<E> {

    private Iterator<E> first, second; // invariant: first != null and second != null
    public AltIterator(Iterator<E> first, Iterator<E> second) { /* completare */ }

    public boolean hasNext() { /* completare */ }

    public E next() { /* completare */ }

}
```

- (a) completare il costruttore della classe `AltIterator`.
(b) completare il metodo `hasNext()` della classe `AltIterator`.
(c) completare il metodo `next()` della classe `AltIterator`.

```
private class AltIterator<E> implements Iterator<E> {
    private Iterator<E> first, second;
```

```
private AltIterator(Iterator<E> first, Iterator<E> second) {
    this.first = requireNonNull(first);
    this.second = requireNonNull(second);
}
```

@Override

```
public boolean hasNext() {
    return first.hasNext() || second.hasNext();
}
```

@Override

```
public E next() {
    if (first.hasNext()) {
        var tmp = first;
        first = second;
        second = temp;
    }
    return second.next();
}
```

maggio 2022

4. Considerare la classe `FibIterator` per generare la successione dei numeri di Fibonacci a_0, a_1, \dots, a_n minori o uguali di un certo numero `max`, dove $a_0 = 0, a_1 = 1, a_i = a_{i-1} + a_{i-2}$ per ogni $i = 2 \dots n$.

Esempio:

```
var it = new FibIterator(6);
while (it.hasNext())
    System.out.println(it.next()); // stampa 0 1 1 2 3 5
it.reset(2);
while (it.hasNext())
    System.out.println(it.next()); // stampa 0 1 1 2
it.reset(-1);
assert !it.hasNext();
```

Codice della classe `FibIterator`:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class FibIterator implements Iterator<Integer> {

    private int next = 0; // prossimo elemento da restituire
    private int next_next = 1; // elemento da restituire dopo next
    private int max; // tutti gli elementi della successione sono <= max

    public FibIterator(int max) { /* completare */ }

    public boolean hasNext() { /* completare */ }

    public Integer next() { /* completare */ }

    public void reset(int max) { /* completare */ }
}
```

- (a) completare il costruttore `FibIterator(int max)` che inizializza l'iteratore.
(b) completare i metodi `hasNext()` e `next()` che implementano l'interfaccia `java.util.Iterator`; entrambi i metodi devono avere complessità temporale costante nel caso peggiore.
(c) completare il metodo `reset(int max)` che reimposta i campi dell'iteratore.

```
public class FibIterator implements Iterator<Integer> {
    private int next = 0;
    private int next_next = 1;
    private int max;
```

```
public FibIterator(int max) {
    this.max = max;
}
```

@Override

```
public boolean hasNext() {
    return next <= max;
}
```

@Override

```
public Integer next() {
    if (!hasNext())
        throw new NoSuchElementException();
    var res = next;
    next = next_next;
    next_next += res;
    return res;
}
```

```
public void reset(int max) {
    next = 0;
    next_next = 1;
    this.max = max;
}
```

giugno 2022

4. Considerare la classe PowIterator per generare sequenze di potenze di numeri interi nel seguente modo: se $b \neq 0$, allora **new PowIterator(*b*, *s*)** genera la sequenza di *s* elementi b^0, b^1, \dots, b^{s-1} ; se $b = 0$ viene sollevata un'eccezione; se $s \leq 0$ non vengono generati elementi.

Importante: per motivi di efficienza, il metodo `next()` deve eseguire una sola moltiplicazione per ogni chiamata.

Esempio:

```
var it = new PowIterator(2, 3); // iteratore per le prime 3 potenze di 2: 20, 21, 22
while (it.hasNext())
    System.out.println(it.next()); // stampa 1 2 4
it.reset(-1, 4); // stesso iteratore, ma per le prime 4 potenze di -1: -10, -11, -12, -13
while (it.hasNext()) // stampa 1 -1 1 -1
    System.out.println(it.next());
```

Codice della classe PowIterator:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class PowIterator implements Iterator<Integer> {

    private int base; // invariant: base != 0
    private int next = 1; // prossimo elemento da restituire
    private int size; // numero elementi da restituire, nessun elemento se size<=0

    protected static int checkBase(int base) { /* completare */ }

    public PowIterator(int base, int size) { /* completare */ }

    public boolean hasNext() { /* completare */ }

    public Integer next() { /* completare */ }

    public void reset(int base, int size) { /* completare */ }
}
```

- (a) completare i metodi `hasNext()` e `next()` che implementano l'interfaccia `java.util.Iterator`; il metodo `next()` deve eseguire una sola moltiplicazione per ogni chiamata.
(b) completare il metodo `checkBase(int base)` che solleva un'eccezione di tipo appropriato se base è uguale a 0, restituiscce base altrimenti.
(c) completare il costruttore `PowIterator(int base, int size)` che inizializza i campi `base` e `size`, e il metodo `reset(int base, int size)` che reimposta i campi `base` e `size` dell'iteratore; per entrambi i casi deve essere garantito l'invariante per `base`.

giugno 2021

3. Completare la seguente classe di iteratori `CharSeqIterator` per iterare, nell'ordine convenzionale dal primo all'ultimo elemento, sui caratteri di una sequenza di tipo `java.lang.CharSequence`.

Esempio:

```
for (var ch : new CharSeqIterator("abc")) System.out.println(ch); // prints a b c
for (var ch : new CharSeqIterator("")) System.out.println(ch); // no printed chars
for (var ch : new CharSeqIterator("aBc")) System.out.println(ch); // prints a B c
```

L'interfaccia predefinita `java.lang(CharSequence` (implementata da `java.lang.String`) contiene, tra gli altri, i metodi `charAt(int index)` e `int length()`: il primo restituisce il carattere della sequenza che si trova all'indice `index` (gli indici partono da zero), il secondo calcola la lunghezza della sequenza.

Codice da completare:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import static java.util.Objects.requireNonNull;

class CharSeqIterator implements Iterator<Character>, Iterable<Character> {

    // dichiarare i campi mancanti
    // invariant charSeq!=null

    public CharSeqIterator(CharSequence charSeq) {
        // completare
    }

    @Override
    public boolean hasNext() {
        // completare
    }

    @Override
    public Character next() {
        // completare
    }

    @Override
    public Iterator<Character> iterator() { return this; }
}
```

```
public class PowIterator implements Iterator<Integer> {
    private int base;
    private int next = 1;
    private int size;

    protected static int checkBase(int base) {
        if (base == 0)
            throw new IllegalArgumentException("Base cannot be zero");
        return base;
    }

    public PowIterator(int base, int size) {
        this.base = checkBase(base);
        this.size = size;
    }

    @Override
    public boolean hasNext() {
        return size > 0;
    }

    @Override
    public Integer next() {
        if (!hasNext())
            throw new NoSuchElementException();
        var res = next;
        next *= base;
        size--;
        return res;
    }

    public void reset(int base, int size) {
        this.base = checkBase(base);
        this.next = 1;
        this.size = size;
    }
}

class CharSeqIterator implements Iterator<Character>, Iterable<Character> {
    private final CharSequence charSeq; // invariant charSeq != null
    private final int length;
    private int index;

    public CharSeqIterator(CharSequence charSeq) {
        this.charSeq = requireNonNull(charSeq);
        length = charSeq.length();
    }

    @Override
    public boolean hasNext() {
        return index < length;
    }

    @Override
    public Character next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return charSeq.charAt(index++);
    }

    @Override
    public Iterator<Character> iterator() {
        return this;
    }
}
```

giugno 2021

3. Completare la seguente classe di iteratori RangeIterator per generare sequenze di interi da un estremo (`start`) incluso fino a un estremo (`end`) escluso con passo (`step`) diverso da zero.

Esempio:

```
for (var i : new RangeIterator(3)) // start=0 end=3 step=1
    System.out.println(i); // prints 0 1 2
for (var i : new RangeIterator(3, -1)) // start=3 end=-1 step=1
    System.out.println(i); // no printed values
for (var i : new RangeIterator(3, -1, -3)) // start=3 end=-1 step=-3
    System.out.println(i); // prints 3 0
```

Codice da completare:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

class RangeIterator implements Iterator<Integer>, Iterable<Integer> {

    // dichiarare i campi mancanti
    // invariant step!=0

    // ranges from start (inclusive) to end (exclusive) with step!=0
    public RangeIterator(int start, int end, int step) {
        // completare
    }

    // ranges from start (inclusive) to end (exclusive) with step 1
    public RangeIterator(int start, int end) { this(start, end, 1); }

    // ranges from 0 (inclusive) to end (exclusive) with step 1
    public RangeIterator(int end) { this(0, end); }

    @Override
    public boolean hasNext() {
        // completare
    }

    @Override
    public Integer next() {
        // completare
    }

    @Override
    public Iterator<Integer> iterator() { return this; }
}
```

class RangeIterator implements Iterator<Integer>, Iterable<Integer> {

```
private int next;
private final int end, step; // invariant step!=0
```

```
// ranges from start (inclusive) to end (exclusive) with step!=0
```

```
public RangeIterator(int start, int end, int step) {
    if (step == 0)
        throw new IllegalArgumentException("Step cannot be 0");
    this.next = start;
    this.end = end;
    this.step = step;
}
```

```
// ranges from start (inclusive) to end (exclusive) with step 1
```

```
public RangeIterator(int start, int end) {
    this(start, end, 1);
}
```

```
// ranges from 0 (inclusive) to end (exclusive) with step 1
```

```
public RangeIterator(int end) {
    this(0, end);
}
```

@Override

```
public boolean hasNext() {
    return step > 0 ? next < end : next > end;
}
```

@Override

```
public Integer next() {
    if (!hasNext())
        throw new NoSuchElementException();
    var res = next;
    next += step;
    return res;
}
```

@Override

```
public Iterator<Integer> iterator() {
    return this;
}
```

```
public class PairImp<T1, T2> implements Pair<T1, T2> {
    public final T1 first;
    public final T2 second;
```

```
public PairImp(T1 first, T2 second) {
    this.first = requireNonNull(first);
    this.second = requireNonNull(second);
}
```

```
public T1 getFirst() {
    return first;
}
```

```
public T2 getSecond() {
    return second;
}
```

@Override

```
public String toString() {
    return "(" + first + "," + second + ")";
}
```

```
public class Zipper<T1, T2> implements Iterator<Pair<T1, T2>> {
    private final Iterator<T1> iterator1;
    private final Iterator<T2> iterator2;
```

```
public Zipper(Iterable<T1> iterable1, Iterable<T2> iterable2) /* completare */
public boolean hasNext() /* completare */
public Pair<T1, T2> next() /* completare */
```

```
}
```

@Override

```
public boolean hasNext() {
    return iterator1.hasNext() && iterator2.hasNext();
}
```

@Override

```
public Pair<T1, T2> next() {
    if (!hasNext())
        throw new NoSuchElementException();
    return new PairImp<>(iterator1.next(), iterator2.next());
}
```

gennaio 2020

4. (a) Completare la classe `PairImp` che implementa coppie di valori diversi da `null`.

```
public interface Pair<T1, T2> {
    T1 getFirst();
    T2 getSecond();
}

public class PairImp<T1, T2> implements Pair<T1, T2> {
    public final T1 first; /* invariant: first != null */
    public final T2 second; /* invariant: second != null */

    public PairImp(T1 first, T2 second) /* completare */
    public T1 getFirst() /* completare */
    public T2 getSecond() /* completare */
    public String toString() /* completare */ // restituisce "(first, second)"
}
```

(b) Completare la classe `Zipper` che definisce iteratori zipper a partire da due iteratori `iterator1` e `iterator2` inizialmente ottenuti da due oggetti `iterable1` e `iterable2` di tipo `Iterable`. A ogni iterazione il metodo `next()` dello zipper restituisce la coppia di elementi rispettivamente ottenuti dai due iteratori `iterator1` e `iterator2` tramite il metodo `next()`; l'iterazione termina quando uno dei due iteratori `iterator1` e `iterator2` non ha più elementi da restituire.

```
public class Zipper<T1, T2> implements Iterator<Pair<T1, T2>> {
    private final Iterator<T1> iterator1;
    private final Iterator<T2> iterator2;

    public Zipper(Iterable<T1> iterable1, Iterable<T2> iterable2) /* completare */
    public boolean hasNext() /* completare */
    public Pair<T1, T2> next() /* completare */
}
```

Il seguente codice mostra un semplice esempio di uso della classe `Zipper`. Il metodo statico `java.util.Arrays.asList` crea una nuova lista a partire dagli argomenti passati; per esempio, `asList(1, 2, 3)` restituisce la lista `[1, 2, 3]`.

```
import static java.util.Arrays.asList;

public class Test {
    public static void main(String[] args) {
        Zipper<Integer, String> zipper = new Zipper<>(asList(1, 2, 3), asList("one", "two", "three"));
        while (zipper.hasNext())
            System.out.println(zipper.next()); // stampa (1,one) (2,two) (3,three)
        zipper = new Zipper<>(asList(1, 2), asList("one", "two", "three"));
        while (zipper.hasNext())
            System.out.println(zipper.next()); // stampa (1,one) (2,two)
        zipper = new Zipper<>(asList(1, 2, 3), asList("one", "two"));
        while (zipper.hasNext())
            System.out.println(zipper.next()); // stampa (1,one) (2,two)
    }
}
```

```
public class Zipper<T1, T2> implements Iterator<Pair<T1, T2>> {
    private final Iterator<T1> iterator1;
    private final Iterator<T2> iterator2;
```

```
public Zipper(Iterable<T1> iterable1, Iterable<T2> iterable2) {
    iterator1 = iterable1.iterator();
    iterator2 = iterable2.iterator();
}
```

@Override

```
public boolean hasNext() {
    return iterator1.hasNext() && iterator2.hasNext();
}
```

@Override

```
public Pair<T1, T2> next() {
    if (!hasNext())
        throw new NoSuchElementException();
    return new PairImp<>(iterator1.next(), iterator2.next());
}
```

giugno 2019

3. Completare la seguente classe di iteratori `Powers` per generare la sequenza di potenze di un numero intero in ordine crescente di esponente a partire da 1. Per esempio, il seguente codice

```
// genera la sequenza 31, 32, 33, 34
for (int n : new Powers(3, 4))
    System.out.println(n);

stampa la sequenza
3
9
27
81

import java.util.Iterator;
public class Powers implements Iterator<Integer>, Iterable<Integer> {

    private final int base; // base dell'esponente
    private int items; // numero di elementi ancora da generare
    private int next; // prossimo elemento da restituire

    // precondizione: items >= 0
    public Powers(int base, int items) {
        // completare
    }
    public boolean hasNext() {
        // completare
    }
    public Integer next() {
        // completare
    }
    // restituisce se stesso
    public Iterator<Integer> iterator() {
        // completare
    }
}
```

giugno 2019

3. Completare la seguente classe di iteratori `Multiples` per generare in ordine crescente la sequenza dei multipli di un numero. Per esempio, il seguente codice

```
for (int n : new Multiples(3, 4)) // genera i primi 4 multipli di 3
    System.out.println(n);

stampa la sequenza
3
6
9
12

import java.util.Iterator;
public class Multiples implements Iterator<Integer>, Iterable<Integer> {
    private final int step; // passo tra un elemento della sequenza e il suo seguente
    private final int max; // ultimo numero della sequenza
    private int next; // prossimo numero della sequenza

    /* parametro step: passo tra un elemento della sequenza e il suo seguente
     * parametro items: numero totale di elementi della sequenza
     * pre-condizione: step > 0 && items >= 0 */
    public Multiples(int step, int items) {
        // completare
    }
    public boolean hasNext() {
        // completare
    }
    public Integer next() {
        // completare
    }
    public Iterator<Integer> iterator() { // restituisce se stesso
        // completare
    }
}
```

```
public class Powers implements Iterator<Integer>, Iterable<Integer> {

    private final int base; // base dell'esponente
    private int items; // numero di elementi ancora da generare
    private int next; // prossimo elemento da restituire

    // precondizione: items >= 0
    public Powers(int base, int items) {
        if (items < 0)
            throw new IllegalArgumentException();
        this.base = this.next = base;
        this.items = items;
    }

    @Override
    public boolean hasNext() {
        return items > 0;
    }

    @Override
    public Integer next() {
        if (!hasNext())
            throw new NoSuchElementException();
        int res = next;
        next *= base;
        items--;
        return res;
    }

    // restituisce se stesso
    @Override
    public Iterator<Integer> iterator() {
        return this;
    }
}
```

```
public class Multiples implements Iterator<Integer>, Iterable<Integer> {

    private final int step;
    private final int max;
    private int next;

    public Multiples(int step, int items) {
        if (step <= 0 || items < 0)
            throw new IllegalArgumentException();
        this.step = this.next = step;
        this.max = step * items;
    }

    @Override
    public boolean hasNext() {
        return next <= max;
    }

    @Override
    public Integer next() {
        if (!hasNext())
            throw new NoSuchElementException();
        int res = next;
        next += step;
        return res;
    }

    @Override
    public Iterator<Integer> iterator() {
        return this;
    }
}
```

Luglio 2018

3. Completare la seguente classe `OddOnlyIterator` di iteratori definiti a partire da un iteratore di base `baseIterator` che restituiscono solo gli elementi di posizione dispari di `baseIterator` (considerando dispari la posizione del primo elemento).

```
public class OddOnlyIterator<E> implements Iterator<E> {

    private final Iterator<E> baseIterator; // non opzionale
    private boolean returnNext = true; /* stabilisce se il prossimo elemento di baseIterator va restituito; inizialmente true poiche' il primo elemento di baseIterator e' in posizione dispari */

    public OddOnlyIterator(Iterator<E> baseIterator) { /* completare */ }
    public boolean hasNext() { /* completare */ }
    public E next() { /* completare */ }
}
```

Per esempio, nel codice sottostante viene creato l'iteratore `altIt` a partire dall'iteratore `it` della lista [2, 4, 6, 8] e l'iterazione su `altIt` restituisce solo gli elementi 2 e 6.

```
Iterator<Integer> it = asList(2, 4, 6, 8).iterator();
Iterator<Integer> altIt = new OddonlyIterator<>(it);
while (altIt.hasNext())
    System.out.println(altIt.next()); // stampa 2\n6\n
```

```
public class OddOnlyIterator<E> implements Iterator<E> {

    private final Iterator<E> baseIterator; // non opzionale
    private boolean returnNext = true; /* stabilisce se il prossimo elemento di baseIterator va restituito; inizialmente true poiche' il primo elemento di baseIterator e' in posizione dispari */

    public OddOnlyIterator(Iterator<E> baseIterator) {
        this.baseIterator = requireNonNull(baseIterator);
    }

    @Override
    public boolean hasNext() {
        if (!baseIterator.hasNext())
            return false;
        if (returnNext)
            return true;
        baseIterator.next();
        returnNext = true;
        return baseIterator.hasNext();
    }

    @Override
    public E next() {
        if (!hasNext())
            throw new NoSuchElementException();
        returnNext = false;
        return baseIterator.next();
    }
}
```

giugno 2018

3. Completare la seguente classe CondIterator che permette di creare iteratori a partire da un iteratore di base baseIterator e un predicato cond che decide se l'iterazione di baseIterator può continuare.

```
import java.util.Iterator;
import java.util.function.Predicate; // public interface Predicate<E> {boolean test(E el);}
public class CondIterator<E> implements Iterator<E> {
    private final Iterator<E> baseIterator; // non opzionale
    private final Predicate<E> cond; // non opzionale
    private E cachedNext; // memorizza il prossimo elemento di baseIterator
    private boolean nextIsCached; /* true se e solo cachedNext contiene
        gia' il prossimo elemento di baseIterator;
        campo necessario perche' baseIterator
        potrebbe avere elementi null */
    // svuota la cache
    private void resetCache() { cachedNext = null; nextIsCached = false; }
    /* salva nella cache il prossimo elemento di baseIterator
     * nota bene: da usare solo se tale elemento esiste */
    private void cacheNext() { cachedNext = baseIterator.next(); nextIsCached = true; }
    public CondIterator(Iterator<E> baseIterator, Predicate<E> cond) { /* completare */ }
    /*
     * restituisce true se e solo se
     * baseIterator ha un prossimo elemento el e cond.test(el) restituisce true
     * nota bene: il prossimo elemento el potrebbe gia' essere in cache;
     * se non in cache, allora se esiste viene salvato in cache
     */
    public boolean hasNext() { /* completare */ }
    /*
     * lancia NoSuchElementException se non esiste un prossimo elemento,
     * altrimenti restituisce l'elemento in cache e fa reset della cache
     */
    public E next() { /* completare */ }
}
```

giugno 2018

3. (a) Completare la seguente classe LimitIterator che permette di creare iteratori con un limite massimo di elementi.

```
import java.util.Iterator;

public class LimitIterator<E> implements Iterator<E> {
    private final Iterator<E> baseIterator; // non opzionale
    private final int limit; // limite massimo elementi
    private int items = 0; // numero elementi restituiti

    public LimitIterator(Iterator<E> baseIterator, int limit) { /* completare */ }
    public boolean hasNext() { /* completare */ }
    public E next() { /* completare */ }
}
```

Per esempio, il codice sottostante crea un iteratore lim_it a partire dall'iteratore di numeri interi it, in modo che lim_it restituisca solo i primi 5 elementi che restituirebbe it.

```
Iterator<Integer> it = asList(1, 2, 3, 4, 5, 6, 7, 8, 9).iterator();
Iterator<Integer> lim_it = new LimitIterator<>(it, 5);
int i = 0;
while (lim_it.hasNext())
    assert ++i == lim_it.next();
assert i == 5;
```

- (b) Utilizzando la classe LimitIterator, implementare il seguente metodo found.

```
/*
 * restituisce true se e solo se it contiene tra i suoi primi limit elementi un
 * oggetto uguale a elem
 */
public static <E> boolean found(E elem, Iterator<E> it, int limit) {
    /* completare usando LimitIterator */
}
```

Per esempio, se un iteratore della classe GenBinLit genera la sequenza infinita di stringhe binarie "0", "1", "10", "11", "100", ..., allora il metodo found si comporta nel seguente modo:

```
assert !found("1111", new GenBinLit(), 7);
assert found("1111", new GenBinLit(), 15);
```

gennaio 2018

3. Completare il costruttore e i metodi della classe FunIterator<T> che permette di creare iteratori infiniti a partire da un oggetto di tipo Function<T, T> e un valore iniziale di tipo T; i valori di tipo Function<T, T> rappresentano funzioni da valori di tipo T a valori di tipo T.

```
// functions from T to R
public interface Function<T, R> {
    R apply(T t); // applies the function to t
}

public class FunIterator<T> implements Iterator<T> {
    private final Function<T, T> fun;
    private T first; // allowed to be null

    public FunIterator(Function<T, T> fun, T first) {
        // completare
    }

    public boolean hasNext() {
        // completare
    }

    public T next() {
        // completare
    }
}
```

Per esempio, le assert nel seguente frammento di codice sono sempre verificate:

```
Function<String, String> append = // oggetto che rappresenta la funzione OCaml fun x -> "a" ^ x
Iterator<String> it = new FunIterator<String>(append, "");
assert it.next().equals("");
assert it.next().equals("a");
assert it.next().equals("aa");
```

```
public class CondIterator<E> implements Iterator<E> {
    private final Iterator<E> baseIterator; // non opzionale
    private final Predicate<E> cond; // non opzionale
    private E cachedNext; // memorizza il prossimo elemento di baseIterator
    private boolean nextIsCached; // true se e solo cachedNext contiene gia' il prossimo elemento di baseIterator;
    // campo necessario perche' baseIterator potrebbe avere elementi null
    /*
     * svuota la cache
     */
    private void resetCache() {
        cachedNext = null;
        nextIsCached = false;
    }
    /* salva nella cache il prossimo elemento di baseIterator
     * nota bene: da usare solo se tale elemento esiste */
    private void cacheNext() {
        cachedNext = baseIterator.next();
        nextIsCached = true;
    }
    public CondIterator(Iterator<E> baseIterator, Predicate<E> cond) {
        this.cond = requireNonNull(cond);
        this.baseIterator = requireNonNull(baseIterator);
    }
    /*
     * restituisce true se e solo se baseIterator ha un prossimo elemento el e
     * cond.test(el) restituisce true nota bene: il prossimo elemento el potrebbe
     * gia' essere in cache; se non in cache, allora se esiste viene salvato in cache
     */
    @Override
    public boolean hasNext() {
        if (!nextIsCached) {
            if (!baseIterator.hasNext())
                return false;
            cacheNext();
        }
        return cond.test(cachedNext);
    }
    /*
     * lancia NoSuchElementException se non esiste un prossimo elemento, altrimenti
     * restituisce l'elemento in cache e fa reset della cache
     */
    @Override
    public E next() {
        if (!hasNext())
            throw new NoSuchElementException();
        E res = cachedNext;
        resetCache();
        return res;
    }
}

public class LimitIterator<E> implements Iterator<E> {
    private final Iterator<E> baseIterator; // non opzionale
    private final int limit;
    private int items = 0; // numero elementi restituiti

    public LimitIterator(Iterator<E> baseIterator, int limit) {
        this.baseIterator = requireNonNull(baseIterator);
        this.limit = limit;
    }
    @Override
    public boolean hasNext() {
        return baseIterator.hasNext() && items < limit;
    }
    @Override
    public E next() {
        // genera uno stream infinito di stringhe binarie
        public class GenBinLit implements Iterator<String> {
            private int count = 1;
            @Override
            public boolean hasNext() {
                return true;
            }
            @Override
            public String next() {
                return Integer.toBinaryString(count++);
            }
        }
        public class Utils {
            /*
             * restituisce true se e solo se it contiene tra i suoi primi limit elementi un
             * oggetto uguale a elem
             */
            public static <E> boolean found(E elem, Iterator<E> it, int limit) {
                Iterator<E> limIt = new LimitIterator<>(it, limit);
                while (limIt.hasNext())
                    if (Objects.equals(limIt.next(), elem))
                        return true;
                    return false;
            }
            public static void main(String[] args) {
                assert !found("1111", new GenBinLit(), 7);
                assert found("1111", new GenBinLit(), 15);
            }
        }
    }
}

public class FunIterator<T> implements Iterator<T> {
    private final Function<T, T> fun;
    private T first; // allowed to be null

    public FunIterator(Function<T, T> fun, T first) {
        this.fun = requireNonNull(fun);
        this.first = first;
    }
    @Override
    public boolean hasNext() {
        return true;
    }
    @Override
    public T next() {
        T res = first;
        first = fun.apply(first);
        return res;
    }
}
```

Settembre 2017

3. La seguente classe `FilteredIterator` permette di iterare sugli elementi appartenenti a una lista di tipo `ArrayList<E>` che soddisfano un predicato di tipo `Predicate<E>`.

```
public interface Predicate<E> {
    boolean test(E t);
}

public class FilteredIterator<E> implements Iterator<E> {
    private final Predicate<E> pred;
    private final ArrayList<E> list;
    private int curr; // index of the current element

    public FilteredIterator(Predicate<E> pred, ArrayList<E> list) { /* da completare */ }
    public boolean hasNext() { /* da completare */ }
    public E next() { /* da completare */ }
}
```

Per esempio, le `assert` nel seguente frammento di codice sono sempre verificate:

```
Predicate<Integer> odd = ... // predicato che testa se un intero e' dispari
ArrayList<Integer> list = ... // lista [1, 2, 4, 3, 5, 6]
FilteredIterator<Integer> fit = new FilteredIterator<>(odd, list);
int elem = 1;
int count = 0;
while (fit.hasNext()) {
    assert fit.next().equals(elem);
    elem += 2;
    count++;
}
assert count == 3;
```

(a) Completare le definizioni del costruttore della classe.

(b) Completare le definizioni dei metodi `hasNext()` e `next()`.

(c) Utilizzando la classe `FilteredIterator`, completare il metodo `find` che restituisce il primo elemento di una lista di tipo `ArrayList<E>` che verifica un predicato di tipo `Predicate<E>`, se tale elemento esiste, o solleva l'eccezione `NoSuchElementException` altrimenti.

```
public static <E> E find(Predicate<E> pred, ArrayList<E> list) { /* completare */ }
```

Per esempio, la seguente `assert` ha successo:

```
Predicate<Integer> positive = ... // predicato che testa se un numero intero e' positivo
ArrayList<Integer> list = ... // lista [-1, -2, -3, -4, 5, 6]
assert find(positive, list).equals(5);
```

febbraio 2017

3. Considerare la seguente implementazione di successioni finite di interi tale che `new IntSeq(min, max, step)` rappresenta l'insieme di interi $\{min + k \cdot step \mid k \geq 0 \text{ e } min + k \cdot step \leq max\}$.

Per esempio, `new IntSeq(1, 10, 4)` corrisponde all'insieme $\{1, 5, 9\}$.

```
public class IntSeq implements Iterable<Integer> {
    /* implements the set { min+k*step | k >= 0 and min+k*step <= max } */
    private final int min;
    private final int max;
    private final int step; // invariant: step > 0

    public IntSeq(int min, int max) { // default step is 1
        ...
    }
    public IntSeq(int min, int max, int step) {
        ...
    }
    public int getMin() {
        ...
    }
    public int getMax() {
        ...
    }
    public int getStep() {
        ...
    }
    public Iterator<Integer> iterator() {
        ...
    }
}
class IntSeqIterator implements Iterator<Integer> {
    private int min;
    private final int max;
    private final int step;

    public IntSeqIterator(IntSeq seq) {
        ...
    }
    public boolean hasNext() {
        ...
    }
    public Integer next() {
        ...
    }
}
```

(a) Completare le definizioni dei costruttori e dei metodi `getter` della classe `IntSeq`.

(b) Completare la definizione del metodo `iterator()` della classe `IntSeq`.

(c) Completare la definizione del costruttore della classe `IntSeqIterator`.

(d) Completare le definizioni dei metodi `hasNext()` e `next()` della classe `IntSeqIterator`.

```
public class FilteredIterator<E> implements Iterator<E> {
    private final Predicate<E> pred;
    private final ArrayList<E> list;
    private int curr;

    public FilteredIterator(Predicate<E> pred, ArrayList<E> list) {
        this.pred = requireNonNull(pred);
        this.list = requireNonNull(list);
    }

    @Override
    public boolean hasNext() {
        while (curr < list.size()) {
            if (pred.test(list.get(curr)))
                return true;
            curr++;
        }
        return false;
    }

    @Override
    public E next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return list.get(curr++);
    }
}

public class Find {
    public static <E> E find(Predicate<E> pred, ArrayList<E> list) {
        return new FilteredIterator<>(pred, list).next();
    }

    public static void main(String[] args) {
        Predicate<Integer> odd = x -> x % 2 != 0;
        FilteredIterator<Integer> fit = new FilteredIterator<>(odd, new ArrayList<>(asList(1, 2, 4, 3, 5, 6)));
        int elem = 1;
        int count = 0;
        while (fit.hasNext()) {
            assert fit.next().equals(elem);
            elem += 2;
            count++;
        }
        assert count == 3;
        Predicate<Integer> positive = x -> x > 0;
        assert find(positive, new ArrayList<>(asList(-1, -2, -3, -4, 5, 6))).equals(5);
    }
}

public class IntSeq implements Iterable<Integer> {
    /* implements the set { min+k*step | k >= 0 and min+k*step <= max } */
    private final int min;
    private final int max;
    private final int step; // invariant: step > 0

    public IntSeq(int min, int max) {
        this(min, max, 1);
    }

    public IntSeq(int min, int max, int step) {
        if (step <= 0)
            throw new IllegalArgumentException();
        this.min = min;
        this.max = max;
        this.step = step;
    }

    public int getMin() {
        return min;
    }

    public int getMax() {
        return max;
    }

    public int getStep() {
        return step;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new IntSeqIterator(this);
    }
}

class IntSeqIterator implements Iterator<Integer> {
    private int min;
    private final int max;
    private final int step;

    public IntSeqIterator(IntSeq seq) {
        this.min = seq.getMin();
        this.max = seq.getMax();
        this.step = seq.getStep();
    }

    @Override
    public boolean hasNext() {
        return min <= max;
    }

    @Override
    public Integer next() {
        if (!hasNext())
            throw new NoSuchElementException();
        int res = min;
        min += step;
        return res;
    }
}
```

Settembre 2016

3. (a) Completare la classe `CatIterator<E>` che implementa la concatenazione di due iteratori `it1` e `it2` su elementi di tipo `E`. L'iteratore ottenuto dalla concatenazione di `it1` e `it2` restituisce nell'ordine prima tutti gli elementi di `it1` e poi quelli di `it2`.

```
import java.util.Iterator;
public class CatIterator<E> implements Iterator<E> {
    private final Iterator<E> it1;
    private final Iterator<E> it2;
    public CatIterator(Iterator<E> it1, Iterator<E> it2) { /* da completare */ }
    @Override public boolean hasNext() { /* da completare */ }
    @Override public E next() { /* da completare */ }
}
```

- (b) Completare la classe `CombIterator` che permette di combinare due iteratori `it1` e `it2`, rispettivamente di tipo `Iterator<T1>` e `Iterator<T2>`, tramite il metodo `R apply(T1 t1, T2 t2)` di un oggetto `comb` di tipo `BiFunction<T1, T2, R>`.

A ogni iterazione l'iteratore ottenuto combinando `it1` e `it2` si comporta nel seguente modo:

- se entrambi gli iteratori hanno, rispettivamente, un prossimo elemento e_1 ed e_2 , allora viene restituito `comb.apply(e_1, e_2)` come prossimo elemento e l'iterazione avanza per entrambi gli iteratori;
- se solo `it1` ha un prossimo elemento e_1 , allora viene restituito `comb.apply(e_1, null)` come prossimo elemento e l'iterazione avanza solo per `it1`;
- se solo `it2` ha un prossimo elemento e_2 , allora viene restituito `comb.apply(null, e_2)` come prossimo elemento e l'iterazione avanza solo per `it2`;
- se nessuno dei due iteratori ha un prossimo elemento, allora l'iterazione termina.

```
import java.util.Iterator;
public interface BiFunction<T, U, R> { R apply(T t, U u); }
public class CombIterator<T1, T2, R> implements Iterator<R> {
    private final Iterator<T1> it1;
    private final Iterator<T2> it2;
    private final BiFunction<T1, T2, R> comb;
    public CombIterator(Iterator<T1> it1, Iterator<T2> it2,
        BiFunction<T1, T2, R> comb) { /* da completare */ }
    @Override public boolean hasNext() { /* da completare */ }
    @Override public R next() { /* da completare */ }
}
```

- (c) Completare la classe `MergeIterator` che permette di combinare due iteratori `it1` e `it2` di tipo `Iterator<Integer>` per ottenere un nuovo iteratore di tipo `Iterator<Integer>`, assumendo che entrambi gli iteratori non restituiscono mai il valore `null`.

A ogni iterazione l'iteratore ottenuto combinando `it1` e `it2` si comporta nel seguente modo:

- se entrambi gli iteratori hanno, rispettivamente, un prossimo elemento i_1 e i_2 , allora se $i_1 \leq i_2$ viene restituito come prossimo elemento i_1 e l'iterazione avanza solo per `it1`, altrimenti (ossia, se $i_1 > i_2$) viene restituito come prossimo elemento i_2 e l'iterazione avanza solo per `it2`;
- se solo `it1` ha un prossimo elemento i_1 , allora viene restituito i_1 come prossimo elemento e l'iterazione avanza solo per `it1`;
- se solo `it2` ha un prossimo elemento i_2 , allora viene restituito i_2 come prossimo elemento e l'iterazione avanza solo per `it2`;
- se nessuno dei due iteratori ha un prossimo elemento, allora l'iterazione termina.

```
public class CatIterator<E> implements Iterator<E> {
    private final Iterator<E> it1;
    private final Iterator<E> it2;

    public CatIterator(Iterator<E> it1, Iterator<E> it2) {
        if (it1 == null || it2 == null)
            throw new NullPointerException();
        this.it1 = it1;
        this.it2 = it2;
    }

    @Override
    public boolean hasNext() {
        return it1.hasNext() || it2.hasNext();
    }

    @Override
    public E next() {
        if (it1.hasNext())
            return it1.next();
        if (it2.hasNext())
            return it2.next();
        throw new NoSuchElementException();
    }
}
```

```
public class CombIterator<T1, T2, R> implements Iterator<R> {
    private final Iterator<T1> it1;
    private final Iterator<T2> it2;
    private final BiFunction<T1, T2, R> comb;

    public CombIterator(Iterator<T1> it1, Iterator<T2> it2,
        BiFunction<T1, T2, R> comb) {
        if (it1 == null || it2 == null || comb == null)
            throw new NullPointerException();
        this.it1 = it1;
        this.it2 = it2;
        this.comb = comb;
    }

    @Override
    public boolean hasNext() {
        return it1.hasNext() || it2.hasNext();
    }

    @Override
    public R next() {
        if (it1.hasNext())
            return comb.apply(it1.next(), it2.hasNext() ? it2.next() : null);
        if (it2.hasNext())
            return comb.apply(null, it2.next());
        throw new NoSuchElementException();
    }
}
```

```
public class MergeIterator implements Iterator<Integer> {
    private final Iterator<Integer> it0;
    private final Iterator<Integer> it1;
    // curr[0] current element of it0, curr[1] current element of it1
    private Integer curr[] = new Integer[2];

    private Integer tryNext(Iterator<Integer> it) {
        if (it.hasNext())
            return it.next();
        return null;
    }

    private Integer advance(int i, Iterator<Integer> it) {
        Integer res = curr[i];
        curr[i] = tryNext(it);
        return res;
    }

    public MergeIterator(Iterator<Integer> it0, Iterator<Integer> it1) {
        if (it0 == null || it1 == null)
            throw new NullPointerException();
        this.it0 = it0;
        this.it1 = it1;
        curr[0] = tryNext(it0);
        curr[1] = tryNext(it1);
    }

    @Override
    public boolean hasNext() {
        return curr[0] != null || curr[1] != null;
    }

    @Override
    public boolean hasNext() {
        return curr[0] != null || curr[1] != null;
    }

    @Override
    public Integer next() {
        if (curr[0] != null)
            if (curr[1] == null || curr[0] <= curr[1])
                return advance(0, it0);
            else
                return advance(1, it1);
        if (curr[1] == null)
            if (curr[0] == null || curr[1] <= curr[0])
                return advance(1, it1);
            else
                return advance(0, it0);
        throw new NoSuchElementException();
    }
}
```

luglio 2015

3. (a) Considerare la classe `ArrayBinNum` che implementa i numeri binari memorizzando in un array i loro bit nell'ordine usuale (ossia, a partire dal bit più significativo). Per esempio, le asserzioni nel seguente frammento di codice hanno tutte successo:

```
assert new ArrayBinNum(0, 0, 0, 1).decode() == 1;
assert new ArrayBinNum(0, 0, 0, 1).length() == 4;
assert new ArrayBinNum(1, 1, 1).decode() == 7;
assert new ArrayBinNum(1, 1, 1).length() == 3;
```

Completare la seguente definizione della classe `ArrayBinNum`:

```
import java.util.Iterator;
public interface BinNum extends Iterable<Byte> {
    Iterator<Byte> revIterator();
    long decode();
    BinNum add(BinNum other);
    int length();
}
import java.util.Iterator;

public abstract class AbstractBinNum implements BinNum {
    @Override
    public long decode() {
        // da completare
    }
    @Override
    public BinNum add(BinNum other) {
        // da completare
    }
}
import java.util.Iterator;
public class ArrayBinNum extends AbstractBinNum {
    private final byte[] digits;
    /** new ArrayBinNum() equivale a new ArrayBinNum(0) */
    public ArrayBinNum(int... digits) {
        // da completare
    }
    /** itera sulle cifre binarie a partire da quella piu' significativa */
    public Iterator<Byte> iterator() {
        return new BinNumIterator(digits);
    }
    /** itera sulle cifre binarie a partire da quella meno significativa */
    public Iterator<Byte> revIterator() {
        return new RevBinNumIterator(digits);
    }
    /** restituisce il numero di cifre binarie */
    @Override
    public int length() {
        // da completare
    }
}
```

- (b) Completare i metodi `decode` e `add` della classe astratta `AbstractBinNum`. Il metodo `decode` decodifica il numero binario.

Per esempio, `assert new ArrayBinNum(1, 1, 1).decode() == 7;` ha successo.

Il metodo `add` restituisce il numero binario ottenuto sommando il numero binario `this` con il numero binario `other`. Per esempio, `new ArrayBinNum(1, 1, 1).add(new ArrayBinNum(1))` deve restituire un'istanza di `ArrayBinNum` che rappresenta il numero binario 1000.

- (c) Completare la classe `BinNumIterator` che itera sulle cifre binarie.

```
import java.util.Iterator;
import java.util.NoSuchElementException;
class BinNumIterator implements Iterator<Byte> {
    private int index;
    private final byte[] digits;
    BinNumIterator(byte[] digits) {
        if (digits == null)
            throw new NullPointerException();
        this.digits = digits;
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Byte next() {
        // da completare
    }
}
```

- (d) Completare la classe `RevBinNumIterator` che itera sulle cifre binarie in ordine inverso.

```
import java.util.Iterator;
import java.util.NoSuchElementException;
class RevBinNumIterator implements Iterator<Byte> {
    private int index;
    private final byte[] digits;
    RevBinNumIterator(byte[] digits) {
        // da completare
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Byte next() {
        // da completare
    }
}
```

```
public abstract class AbstractBinNum implements BinNum {
    @Override
    public long decode() {
        long res = 0;
        for (byte digit : this)
            res = 2 * res + digit;
        return res;
    }
    @Override
    public BinNum add(BinNum other) {
        int index = Math.max(length(), other.length());
        final int[] digits = new int[index + 1];
        Iterator<Byte> thisIt = this.revIterator();
        Iterator<Byte> otherIt = other.revIterator();
        int carry = 0;
        while (thisIt.hasNext() && otherIt.hasNext()) {
            int sum = thisIt.next() + otherIt.next() + carry;
            digits[index--] = sum % 2;
            carry = sum / 2;
        }
        Iterator<Byte> contIt = thisIt.hasNext() ? thisIt : otherIt;
        while (contIt.hasNext()) {
            int sum = contIt.next() + carry;
            digits[index--] = sum % 2;
            carry = sum / 2;
        }
        digits[0] = carry;
        return new ArrayBinNum(digits);
    }
}

public class ArrayBinNum extends AbstractBinNum {
    private final byte[] digits;
    /** new ArrayBinNum() equivale a new ArrayBinNum(0) */
    public ArrayBinNum(int... digits) {
        int length = digits.length;
        if (length == 0) {
            this.digits = new byte[1];
            return;
        }
        this.digits = new byte[length];
        for (int index = 0; index < length; index++) {
            int digit = digits[index];
            if (digit != 0 && digit != 1)
                throw new IllegalArgumentException();
            this.digits[index] = (byte) digit;
        }
    }
    /** itera sulle cifre binarie a partire da quella piu' significativa */
    public Iterator<Byte> iterator() {
        return new BinNumIterator(digits);
    }
    /** itera sulle cifre binarie a partire da quella meno significativa */
    public Iterator<Byte> revIterator() {
        return new RevBinNumIterator(digits);
    }
    /** restituisce il numero di cifre binarie */
    @Override
    public int length() {
        return digits.length;
    }
}

class BinNumIterator implements Iterator<Byte> {
    private int index;
    private final byte[] digits;
    BinNumIterator(byte[] digits) {
        if (digits == null)
            throw new NullPointerException();
        this.digits = digits;
    }
    @Override
    public boolean hasNext() {
        return index < digits.length;
    }
    @Override
    public Byte next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return digits[index++];
    }
}

class RevBinNumIterator implements Iterator<Byte> {
    private int index;
    private final byte[] digits;
    RevBinNumIterator(byte[] digits) {
        index = digits.length - 1;
        this.digits = digits;
    }
    @Override
    public boolean hasNext() {
        return index >= 0;
    }
    @Override
    public Byte next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return digits[index--];
    }
}
```

giugno 2015

3. (a) Completare la definizione della classe `ArrayPoly` che implementa i polinomi a una variabile rappresentandoli con l'array dei loro coefficienti in ordine decrescente di grado. Per esempio, `new ArrayPoly(1., 0.5, -2., 0.)` rappresenta il polinomio $x^3 + \frac{1}{2}x^2 - 2x$, mentre `new ArrayPoly(-2., 0., 0.)` rappresenta $-2x^2$.

```

public interface Polynomial extends Iterable<Double> {
    Iterator<Double> revIterator();
    int degree();
    double eval(double val);
    Polynomial add(Polynomial other);
}

public abstract class AbstractPoly implements Polynomial {
    @Override
    public double eval(double val) {
        ...
    }
    @Override
    public Polynomial add(Polynomial other) {
        ...
    }
}

public class ArrayPoly extends AbstractPoly {
    private final double[] coeffs;
    /* new ArrayPoly() deve essere equivalente a new ArrayPoly(0.) */
    public ArrayPoly(double... coeffs) {
        // da completare
    }
    /** itera sui coefficienti a partire da quello di grado massimo */
    public Iterator<Double> iterator() {
        return new PolyIterator(coeffs);
    }
    /** itera sui coefficienti a partire da quello di grado 0 */
    public Iterator<Double> revIterator() {
        return new RevPolyIterator(coeffs);
    }
    /** restituisce il grado del polinomio */
    @Override
    public int degree() {
        // da completare
    }
}

```

- (b) Completare i metodi `eval` e `add` della classe astratta `AbstractPoly`. Il metodo `eval` valuta il polinomio sostituendo all'`incognita` il valore `val`.

Per esempio, `assert new ArrayPoly(4., -4., 1.).eval(3.) == 25.` ha successo.

Il metodo `add` restituisce il polinomio ottenuto sommando il polinomio `this` con il polinomio `other`. Per esempio, `new ArrayPoly(4., -4., 1.).add(new ArrayPoly(1., 0., 0., -1.))` deve restituire un'istanza di `ArrayPoly` che rappresenta il polinomio $x^3 + 4x^2 - 4x$.

```

public abstract class AbstractPoly implements Polynomial {
    @Override
    public double eval(double val) {
        // da completare
    }
    @Override
    public Polynomial add(Polynomial other) {
        // da completare
    }
}

```

- (c) Completare la classe `PolyIterator` che itera su un array di `double`.

```

class PolyIterator implements Iterator<Double> {
    private int index;
    private final double[] coeffs;
    PolyIterator(double[] coeffs) {
        if (coeffs == null)
            throw new NullPointerException();
        this.coeffs = coeffs;
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Double next() {
        // da completare
    }
}

```

- (d) Completare la classe `RevPolyIterator` che itera su un array di `double` in ordine inverso.

```

class RevPolyIterator implements Iterator<Double> {
    private int index;
    private final double[] coeffs;
    RevPolyIterator(double[] coeffs) {
        index = coeffs.length - 1;
        this.coeffs = coeffs;
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Double next() {
        // da completare
    }
}

```

```

public class ArrayPoly extends AbstractPoly {
    private final double[] coeffs;
    public ArrayPoly(double... coeffs) {
        int length = coeffs.length;
        int maxCoeff = 0;
        for (; maxCoeff < length; maxCoeff++)
            if (coeffs[maxCoeff] != 0.)
                break;
        if (maxCoeff == length)
            this.coeffs = new double[1];
        else {
            length -= maxCoeff;
            this.coeffs = new double[length];
            int i = 0;
            do {
                this.coeffs[i] = coeffs[i] + maxCoeff;
                i++;
            } while (i < length);
        }
    }
    public Iterator<Double> iterator() {
        return new PolyIterator(coeffs);
    }
    public Iterator<Double> revIterator() {
        return new RevPolyIterator(coeffs);
    }
    @Override
    public int degree() {
        return coeffs.length - 1;
    }
}

class RevPolyIterator implements Iterator<Double> {
    private int index;
    private final double[] coeffs;
    RevPolyIterator(double[] coeffs) {
        index = coeffs.length - 1;
        this.coeffs = coeffs;
    }
    @Override
    public boolean hasNext() {
        return index >= 0;
    }
    @Override
    public Double next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return coeffs[index--];
    }
}

class PolyIterator implements Iterator<Double> {
    private int index;
    private final double[] coeffs;
    PolyIterator(double[] coeffs) {
        if (coeffs == null)
            throw new NullPointerException();
        this.coeffs = coeffs;
    }
    @Override
    public boolean hasNext() {
        return index < coeffs.length;
    }
    @Override
    public Double next() {
        if (!hasNext())
            throw new NoSuchElementException();
        return coeffs[index++];
    }
}

public abstract class AbstractPoly implements Polynomial {
    @Override
    public double eval(double val) {
        double res = 0;
        for (double coeff : this)
            res = val * res + coeff;
        return res;
    }
    @Override
    public Polynomial add(Polynomial other) {
        int i = Math.max(degree(), other.degree());
        final double[] coeffs = new double[i + 1];
        Iterator<Double> thisIt = this.revIterator();
        Iterator<Double> otherIt = other.revIterator();
        while (thisIt.hasNext() && otherIt.hasNext())
            coeffs[i--] = thisIt.next() + otherIt.next();
        Iterator<Double> contIt = thisIt.hasNext() ? thisIt : otherIt;
        while (contIt.hasNext())
            coeffs[i--] = contIt.next();
        return new ArrayPoly(coeffs);
    }
}

```

febbraio 2015

3. (a) Completare la definizione della classe `KeepPositive` con il metodo `keep` in modo che restituisca `true` se e solo se il suo argomento è positivo.

```
public interface Filter<T> {
    boolean keep(T t);
}

public class KeepPositive implements Filter<Integer> {
    // completare
}
```

- (b) Completare il costruttore e il metodo `hasCurrent` della classe `EnhancedIteratorClass<T>` che permette di estendere le funzionalità di un oggetto di tipo `Iterator<T>`.

```
import java.util.Iterator;

public interface EnhancedIterator<T> extends Iterator<T> {
    boolean hasCurrent();
    T getCurrent();
    boolean moveNext();
}

import java.util.Iterator;
import java.util.NoSuchElementException;

public class EnhancedIteratorClass<T> implements EnhancedIterator<T> {
    private final Iterator<T> iterator;
    private T current;
    private boolean hasCurrent;

    public EnhancedIteratorClass(Iterator<T> iterator) {
        // completare
    }

    @Override
    public boolean hasCurrent() {
        // completare
    }

    @Override
    public T getCurrent() {
        if (!hasCurrent())
            throw new NoSuchElementException();
        return current;
    }

    @Override
    public boolean hasNext() {
        return iterator.hasNext();
    }

    @Override
    public T next() {
        if (moveNext())
            return current;
        else
            throw new NoSuchElementException();
    }

    @Override
    public boolean moveNext() {
        if (hasCurrent = hasNext())
            current = iterator.next();
        return hasCurrent;
    }
}
```

- (c) Utilizzando la classe `EnhancedIteratorClass<T>` completare il costruttore e i metodi `hasNext` e `next` della classe `FilteredIterator<T>` che permette di costruire un nuovo iteratore i' , a partire da un altro iteratore i e da un filtro f ; i' restituisce nello stesso ordine tutti e soli gli elementi restituiti da i che sono filtrati da f , ossia: gli elementi i tali che $f.keep(i)$ si valuta in `true`.

Esempio:

```
public class KeepNotEmpty implements Filter<String> {
    @Override
    public boolean keep(String st) {
        return st != null && st.length() > 0;
    }
}

...
java.util.List<String> l = java.util.Arrays.asList(null, "a", "", "ab", "");
FilteredIterator<String> it = new FilteredIterator<>(l.iterator(), new KeepNotEmpty());
while(it.hasNext())
    System.out.print(it.next().length() + " "); // stampa 1 2
```

Definizione della classe `FilteredIterator<T>`:

```
import java.util.Iterator;

public class FilteredIterator<T> implements Iterator<T> {
    private final EnhancedIterator<T> iterator;
    private final Filter<T> filter;

    public FilteredIterator(Iterator<T> iterator, Filter<T> filter) {
        // completare
    }

    @Override
    public boolean hasNext() {
        // completare
    }

    @Override
    public T next() {
        // completare
    }
}
```

- (d) Utilizzando le classi `FilteredIterator<T>` e `KeepPositive`, completare la definizione del metodo `getAllPositive` che presa una collezione di interi, restituisce un iteratore che genera tutti e solo gli elementi positivi della collezione.

```
import java.util.Collection;
public class Util {
    public static FilteredIterator<Integer> getAllPositive(
        Collection<Integer> col) {
        // completare
    }
}
```

```
public class KeepPositive implements Filter<Integer> {
```

```
    public boolean keep(Integer i) {
        return i != null && i > 0;
    }
}
```

```
public class EnhancedIteratorClass<T> implements EnhancedIterator<T> {
    private final Iterator<T> iterator;
    private T current;
    private boolean hasCurrent;

    public EnhancedIteratorClass(Iterator<T> iterator) {
        if (iterator == null)
            throw new IllegalArgumentException();
        this.iterator = iterator;
    }

    @Override
    public boolean hasCurrent() {
        return hasCurrent;
    }

    @Override
    public T getCurrent() {
        if (!hasCurrent())
            throw new NoSuchElementException();
        return current;
    }

    @Override
    public boolean hasNext() {
        return iterator.hasNext();
    }

    @Override
    public T next() {
        if (moveNext())
            return current;
        else
            throw new NoSuchElementException();
    }

    @Override
    public boolean moveNext() {
        if (hasCurrent = hasNext())
            current = iterator.next();
        return hasCurrent;
    }
}
```

```
public class FilteredIterator<T> implements Iterator<T> {
```

```
    private final EnhancedIterator<T> iterator;
    private final Filter<T> filter;
```

```
    public FilteredIterator(Iterator<T> iterator, Filter<T> filter) {
        if (filter == null)
            throw new IllegalArgumentException();
        this.iterator = new EnhancedIteratorClass<>(iterator);
        this.filter = filter;
        findNext();
    }
```

```
    private void findNext() {
        while (iterator.hasNext())
            if (filter.keep(iterator.next()))
                return;
        iterator.moveNext();
    }
}
```

```
@Override
public boolean hasNext() {
    return iterator.hasNext();
}
```

```
@Override
public T next() {
    if (!hasNext())
        throw new NoSuchElementException();
    T res = iterator.getCurrent();
    findNext();
    return res;
}
```

```
public class Util {
```

```
    public static FilteredIterator<Integer> getAllPositive(
        Collection<Integer> col) {
        return new FilteredIterator<>(col.iterator(), new KeepPositive());
    }
}
```

```
    public static void test(Collection<Integer> col) {
        out.print("[ ");
        FilteredIterator<Integer> it = getAllPositive(col);
        while (it.hasNext())
            out.print(it.next() + " ");
        out.println("]");
    }
}
```

```
    public static void main(String[] args) {
        test(asList(-3, 1, -2, -4, 2, -3, 3, -10));
        test(asList(-3, -1, -2, -4, -2, -3, -3, -10));
        test(asList(1, 2, 3));
        test(asList(1, 0, 0, 0, 2));
    }
}
```

```
    List<String> l = asList(null, "a", "", "ab", "");
    FilteredIterator<String> it = new FilteredIterator<>(l.iterator(),
        new KeepNotEmpty());
    // stampa 1 2
    while (it.hasNext())
        out.print(it.next().length() + " ");
}
```

gennaio 2015

3. (a) Completare la definizione del costruttore e dei metodi della classe `Pair<E1,E2>` che implementa coppie di oggetti rispettivamente di tipo `E1` ed `E2`.

```
public final class Pair<E1, E2> {
    private final E1 first;
    private final E2 second;

    public Pair(E1 first, E2 second) {
        // completare
    }
    public E1 getFirst() {
        // completare
    }
    public E2 getSecond() {
        // completare
    }

    @Override
    public int hashCode() {
        // completare
    }
    @Override
    public boolean equals(Object obj) {
        // completare
    }
    @Override
    public String toString() {
        // completare
    }
}
```

- (b) Considerare la classe `ComposeIterator<E1, E2, T>` che permette di comporre, tramite un oggetto di tipo `Composer<E1, E2, T>`, due iteratori rispettivamente di tipo `Iterator<E1>` e `Iterator<E2>`, per ottenere un iteratore di tipo `Iterator<T>`. L'iteratore produce elementi ottenuti componendo, tramite l'oggetto `composer`, le coppie di elementi via via restituite da `firstIterator` e `secondIterator`. L'iterazione termina quando uno dei due iteratori termina.

Esempio di uso:

```
public class AddLengths implements Composer<String, String, Integer> {
    @Override
    public Integer compose(String e1, String e2) {
        return e1.length() + e2.length();
    }
}

Iterator<String> it1 = Arrays.asList("a", "ab").iterator();
Iterator<String> it2 = Arrays.asList("", "abc", "a").iterator();
Iterator<Integer> it3 = new ComposeIterator<E1, E2, T>(it1, it2, new AddLengths());
while(it3.hasNext())
    // stampa "1 5" ossia la lunghezza di "a+"" e di "ab"+"abc"
    System.out.print(it3.next()+" ");
}
```

Completare la definizione del costruttore e dei metodi della classe `ComposeIterator<E1, E2, T>`.

```
public interface Composer<E1, E2, T> {
    T compose(E1 e1, E2 e2);
}

public class ComposeIterator<E1, E2, T> implements Iterator<T> {
    private final Iterator<E1> firstIterator;
    private final Iterator<E2> secondIterator;
    private final Composer<E1, E2, T> composer;

    public ComposeIterator(Iterator<E1> firstIterator,
                          Iterator<E2> secondIterator, Composer<E1, E2, T> composer) {
        // completare
    }

    @Override
    public boolean hasNext() {
        // completare
    }

    @Override
    public T next() {
        // completare
    }
}
```

```
public final class Pair<E1, E2> {
    private final E1 first;
    private final E2 second;

    public Pair(E1 first, E2 second) {
        this.first = first;
        this.second = second;
    }

    public E1 getFirst() {
        return first;
    }

    public E2 getSecond() {
        return second;
    }

    @Override
    public int hashCode() {
        return 31 * ((first == null) ? 0 : first.hashCode())
               + ((second == null) ? 0 : second.hashCode());
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Pair))
            return false;
        Pair<?, ?> pair = (Pair<?, ?>) obj;
        return first.equals(pair.first) && second.equals(pair.second);
    }

    @Override
    public String toString() {
        return "(" + first + "," + second + ")";
    }
}

public class ComposeIterator<E1, E2, T> implements Iterator<T> {
    private final Iterator<E1> firstIterator;
    private final Iterator<E2> secondIterator;
    private final Composer<E1, E2, T> composer;

    public ComposeIterator(Iterator<E1> firstIterator,
                          Iterator<E2> secondIterator, Composer<E1, E2, T> composer) {
        if (firstIterator == null || secondIterator == null
            || composer == null)
            throw new IllegalArgumentException();
        this.firstIterator = firstIterator;
        this.secondIterator = secondIterator;
        this.composer = composer;
    }

    @Override
    public boolean hasNext() {
        return firstIterator.hasNext() && secondIterator.hasNext();
    }

    @Override
    public T next() {
        return composer.compose(firstIterator.next(), secondIterator.next());
    }
}
```

settembre 2014

4. Considerare le due interfacce generiche `Visitor` e `BinTree`.

```
public interface Visitor<E, R> {
    // E type of node labels, R type of the result of the visit

    public R visitLeaf(E e);

    public R visitBranch(E e, BinTree<E> left, BinTree<E> right);
}

public interface BinTree<E> {
    <R> R accept(Visitor<E, R> v);
}
```

- (a) Completare la definizione della classe `BinTreeFactory`.

```
public class BinTreeFactory {
    private BinTreeFactory() {
    }

    public static <T> BinTree<T> leaf(final T e) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                // DA COMPLETARE
            }
        };
    }

    public static <T> BinTree<T> branch(final T e, final BinTree<T> l, final BinTree<T> r) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                // DA COMPLETARE
            }
        };
    }
}
```

- (b) Completare la classe `Depth` che calcola la profondità di un albero (ossia, la lunghezza di uno dei cammini massimi dalla radice a una foglia).

```
public class Depth<E> implements Visitor<E, Integer> {
    @Override
    public Integer visitLeaf(E elt) {
        // DA COMPLETARE
    }

    @Override
    public Integer visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        // DA COMPLETARE
    }
}
```

- (c) Completare la classe `InOrder` che calcola la stringa (tramite un oggetto di tipo `StringBuilder`) corrispondente alla visita in-order di un albero.

```
public class InOrder<E> implements Visitor<E, StringBuilder> {
    final private StringBuilder stb = new StringBuilder();

    @Override
    public StringBuilder visitLeaf(E e) {
        // DA COMPLETARE
    }

    @Override
    public StringBuilder visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        // DA COMPLETARE
    }
}
```

a)

```
public class BinTreeFactory {
    private BinTreeFactory() {
    }

    public static <T> BinTree<T> leaf(final T e) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                return v.visitLeaf(e);
            }
        };
    }

    public static <T> BinTree<T> branch(final T e, final BinTree<T> l, final BinTree<T> r) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                return v.visitBranch(e, l, r);
            }
        };
    }
}
```

b)

```
@Override
public class Depth<E> implements Visitor<E, Integer> {
    @Override
    public Integer visitLeaf(E elt) {
        return 0;
    }

    @Override
    public Integer visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        return Math.max(left.accept(this), right.accept(this)) + 1;
    }
}
```

c)

```
@Override
public class InOrder<E> implements Visitor<E, StringBuilder> {
    final private StringBuilder stb = new StringBuilder();

    @Override
    public StringBuilder visitLeaf(E e) {
        return stb.append(e.toString());
    }

    @Override
    public StringBuilder visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        left.accept(this);
        stb.append(" ").append(e.toString()).append(" ");
        return right.accept(this);
    }
}
```

Luglio 2014

4. Completare i metodi delle classi `SetUtil` e `Test`. Il metodo `multiplyAll` deve essere definito utilizzando la classe `Prod` e il metodo `SetUtil.fold`.

```
import java.util.Set;

interface Function<X, Y> { // Functions of type X -> Y
    Y apply(X x); // applies this function to x (and returns the result of type Y)
}

interface SetFactory {
    <X> Set<X> newEmptySet(); // creates an empty set of X
}

class SetUtil {
    private final SetFactory setFactory; // use this to create empty sets
    public SetUtil(SetFactory setFactory) {
        this.setFactory = setFactory;
    }

    // returns true iff predicate p is true for all elements of s
    public <X> boolean all(Set<X> s, Function<X, Boolean> p) /* completare */

    // returns true iff predicate p is true for at least one element of s
    public <X> boolean any(Set<X> s, Function<X, Boolean> p) /* completare */

    // returns the results of applying f to all elements of s
    public <X, Y> Set<Y> map(Set<X> s, Function<X, Y> f) /* completare */

    // f is a curried function of type X -> Y -> X where the accumulator has type X
    // the method returns f (... (f initVal e_1) e_2) ... ) e_n where s = {e_1, e_2, ..., e_n}
    public <X, Y> X fold(Set<Y> s, Function<X, Function<Y, X>> f, X initVal) /* completare */

    // returns the union of inSet1 and inSet2
    public <X> Set<X> union(Set<X> inSet1, Set<X> inSet2) /* completare */

    // returns the intersection of inSet1 and inSet2
    public <X> Set<X> intersect(Set<X> inSet1, Set<X> inSet2) /* completare */
}

public class Test {

    // defines the function f such that f x y = x * y
    static final class Prod implements Function<Integer, Function<Integer, Integer>> {
        @Override
        public Function<Integer, Integer> apply(final Integer x) {
            return new Function<Integer, Integer>() {
                /* completare */
            };
        }
    }

    // returns the product of all elements of s
    // by using class Prod and method SetUtil.fold
    static Integer multiplyAll(Set<Integer> s) /* completare */
}

4. import java.util.HashSet;
/*
 */

class SetUtil {
    private final SetFactory setFactory; // use this to create empty sets
    public SetUtil(SetFactory setFactory) {
        this.setFactory = setFactory;
    }

    // returns true iff predicate p is true for all elements of s
    public <X> boolean all(Set<X> s, Function<X, Boolean> p) {
        for(X elem : s)
            if (!p.apply(elem))
                return false;
        return true;
    }

    // returns true iff predicate p is true for at least one element of s
    public <X> boolean any(Set<X> s, Function<X, Boolean> p) {
        for(X elem : s)
            if (p.apply(elem))
                return true;
        return false;
    }

    // returns the results of applying f to all elements of s
    public <X, Y> Set<Y> map(Set<X> s, Function<X, Y> f) {
        Set<Y> result = this.setFactory.newEmptySet();
        for(X elem : s)
            result.add(f.apply(elem));
        return result;
    }

    // f is a curried function of type X -> Y -> X where the accumulator has type X
    // the method returns f (... (f (f initVal e_1) e_2) ... ) e_n where s = {e_1, e_2, ..., e_n}
    public <X, Y> X fold(Set<Y> s, Function<X, Function<Y, X>> f, X initVal) {
        X result = initVal;
        for(Y elem : s)
            result = f.apply(result).apply(elem);
        return result;
    }

    // returns the union of inSet1 and inSet2
    public <X> Set<X> union(Set<X> inSet1, Set<X> inSet2) {
        Set<X> result = this.setFactory.newEmptySet();
        result.addAll(inSet1);
        result.addAll(inSet2);
        return result;
    }

    // returns the intersection of inSet1 and inSet2
    public <X> Set<X> intersect(Set<X> inSet1, Set<X> inSet2) {
        Set<X> result = this.setFactory.newEmptySet();
        for(X elem : inSet1)
            if (inSet2.contains(elem))
                result.add(elem);
        return result;
    }

    class Test {

        // defines the function f such that f x y = x * y
        static final class Prod implements Function<Integer, Function<Integer, Integer>> {
            @Override
            public Function<Integer, Integer> apply(final Integer x) {
                return new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer y) {
                        return x*y;
                    }
                };
            }
        }

        // returns the product of all elements of s
        // by using class Prod and method SetUtil.fold
        static Integer multiplyAll(Set<Integer> s) {
            SetUtil su = new SetUtil(new SetFactory() {
                @Override
                public <X> Set<X> newEmptySet() {
                    return new HashSet<X>();
                }
            });
            return su.fold(s, new Prod(), 1);
        }
    }
}
```

Settembre 2013

4. Considerare i package ast e visitor che implementano abstract syntax tree e visite su di essi per espressioni booleane formate a partire da variabili, l'operatore unario di negazione e quelli binari di congiunzione e disgiunzione logica.

```

package ast;
import visitor.Visitor;
public interface Exp {
    Iterable<Exp> getChildren();
    void accept(Visitor v);
}

-----
package ast;
import static java.util.Arrays.asList;
public abstract class AbsExp implements Exp {
    private final Iterable<Exp> children;
    protected AbsExp(Exp... children) {
        this.children = asList(children);
    }
    @Override
    public Iterable<Exp> getChildren() {
        return children;
    }
}

-----
package ast;
import visitor.Visitor;
public class BoolLit extends AbsExp {
    final private boolean value;
    public BoolLit(boolean value) {
        this.value = value;
    }
    public boolean getValue() {
        return value;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

-----
package ast;
import visitor.Visitor;
public class AndExp extends AbsExp {
    // completare
}

-----
package ast;
import visitor.Visitor;
public class OrExp extends AbsExp {
    // completare
}

-----
package ast;
import visitor.Visitor;
public class NotExp extends AbsExp {
    // completare
}

```

- (a) Completare le definizioni delle classi BoolLit, AndExp, OrExp e NotExp.
(b) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione delle classi EvalVisitor e NegationVisitor.

```

package visitor;
import ast.*;
public interface Visitor {
    void visit(BoolLit e);
    void visit(AndExp e);
    void visit(OrExp e);
    void visit(NotExp e);
}

-----
package visitor;
public abstract class AbstractVisitor<T> implements Visitor {
    protected T result;
    public T getResult() {
        return result;
    }
}

-----
package visitor;
import java.util.Iterator;
public class EvalVisitor extends AbstractVisitor<Boolean> {
    // completare
}

-----
package visitor;
import java.util.Iterator;
public class NegationVisitor extends AbstractVisitor<Exp> {
    // completare
}

```

- la classe EvalVisitor valuta l'espressione visitata.
Esempio:
- ```

Exp exp = new AndExp(new BoolLit(true), new NotExp(new OrExp(
 new BoolLit(false), new BoolLit(true))));
EvalVisitor ev = new EvalVisitor();
exp.accept(ev);
assert !ev.getResult();

```
- la classe NegationVisitor genera l'espressione corrispondente alla negazione dell'espressione visitata costruita applicando le leggi di De Morgan:

$$\begin{aligned}
\neg \text{true} &= \text{false} \\
\neg \text{false} &= \text{true} \\
\neg(e_1 \wedge e_2) &= (\neg e_1) \vee (\neg e_2) \\
\neg(e_1 \vee e_2) &= (\neg e_1) \wedge (\neg e_2) \\
\neg \neg e &= e
\end{aligned}$$

Esempio: dopo l'esecuzione del seguente frammento di codice, la variabile exp contiene l'abstract syntax tree corrispondente all'espressione  $\text{false} \vee (\text{false} \vee \text{true})$ .

```

Exp exp = new AndExp(new BoolLit(true), new NotExp(new OrExp(
 new BoolLit(false), new BoolLit(true))));
NegationVisitor nv = new NegationVisitor();
exp.accept(nv);
exp = nv.getResult();

```

```

public class BoolLit extends AbsExp {
 final private boolean value;
 public BoolLit(boolean value) {
 this.value = value;
 }
 public boolean getValue() {
 return value;
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

public class AndExp extends AbsExp {
 public AndExp(Exp exp1, Exp exp2) {
 super(exp1, exp2);
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

public class OrExp extends AbsExp {
 public OrExp(Exp exp1, Exp exp2) {
 super(exp1, exp2);
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

public class NotExp extends AbsExp {
 public NotExp(Exp exp) {
 super(exp);
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

public class EvalVisitor extends AbstractVisitor<Boolean> {
 public void visit(AndExp exp) {
 Iterator<Exp> it = exp.getChildren().iterator();
 it.next().accept(this);
 boolean res = result;
 it.next().accept(this);
 result = res && result;
 }
 public void visit(OrExp exp) {
 Iterator<Exp> it = exp.getChildren().iterator();
 it.next().accept(this);
 boolean res = result;
 it.next().accept(this);
 result = res || result;
 }
 public void visit(NotExp exp) {
 Iterator<Exp> it = exp.getChildren().iterator();
 it.next().accept(this);
 result = !result;
 }
 public void visit(BoolLit exp) {
 result = exp.getValue();
 }
}

public class NegationVisitor extends AbstractVisitor<Exp> {
 @Override
 public void visit(AndExp exp) {
 Iterator<Exp> it = exp.getChildren().iterator();
 it.next().accept(this);
 Exp res = result;
 it.next().accept(this);
 result = new OrExp(res, result);
 }
 @Override
 public void visit(OrExp exp) {
 Iterator<Exp> it = exp.getChildren().iterator();
 it.next().accept(this);
 Exp res = result;
 it.next().accept(this);
 result = new AndExp(res, result);
 }
 @Override
 public void visit(NotExp exp) {
 result = exp.getChildren().iterator().next();
 }
 @Override
 public void visit(BoolLit exp) {
 result = new BoolLit(!exp.getValue());
 }
}

```

# febbraio 2013

4. Considerare la seguente classe `ConcatLang` che implementa l'interfaccia `Language` e che rappresenta la concatenazione di due linguaggi finiti (ossia, di due insiemi finiti di stringhe).

```
import java.util.Set;
public interface Language extends Iterable<String> {
 boolean contains(String str);
 Set<String> getSet();
}
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Set;

public class ConcatLang implements Language {
 final private Set<String> first;
 final private Set<String> second;
 public ConcatLang(Set<String> first, Set<String> second) {
 // completare
 }
 public boolean contains(String str) {
 // completare
 }
 public Set<String> getSet() {
 // completare
 }
 public Iterator<String> iterator() {
 return new Iterator<String>() {
 private Iterator<String> it1 = first.iterator();
 private Iterator<String> it2 = Collections.<String>emptyIterator();
 private String firstString; // stores the last string returned by it1.next()
 public boolean hasNext() {
 // completare
 }
 public String next() {
 // completare
 }
 public void remove() {
 throw new UnsupportedOperationException();
 }
 };
 }
}

Il costruttore e i metodi implementati nella classe ConcatLang sono i seguenti:

• public ConcatLang(Set<String> first, Set<String> second): costruisce un nuovo oggetto che rappresenta la concatenazione dei due insiemi contenuti rispettivamente in first e second; solleva l'eccezione IllegalArgumentException se first o second contiene null.
• public boolean contains(String str): restituisce true se e solo se la stringa in str appartiene alla concatenazione dei due linguaggi; solleva l'eccezione IllegalArgumentException se str contiene null.
• public Set<String> getSet(): restituisce un nuovo insieme corrispondente alla concatenazione dei due linguaggi.
• public Iterator<String> iterator(): restituisce un iteratore in grado di iterare su tutte le stringhe che appartengono alla concatenazione dei due linguaggi.

A titolo di esempio, il seguente test deve avere successo.

import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public class Test {
 public static void main(String[] args) {
 Language lan = new ConcatLang(new HashSet<String>(Arrays.asList("a", "")), new HashSet<String>(Arrays.asList("b", "c")));
 assert lan.contains("a");
 assert lan.contains("ac");
 assert !lan.contains("d");
 Set<String> set = lan.getSet();
 for(String st:lan)
 assert set.contains(st);
 }
}

(a) Completare il costruttore della classe ConcatLang.
(b) Completare i metodi contains(String str) e getSet() della classe ConcatLang (suggerimento: utilizzare un iteratore).
(c) Completare il metodo hasNext() della classe anonima definita nel metodo iterator() (ricordare che la concatenazione di due insiemi è vuota nel caso in cui uno dei due insiemi sia vuoto).
(d) Completare il metodo next() della classe anonima definita nel metodo iterator(). Assumere che i due insiemi in first e second non contengano null.
```

```
public class ConcatLang implements Language {
 final private Set<String> first;
 final private Set<String> second;
 public ConcatLang(Set<String> first, Set<String> second) {
 if (first == null || second == null)
 throw new IllegalArgumentException();
 this.first = first;
 this.second = second;
 }
 public boolean contains(String str) {
 if (str == null)
 throw new IllegalArgumentException();
 for (String el : this)
 if (str.equals(el))
 return true;
 return false;
 }
 public Set<String> getSet() {
 HashSet<String> set = new HashSet<String>();
 for (String el : this)
 set.add(el);
 return set;
 }
 public Iterator<String> iterator() {
 return new Iterator<String>() {
 private Iterator<String> it1 = first.iterator();
 private Iterator<String> it2 = Collections.<String>emptyIterator();
 private String firstString; // stores the last string returned by it1.next()
 public boolean hasNext() {
 return it2.hasNext() || it1.hasNext() && !second.isEmpty();
 }
 public String next() {
 if (!it2.hasNext())
 if (it1.hasNext())
 firstString = it1.next();
 it2 = second.iterator();
 } else
 throw new NoSuchElementException();
 return firstString + it2.next();
 }
 public void remove() {
 throw new UnsupportedOperationException();
 }
 };
 }
}
```

# gennaio 2013

4. Considerare i package `ast` e `visitor` per rappresentare tramite abstract syntax tree espressioni e implementare visite sui corrispondenti abstract syntax tree; le espressioni considerate sono costruite a partire da:

- literal di tipo `List<Integer>`;
- identificatori di tipo `List<Integer>`;
- un operatore unario che presa una lista `l` restituisce una nuova lista ottenuta rovesciando `l`;
- un operatore binario che prese due liste `l1` ed `l2` restituisce una nuova lista ottenuta concatenando `l1` ed `l2`.

```
package visitor;
import ast.*;
public interface Visitor {
 void visit(AppendExp e);
 void visit(IdentExp e);
 void visit(ReverseExp e);
 void visit(ListLit e);
}

package visitor;
public abstract class AbstractVisitor<T> implements Visitor {
 protected T result;
 public T getResult() {
 return result;
 }
}

package ast;
import visitor.Visitor;
public interface Exp {
 Exp[] getChildren();
 void accept(Visitor v);
}

package ast;
public abstract class AbsExp implements Exp {
 private final Exp[] children;
 protected AbsExp(Exp... children) {
 this.children = children;
 }
 @Override
 public Exp[] getChildren() {
 return children;
 }
}

package ast;
public interface Ident extends Exp {
 String getName();
}

package ast;
import visitor.Visitor;
public class IdentExp extends AbsExp implements Ident {
 private final String name;
 public IdentExp(String name) {
 this.name = name;
 }
 public String getName() {
 return name;
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

package ast;
import java.util.List;
import visitor.Visitor;
public class ListLit extends AbsExp {
 protected final List<Integer> value; /* completare */
 public ListLit(List<Integer> value) { /* completare */ }
 public List<Integer> getValue() { /* completare */ }
 public void accept(Visitor v) { /* completare */ }
}

package ast;
import visitor.Visitor;
public class ReverseExp extends AbsExp {
 public ReverseExp(Exp exp) { /* completare */ }
 public void accept(Visitor v) { /* completare */ }
}

package ast;
import visitor.Visitor;
public class AppendExp extends AbsExp {
 public AppendExp(Exp leftExp, Exp rightExp) { /* completare */ }
 public void accept(Visitor v) { /* completare */ }
}
```

(a) Completare le definizioni delle classi `ListLit`, `ReverseExp` e `AppendExp`.

(b) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione di `EvaluationVisitor` che permette di valutare un'espressione a partire da un ambiente dinamico `DynamicEnv`. La visita deve sollevare un'eccezione di tipo `RuntimeException` se l'espressione contiene una variabile non definita nell'ambiente dinamico.

Il metodo `read` di `DynamicEnv` solleva un'eccezione di tipo `RuntimeException` se l'identificatore non è definito.

```
package visitor;
import java.util.List;
import ast.Ident;
public interface DynamicEnv {
 List<Integer> read(Ident id);
}

package visitor;
import java.util.List;
import java.util.ArrayList;
import static java.util.Collections.reverse;
import test.*;
public class EvaluationVisitor extends AbstractVisitor<List<Integer>> {
 // completare
}
```

(c) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione di `SubstitutionVisitor` che permette di produrre una nuova espressione ottenuta a partire da quella visitata applicando una sostituzione. Una sostituzione è una funzione da identificatori a espressioni.

Il metodo `read` restituisce sempre una nuova copia dell'espressione associata all'identificatore `id`.

```
package visitor;
import test.*;
public interface Substitution {
 Exp readIdent(Idt id);
}

package visitor;
import ast.*;
public class SubstitutionVisitor extends AbstractVisitor<Exp> {
 // completare
}
```

Per esempio, se la sostituzione associa all'identificatore `"x"` l'espressione `new ListLit(Arrays.asList(5,6,7))`, allora la visita dell'albero corrispondente all'espressione

```
new ReverseExp(new AppendExp(new IdentExp("x"), new IdentExp("x")))
```

restituisce un nuovo albero corrispondente all'espressione

```
new ReverseExp(new AppendExp(new ListLit(Arrays.asList(5,6,7)), new ListLit(Arrays.asList(5,6,7))))
```

```
public class ListLit extends AbsExp {
 protected final List<Integer> value;
 public ListLit(List<Integer> value) {
 this.value = value;
 }
 public List<Integer> getValue() {
 return value;
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

public class ReverseExp extends AbsExp {
 public ReverseExp(Exp exp) {
 super(exp);
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

public class AppendExp extends AbsExp {
 public AppendExp(Exp leftExp, Exp rightExp) {
 super(leftExp, rightExp);
 }
 public void accept(Visitor v) {
 v.visit(this);
 }
}

public class EvaluationVisitor extends AbstractVisitor<List<Integer>> {
 private final DynamicEnv env;
 public EvaluationVisitor(DynamicEnv env) {
 this.env = env;
 }
 public void visit(AppendExp e) {
 Exp[] exps = e.getChildren();
 exps[0].accept(this);
 List<Integer> prevRes = new ArrayList<>(result);
 exps[1].accept(this);
 prevRes.addAll(result);
 result = prevRes;
 }
 public void visit(IdentExp id) {
 result = env.read(id);
 }
 public void visit(ReverseExp e) {
 e.getChildren()[0].accept(this);
 result = new ArrayList<>(result);
 reverse(result);
 }
 public void visit(ListLit e) {
 result = e.getValue();
 }
}

public class SubstitutionVisitor extends AbstractVisitor<Exp> {
 private final Substitution subs;
 public SubstitutionVisitor(Substitution subs) {
 this.subs = subs;
 }
 public void visit(AppendExp e) {
 Exp[] exps = e.getChildren();
 exps[0].accept(this);
 Exp prevRes = result;
 exps[1].accept(this);
 result = new AppendExp(prevRes, result);
 }
 public void visit(IdentExp id) {
 result = subs.read(id);
 }
 public void visit(ReverseExp e) {
 e.getChildren()[0].accept(this);
 result = new ReverseExp(result);
 }
 public void visit(ListLit e) {
 result = new ListLit(e.getValue());
 }
}
```

# agosto 2012

3. Considerare le seguenti interfacce e classi Java:

```
public interface IPredicate<T1, T2> {
 boolean isTrueOn(T1 el1, T2 el2);
}

public interface IPair<T1, T2> {
 public T1 getFst();
 public T2 getSnd();
}

import java.util.Iterator;
public interface IBinaryRelation<T1, T2> {
 public Iterator<T1> iterator();
 public Iterator<Pair<T1, T2>> filteredIterator(IPredicate<T1, T2> pred);
}

import java.util.*;
public class BinaryRelation<T1, T2> implements IBinaryRelation<T1, T2> {
 final private List<Pair<T1, T2>> pairs;
 public BinaryRelation() {
 pairs = new LinkedList<Pair<T1, T2>>();
 }
 public BinaryRelation(Iterator<Pair<T1, T2>> extIt) {
 this();
 ListIterator<Pair<T1, T2>> init = this.pairs.listIterator();
 while (extIt.hasNext())
 init.add(extIt.next());
 }
 @Override
 public Iterator<T1> iterator() {
 return new Iterator<T1>() {
 Iterator<Pair<T1, T2>> it = pairs.iterator();
 @Override
 public boolean hasNext() {
 // completare
 }
 @Override
 public T1 next() {
 // completare
 }
 @Override
 public void remove() {
 throw new UnsupportedOperationException();
 }
 };
 }
 @Override
 public Iterator<Pair<T1, T2>> filteredIterator(final IPredicate<T1, T2> pred) {
 return new Iterator<Pair<T1, T2>>() {
 ListIterator<Pair<T1, T2>> it = pairs.listIterator();
 // instance initializer executed just after
 // the initialization of the iterator
 goToNext();
 @Override
 public void goToNext() {
 // that satisfies the predicate
 while (it.hasNext()) {
 Pair<T1, T2> curPair = it.next();
 if (pred.isTrueOn(curPair.getFst(), curPair.getSnd())) {
 it.previous();
 it.remove();
 }
 }
 }
 @Override
 public boolean hasNext() {
 // completare
 }
 @Override
 public Pair<T1, T2> next() {
 // completare
 }
 @Override
 public void remove() {
 throw new UnsupportedOperationException();
 }
 };
 }
}
```

La classe generica `BinaryRelation` implementa una relazione binaria come lista di copie.

Il metodo `iterator` crea un iteratore che restituisce progressivamente la prima componente di tutte le coppie contenute nella relazione.

Il metodo `filteredIterator` crea un iteratore che restituisce progressivamente tutte le coppie contenute nella relazione che soddisfano il predicato che viene passato come argomento.

(a) Completare la seguente classe.

```
public final class Pair<T1, T2> implements IPair<T1, T2> {
 // completare
}
```

# luglio 2012

3. Considerate le seguenti interfacce e classi:

- `Tree<E>`: alberi non vuoti i cui nodi possono avere un numero arbitrario di figli e sono etichettati con valori di tipo `E`.
- `Node<E>`: nodi dell'albero etichettati con valori di tipo `E`.
- `TreeClass<E>`: implementazione di `Tree<E>`.

```
import java.util.Stack;

public interface Tree<E> {
 boolean contains(E elem);
 E get(Stack<Integer> path);
 E set(Stack<Integer> path, E elem);
}

import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

public class TreeClass<E> implements Tree<E> {
 private Node<E> root;

 private static class Node<E> {
 private E elem;
 private List<Node<E>> children;
 private Node(E elem, List<Node<E>> children) {
 this.elem = elem;
 this.children = children;
 }
 private Node(E elem) {
 this(elem, new ArrayList<Node<E>>());
 }
 private boolean contains(E elem) {
 // completare
 }
 }

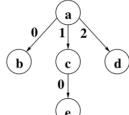
 public TreeClass(E elem) {
 // completare
 }

 @Override
 public E get(Stack<Integer> path) {
 // completare
 }

 @Override
 public E set(Stack<Integer> path, E elem) {
 // completare
 }

 @Override
 public boolean contains(E elem) {
 // completare
 }
}
```

Nei metodi `get` e `set` ogni nodo viene individuato con il cammino (rappresentato da uno stack) dalla radice al nodo stesso, con la convenzione che il primo nodo nella lista `children` corrisponde all'indice 0. Per esempio, nell'albero



i nodi **a**, **b**, **c**, **d**, **e** sono rispettivamente individuati dai path `[]`, `[0]`, `[1]`, `[2]` e `[1, 0]`.

- Completare la definizione del costruttore `TreeClass(E elem)` che crea un albero con la sola radice, etichettata da `elem`.
- Completare la definizione del metodo `boolean contains(E elem)` della classe `Node`, che restituisce `true` se e solo se l'albero la cui radice coincide con il nodo `this` contiene un nodo etichettato con `elem`.
- Completare la definizione del metodo `boolean contains(E elem)` della classe `TreeClass`, che restituisce `true` se e solo se l'albero `this` contiene un nodo etichettato con `elem`.
- Completare la definizione del metodo `E get(Stack<Integer> path)` che restituisce l'etichetta del nodo dell'albero `this` individuato dal cammino path.
- Completare la definizione del metodo `E set(Stack<Integer> path, E elem)` che associa all'etichetta del nodo dell'albero `this` individuato dal cammino path la nuova etichetta `elem`; il metodo restituisce il valore precedente dell'etichetta del nodo.

(b) Completare il seguente frammento di codice che crea un predicato che è soddisfatto se e solo se il secondo argomento non è minore di 1968.

```
IPredicate<String, Integer> pred1 = new IPredicate<String, Integer>() {
 // completare
};

(c) Completare la definizione dei metodi hasNext e next della classe anonima dichiarata nel metodo iterator1.
```

(d) Completare la definizione dei metodi `hasNext` e `next` della classe anonima dichiarata nel metodo `filteredIterator`.

(a) **public final class** `Pair<T1, T2>` **implements** `IPair<T1, T2>` {

```
 private final T1 fst;
 private final T2 snd;
```

```
 public Pair(T1 fst, T2 snd) {
 this.fst = fst;
 this.snd = snd;
 }
```

```
@Override
```

```
public T1 getFst() {
 return fst;
}
```

```
@Override
```

```
public T2 getSnd() {
 return snd;
}
```

(b) **IPredicate<String, Integer>** `pred1` = **new** `IPredicate<String, Integer>()` {

```
 @Override
 public boolean isTrueOn(String el1, Integer el2) {
 return el2 >= 1968;
 }
};
```

(c) **@Override**

```
public boolean hasNext() {
 return it.hasNext();
}
```

```
@Override
public T1 next() {
 return it.next().getFst();
}
```

(d) **public boolean** `hasNext()` {

```
 return it.hasNext();
}
```

```
@Override
public Pair<T1, T2> next() {
 Pair<T1, T2> res = it.next();
 goToNext();
 return res;
}
```

(c) **@Override**

```
public boolean contains(E elem) {
 return root.contains(elem);
}
```

(d) **@Override**

```
public E get(Stack<Integer> index) {
 Node<E> node = root;
 while (!index.isEmpty())
 node = node.children.get(index.pop());
 return node.elem;
}
```

(e) **@Override**

```
public E set(Stack<Integer> index, E elem) {
 Node<E> node = root;
 while (!index.isEmpty())
 node = node.children.get(index.pop());
 E oldElem = node.elem;
 node.elem = elem;
 return oldElem;
}
```