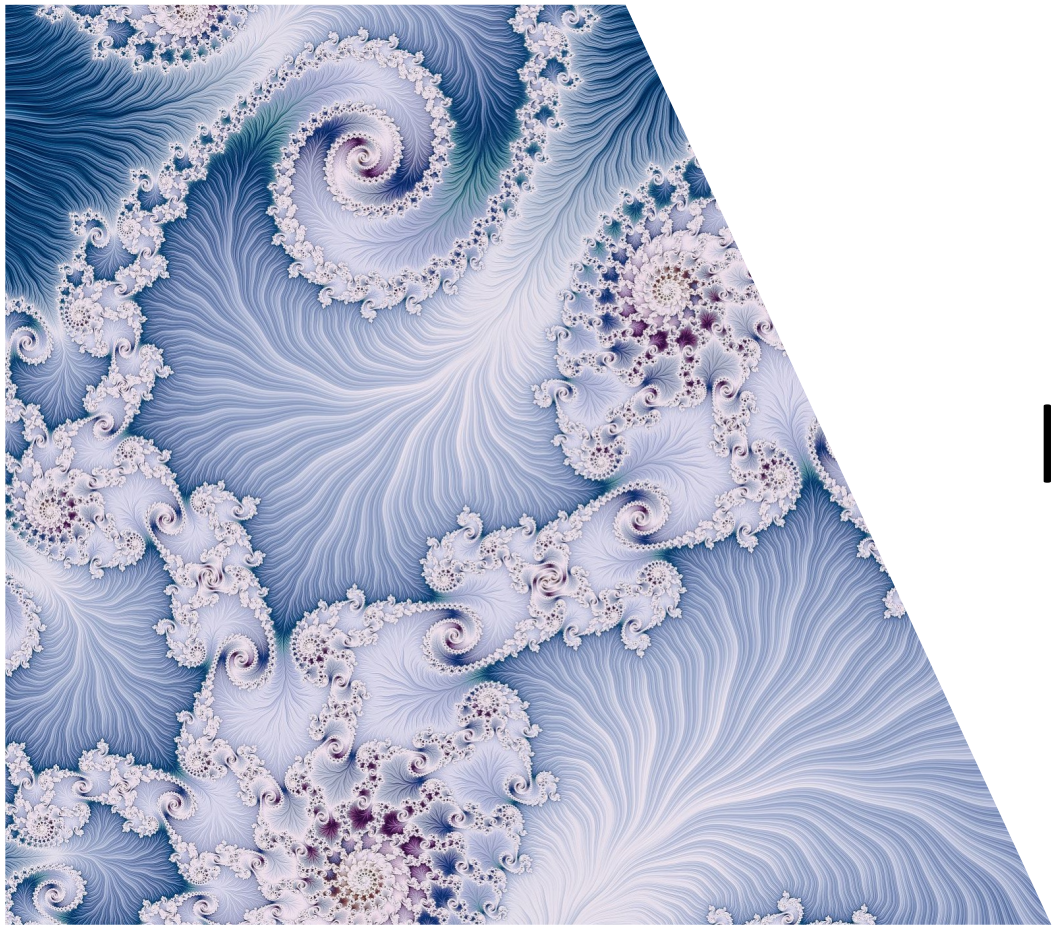


Iterators

MBF

Motivation

We often want to access every item in a collection of items

- **We call this traversing or iterating**

```
List<String> list = List.of("a", "b", "c", "d", "e", "f", "g", "h",
"i", "j");
System.out.println("--------");
for (String s : list){
    System.out.println(s);
}
```

Easy because these objects are java.util.**Collection**…

Collections are **Iterable**

```
public interface Collection<E> extends Iterable<E> {
```

What if we want to traverse an arbitrary collection of objects?

Its underlying implementation may not be known to us

https://www.cs.colostate.edu/~cs161/Fall14/slides/12_iterators.pdf

Motivation

- Rendre une structure « itérable » sans dévoiler sa structure interne

- Par exemples
  - itérer sur les questions d'un Quiz
  - Itérer sur les valeurs contenues dans un arbre
  - Itérer sur les produits placés dans un panier

```
for (Question q : quiz) {
   response = UI. ask(q);
   ….
}
```

```
for (Product product : shoppingCart) {
```

1) Make your "class" iterable

Iterable ?

```
public interface Iterable<T> {

    Iterator<T> iterator();


    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
...
```

You only have to return an Iterator

Quiz Example : Iterate on questions

```
public class Quiz implements Iterable<Question>{
   List<Question> questions;

   public Quiz(List<Question> questions) {
      this.questions = questions;
   }
   public Quiz() {
      this.questions = new ArrayList<>();
   }

    @Override
    public Iterator<Question> iterator() {
       return questions.iterator();
    }
```

```
public static void main(String args[]){
   Quiz quiz = new Quiz();
   quiz.questions.add(new Question("Who was the first president of the United States?", new String[]{"George Washington","Thomas Jefferson","Abraham Lincoln","John Adams"}, 0, "history"));
   quiz.questions.add(new Question("What is the capital of France?", new String[]{"Paris", "Lyon", "Marseille", "Toulouse"}, 0, "Geography"));


for (Question q : quiz) {
   System.out.println(q.question);
}
```

2) Define an Iterator

The Java java.util.Iterator Interface

```
public interface Iterator<E> {

    boolean hasNext(); //returns true if there are more elements to iterate over

    E next(); //returns the next element
                    throws a NoSuchElement exception if a next element does not exist


default void remove() { //removes the last element returned by the iterator
      throw new          UnsupportedOperationException("remove");
   }

   default void forEachRemaining(Consumer<? super E> action) {
      Objects.requireNonNull(action);
      while (hasNext())
         action.accept(next());
   }
}
```

Number Example : innerclass and iterator

```
class Numbers {
   private static final List<Integer> NUMBER_LIST =
                        Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

   public static Iterator<Integer> primeIterator() {
      return new PrimeIterator();  }

   private static class PrimeIterator
                        implements Iterator<Integer> {
      private int cursor;

      @Override
      public Integer next() {
         exist(cursor);
         return NUMBER_LIST.get(cursor++); }

      private void exist(int current) {
         if (current >= NUMBER_LIST.size()) {
            throw new NoSuchElementException();}
      }

      @Override
      public boolean hasNext() {
         if (cursor > NUMBER_LIST.size()) {
            return false;
         }

         for (int i = cursor; i < NUMBER_LIST.size(); i++) {
            if (isPrime(NUMBER_LIST.get(i))) {
               cursor = i;
               return true; }
         }
         return false;
      }

      private boolean isPrime(int number) {
         for (int i = 2; i <= number / 2; ++i) {
            if (number % i == 0) {
               return false; }
         }
         return true;
      }
   }
```

Iterator

hasNext is supposed to be called before

required

Curson is on the next prime Number.

NoSuchElementException

Any implementation of the *next()* method should throw a *NoSuchElementException* exception when there are no more elements left. Otherwise, the iteration can cause unexpected behavior

https://www.baeldung.com/java-iterator-vs-iterable

3) Use an Iterator

Iterating through an ArrayList with an explicit iterator

```
List<String> list = List.of("a", "b", "c", "d", "e", "f", "g", "h", "i", "j");
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
    String s = itr.next();
    System.out.println(s);
}
```

Advantage : the code will work even if we decide to store the data in a different data structure (as long as it provides an iterator)

https://www.cs.colostate.edu/~cs161/Fall14/slides/12_iterators.pdf

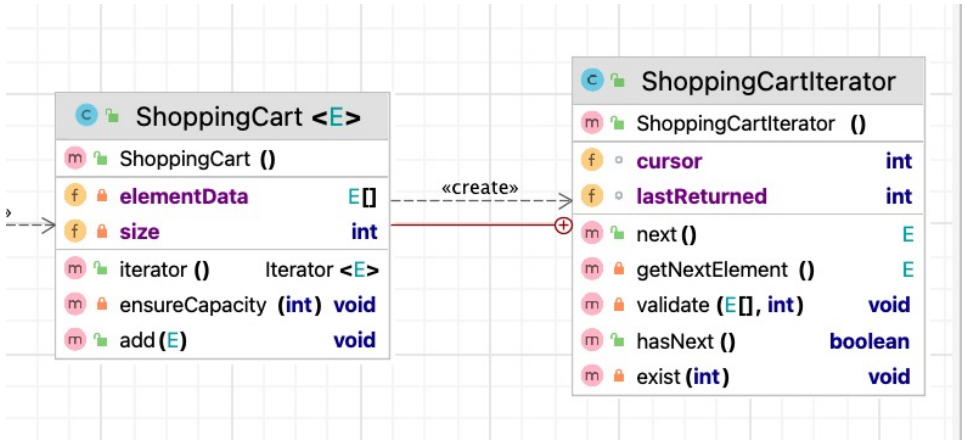
On Numbers

```
Iterator<Integer> iteratorOnPrimeNumbers = Numbers.primeIterator();

iteratorOnPrimeNumbers.forEachRemaining(System.out::println);


iteratorOnPrimeNumbers = Numbers.primeIterator();
while (iteratorOnPrimeNumbers.hasNext()) {
   System.out.println(iteratorOnPrimeNumbers.next());
}
```

4) Other examples

Shopping example

Shopping Example : genericity, innerclass and iterator




```
public class ShoppingCart<E> implements Iterable<E> {

    private E[] elementData;
    private int size;

    public ShoppingCart() {
        this.elementData = (E[]) new Object[]{};
    }

    public void add(E element) {
        ensureCapacity(size + 1);
        elementData[size++] = element;
    }

    @Override
    public Iterator<E> iterator() {
        return new ShoppingCartIterator();
    }
```

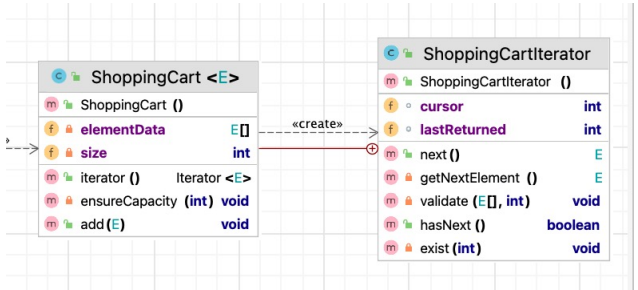
number of products in the Cart

//code : https://github.com/eugenp/tutorials/blob/master/core-java-modules/core-java-collections-2/src/main/java/com/baeldung/collections/iterable/ShoppingCart.java

Shopping Example : genericity, innerclass and iterator

Inner class : see variable of ShoppingCart

```
public class ShoppingCartIterator implements Iterator<E> {
    int cursor;
    int lastReturned = -1;

    private void exist(int current) {
        if (current >= size) {
            throw new NoSuchElementException();
        }
    }
    private void validate(E[] elements, int current) {
        if (current >= elements.length) {
            throw new ConcurrentModificationException();
        }
    }
```




```
    public boolean hasNext() {
        return cursor != size;
    }

    public E next() {
        return getNextElement();
    }

    private E getNextElement() {
        int current = cursor;
        exist(current);

        E[] elements = ShoppingCart.this.elementData;
        validate(elements, current);

        cursor = current + 1;
        lastReturned = current;
        return elements[lastReturned];
    }
}
```

hasNext…

Get the next Product and move the cursor

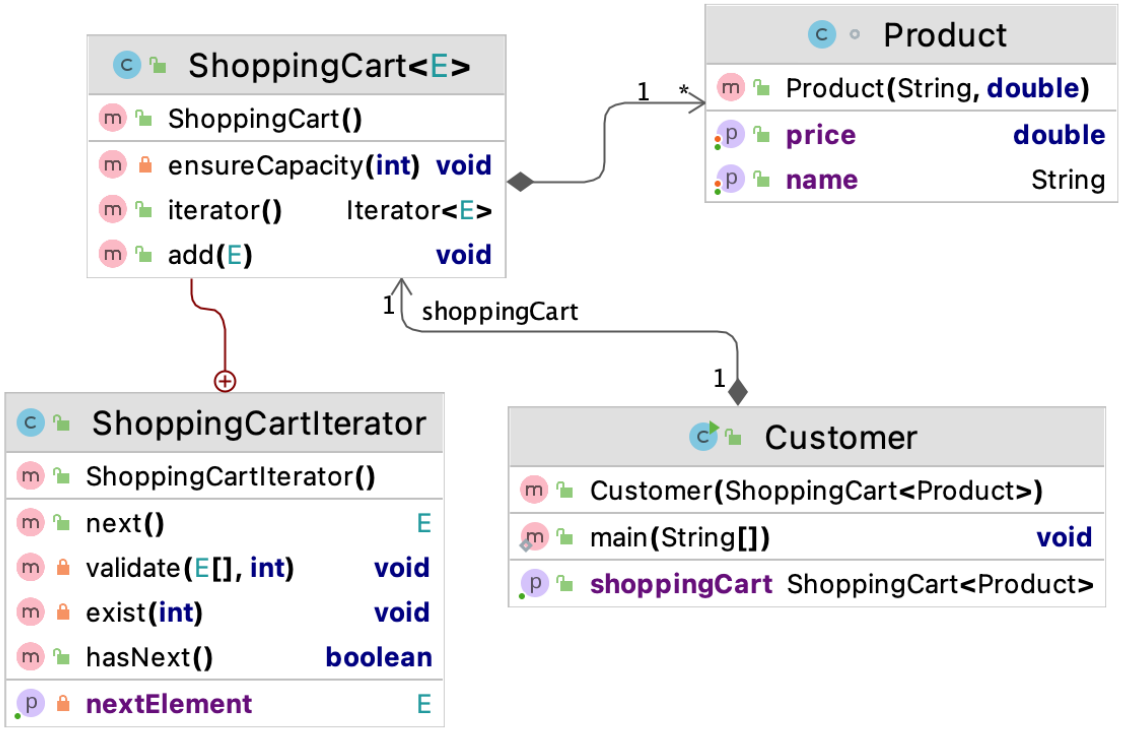
Check again on length

//code : https://github.com/eugenp/tutorials/blob/master/core-java-modules/core-java-collections-2/src/main/java/com/baeldung/collections/iterable/ShoppingCart.java

Shopping Example : usage

This diagram is adapted to present the Client class.
IntelliJ is not a good support in the case of genericity.

```
public class Customer {

  private ShoppingCart<Product> shoppingCart;
  public ShoppingCart<Product> getShoppingCart() {
     return shoppingCart;
  }

  public Customer(ShoppingCart<Product> shoppingCart
     this.shoppingCart = shoppingCart;
  }
```

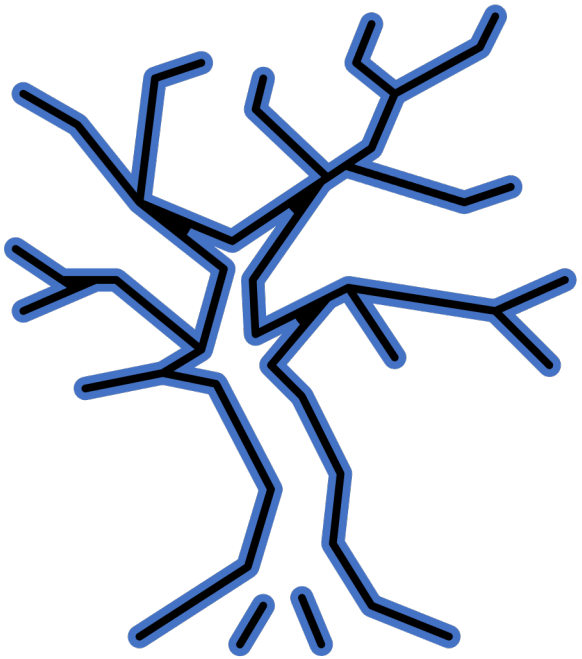


//code : https://github.com/eugenp/tutorials/blob/master/core-java-modules/core-java-collections-2/src/main/java/com/baeldung/collections/iterable/ShoppingCart.java

Shopping Example : usage

```
Tuna
Eggplant
Salad
Banana

---- Mixing products : 2 iterators ----
---- Mix : Tuna
Tuna Tuna
Tuna Eggplant
Tuna Salad
Tuna Banana
---- Mix : Eggplant
Eggplant Tuna
Eggplant Eggplant
Eggplant Salad
Eggplant Banana
---- Mix : Salad
Salad Tuna
Salad Eggplant
Salad Salad
Salad Banana
---- Mix : Banana
Banana Tuna
Banana Eggplant
Banana Salad
Banana Banana
```

```java
public static void main(String[] args) {

   Customer customer = new Customer(new ShoppingCart<>());
   ShoppingCart<Product> shoppingCart = customer.getShoppingCart();
   shoppingCart.add(new Product("Tuna", 42));
   shoppingCart.add(new Product("Eggplant", 65));
   shoppingCart.add(new Product("Salad", 45));
   shoppingCart.add(new Product("Banana", 29));

   shoppingCart.forEach(
         (product) -> System.out.println(product.getName()));

   System.out.println("---- Mixing products : 2 iterators ----");
   for (Product product : shoppingCart) {
      System.out.println("---- Mix : " + product.getName());
      for (Product product2 : shoppingCart){
        System.out.println(product.getName() + " "+  product2.getName());
   }
```



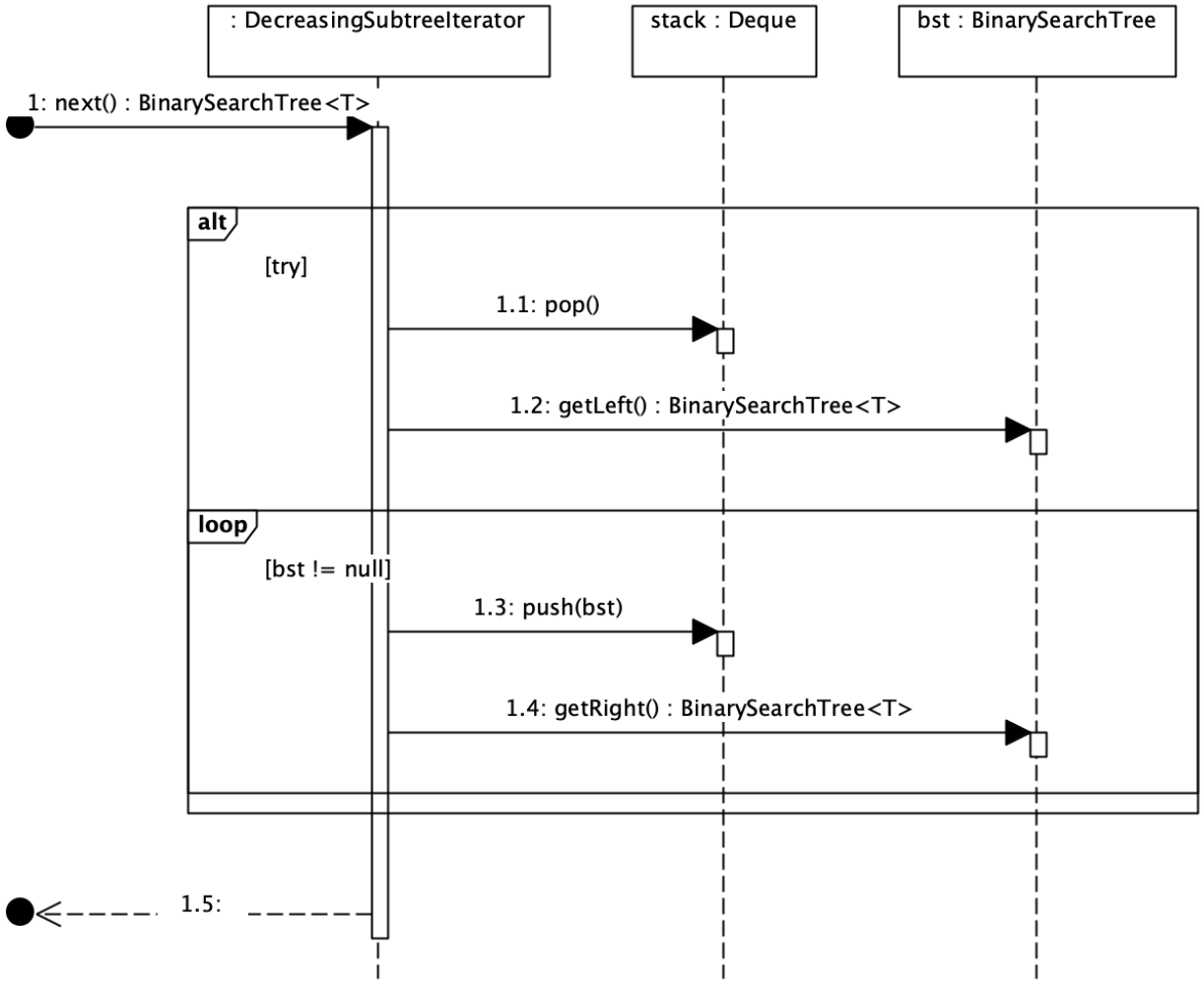
Binary Search Tree example

Iterating on Binary Search Trees

```
public class DecreasingSubtreeIterator<T extends Comparable<? super T>>
                                implements Iterator<BinarySearchTree<T>> {

  Deque<BinarySearchTree<T>> stack;

/**
   * Build an iterator over the binary node n.
   * The elements are enumerated in decreasing order.
   * Only the right subtree of the root is stored in the stack
   *
   */
  public DecreasingSubtreeIterator (BinarySearchTree<T> bst) {
     stack = new ArrayDeque<>();
     while (bst != null) {
        stack.push(bst);
        bst = bst.getRight();
     }
  }
```

```
@Override
public boolean hasNext() {
   return  !stack.isEmpty();
}

@Override
public BinarySearchTree<T> next() {
   try {
   BinarySearchTree<T> bst = stack.pop();
   BinarySearchTree<T> valuetoReturn = bst;
   bst = bst.getLeft();
   while (bst != null) {
      stack.push(bst);
      bst = bst.getRight();
   }
   return valuetoReturn;
} catch (EmptyStackException e) {
   throw new NoSuchElementException(e);
}
}
```

Iterating on Binary Search Trees : **next** as a sequence diagram



Iterating on Binary Search Trees

```
BinarySearchTree<Integer> bst = new BinarySearchTree<>(5);
bst.insert(3);
bst.insert(7);
….

DecreasingSubtreeIterator<Integer> itr = new DecreasingSubtreeIterator<>(bst);

while (itr.hasNext()) {
   BinarySearchTree<Integer> tree = itr.next();
   System.out.println(tree.getElement() + " : " + tree.getSize());
}


itr = new DecreasingSubtreeIterator<>(bst);
itr.forEachRemaining((BinarySearchTree<Integer> t) -> System.out.println(t.getElement()) );
}
```

```
80 : 1
79 : 2
78 : 3
77 : 4
…
9 : 72
8 : 73
7 : 75
6 : 1
5 : 80
4 : 1
3 : 4
2 : 2
1 : 1


80
79
78
…
3
2
1
```

```
public class BinarySearchTree<T>
        implements Iterable<T> {
```

We want to be able to iterate on the elements present in the tree.

public class BinarySearchTree<T> implements Iterable<T> {

BinarySearchTree must implement : iterator()

```
**
   * Return an iterator over the elements of the tree.
   * The elements are enumerated in increasing order.
   */
public Iterator<T> iterator() {
  return new BSTiterator(root);
}
```

We choose to implement an iterator; no structure (no list, no array) gives us the necessary iterator.

```
public class BinarySearchTree<T>
      implements Iterable<T> {
```

A specific iterator : inner class BSTiterator

```
   private class BSTiterator implements Iterator<T> {

      public T next() { …

      public boolean hasNext() {
         return !stack.isEmpty();
      }
```

It is only accessible from outside as an "Iterator".

```
public class BinarySearchTree<T>
      implements Iterable<T> {
```

Usage

```
BinarySearchTree<Integer>  bst = new BinarySearchTree<>(..);

public void iterateTree() {
        for ( Integer n : bst )
        //do something with n
 }
```

Iterable	Iterator
Represents a collection that can be iterated over using a *for*-each loop	Represents an interface that can be used to iterate over a collection
When implementing an *Iterable*, we need to override the *iterator()* method	When implementing an *Iterator*, we need to override the *hasNext()* and *next()* methods
Doesn't store the iteration state	Stores the iteration state
Removing elements during the iteration isn't allowed	Removing elements during the iteration is allowed

https://www.baeldung.com/java-iterator-vs-iterable