Term Project – StackOfInts Version 0.4

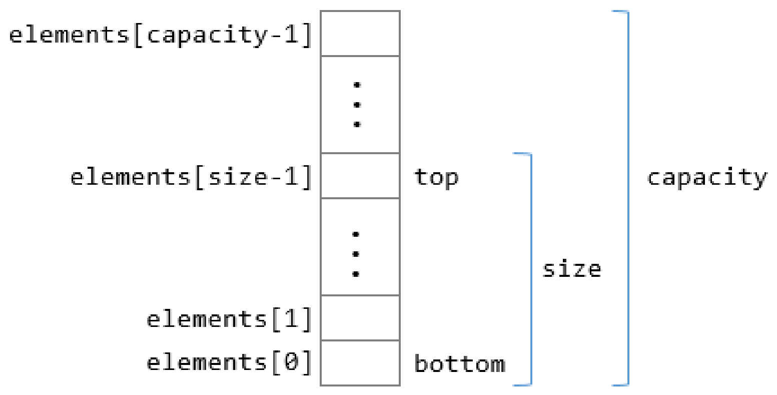
Contents

Stack

The stack is a data structure that holds data in a last-in, first-out(LIFO) fashion. It has many applications. For example, the compiler uses a stack to process method invocation. When a method is invoked, its parameters and local variables are pushed into a stack. When a method finishes its work and returns to its caller, its associated space is released from the stack. The data access is allowed only at the top of the stack, so the operations are limited. Therefore, it supports the following operations:

- push: add an element to the stack

- pop: remove the top element from the stack
- peek: get the top element without removing it
- clear: make the stack empty
- empty: true if the stack is empty
- size: the number of elements in the stack
- capacity: the capacity of the stack



[FYI: For your information] When you want to make a new class BB from an existing class AA, you may Copy and Paste menu in Package Explorer in Eclipse.

1. Go to the Package Explorer.
2. Do a right button click on **AA.java** and select '**copy**'.
3. Do a right button click on the Package Explorer and select '**paste**'.
   Then you see a prompt to enter a new class name. **Enter BB**.

Warming-up: Java Stack class

Java already provides a built-in stack. Run the following statements which use the Java Stack class and answer the following questions.

```
// Warming-up program for java.util.Stack
public class StackDriver {
    public static void main(String[] args) {
       java.util.Stack<Integer> s = new java.util.Stack<>();
       // StackOfInts s = new StackOfInts();
       System.out.println("size:" + s.size() + "  capacity:" + s.capacity());
       s.push(7);
       s.push(4);
       s.push(s.peek() + 2);
       s.pop();
       s.push(s.peek());

       System.out.println("peek:" + s.peek());
       System.out.println("size:" + s.size() + "  capacity:" + s.capacity());
       System.out.println("stack:" + s);

       if (!s.empty()) s.clear();
    }
}
```

1. What is the output?
2. What are contents of the stack after operations?
3. What is the error message when you try to pop() when the stack is empty()?
4. Add elements repeatedly and check how the capacity changes. Describe your findings.
5. In this time, add elements by 100,000 and check the capacity first. Then remove all elements by using clear() and check the capacity. Describe your findings.
6. How can you reduce the capacity when necessary?
7. What is the default stack capacity in Java to begin with? Can it be zero to begin with?

In this project we would like to implement our own stack that emulate a sort of Java stack class. Let's begin with a simple stack of ints.

Part I: Creating StackOfInts class

Let's define a class to model stacks. For simplicity, assume the stack holds the int values. So name the stack class StackOfInts.

Step 1: Stack of Ints

The UML diagram and the **brief** implementation for the StackOfInts class is shown below:

StackOfInts
-elements: int[] -size: int
+StackOfInts() +StackOfInts(capacity: int) +empty(): boolean +peek(): int +push(value: int): void +pop(): int +size(): int +capacity(): int +clear(): int

Once the instance variables are declared in a class, it is very common to provide a set of getters (or accessor methods) and setters (or mutator methods). In this particular case, pop() and push() operations do their roles.

The methods size() and capacity() return the number of elements in the stack and the capacity of the stack, respectively. Notice that they are using size() and capacity() instead of getSize() and getCapacity(). Why? There is no solid rules, however, my reason is "since they don't have the corresponding setters and they are automatically set."

The StackOfInts class encapsulates the stack storage and provides the operations for manipulating the stack.

```
public class StackOfInts {
    private int[] elements;
    private int size;
    private static final int DEFAULT_CAPACITY = 10;

    /** Construct a stack with the default capacity */
    public StackOfInts() {
      this(DEFAULT_CAPACITY);
    }
    /** Construct a stack with the specified capacity */
    public StackOfInts(int capacity) {
      elements = new int[capacity];
    }

    /** Push a new integer into the top of the stack */
    public void push(int item) {
      // Your code here
    }

    /** Return and remove the top element from the stack */
    public int pop() {
      return elements[--size];
    }

    /** Return the top element from the stack */
    public int peek() {
      return elements[size() - 1];
    }

    /** Return true if the stack is empty */
    public boolean empty() {
      return size == 0;
    }

    /** Return the number of elements in the stack */
    public int size() {
      return size;
    }

   /** Return the capacity of the stack */
    public int capacity() {
      return elements.length;
    }

    /** Make the stack empty */
    // Your code here
}
```

Basically, the warming-up program with a minor modification should be work with this StackOfInts class defined as above.

Checkpoint: After you implement StackOfInts including some missing methods, run it while replacing the first line of the warming-up program, StackDriver.java, as shown below:

```
public class StackDriver {
    public static void main(String[] args) {
      // java.util.Stack<Integer> s = new java.util.Stack<>();
      StackOfInts s = new StackOfInts();
      System.out.println("size:" + s.size() + "  capacity:" + s.capacity());. . .
. . .
```

Your result should be the same as one from java.util.Stack() used in Warming-up program.

Step 2: Increasing capacity automatically, trimToSize()

Currently, StackOfInts and java.util.Stack have a capacity of DEFAULT_CAPACITY = 10. If the stack is full (i.e., size >= capacity), create a new array of **twice** the current capacity, copy the contents of the current array to the new array, and assign the reference of the new array to the current array in the stack. Somehow both the size and the capacity becomes zero, you must treat it specially. This algorithm should be coded in push() method.

Besides you complete the push() method, you must implement another method to set the capacity according to its size. **Don't name** it such as setCapacity(). There is a better name for it. You may refer one that is already defined in java.util.Stack. For this work, do Step 3 first and come back to this point.

```
      /** Push a new element into the top of the stack */
      public void push(int item) {
      // If the stack is full or size >= capacity, double the stack capacity.


         elements[size++] = item;
      }
```

Checkpoints: When you use java.util.Stack, its default capacity to begin with is 10. When the stack becomes full and the next push (or at 11th push), the capacity becomes 20, at 21st push it becomes 40. However if you begin with a stack with size = 0 and capacity = 0, the capacity increases 1, 2, 4, 8, … 16, 32…. Make sure that your implementation satisfy these two cases. Use a method in java.util.Stack to set the capacity = 0.

You may use the following code, StackDriver2.java, to test your implementation. Run the StackDriver2 with two different cases by alternating the first two lines as needed.

```
 public class StackDriver2 {
     public static void main(String[] args) {
        java.util.Stack<Integer> s = new java.util.Stack<>();
        // StackOfInts s = new StackOfInts();
        System.out.println("DEFAULT size:" + s.size() + "  capacity:" + s.capacity());

        // make the stack full
        for (int i = 0, n = s.capacity(); i < n; i++) s.push(i);

        // add one more element to the stack, the capacity must be doubled.
        s.push(99);
        System.out.println("FULL+1 size:" + s.size() + "  capacity:" + s.capacity());

        // clear the stack and set the capacity to its size.
        s.clear();
        s.trimToSize();
        System.out.println("AFTER  clear() & trimTosize()");
        System.out.println("       size:" + s.size() + "  capacity:" + s.capacity());

        // push elements one by one and observe how the capacity changes.
        for (int i = 0; i < 9; i++) {
            s.push(i);
            System.out.println("size:" + s.size() + "  capacity:" + s.capacity());
        }
     }
 }
```

Checkpoint: The expected output is shown below. This output should be the same as you **alternate** the first two lines of code in the program StackDriver2.java.

Checkpoint: Mark your grade: Pass or Fail.

```
DEFAULT size:0  capacity:10
FULL+1 size:11  capacity:20
AFTER  clear() & trimTosize()
       size:0  capacity:0
size:1  capacity:1
size:2  capacity:2
size:3  capacity:4
size:4  capacity:4
size:5  capacity:8
size:6  capacity:8
size:7  capacity:8
size:8  capacity:8
size:9  capacity:16
```

Step 3: Implementing trimToSize()

As you noticed previously, there exists a difference between size and capacity of the stack. Occasionally the user wants to make them to be the same. For example, after finishing all operations, he/she wants to keep the memory space used less as possible.

Implement trimToSize() method that works like trimToSize() in java.util.Stack().

Checkpoint: Mark your grade: Pass or Fail.

Step 4: Decreasing capacity automatically

While increasing the capacity automatically in both java.util.Stack and StackOfInts, it does not decrease its capacity automatically. If you had to resize the array every time that you decreased the array, you would end up re-copying the array each time. On the other hand, if the capacity remains the same, then you are taking up unneeded memory. This may cause a problem in a long run.

java.util.Stack does not decrease its capacity. However, rewrite pop() method in StackOfInts class such that its capacity decreases automatically to the half (50%) when its size becomes 1/4 (25%) of the capacity. This will prevent the system from re-copying the array too often. This algorithm goes into the pop() method.

Checkpoint: Let's test this case with **StackDriver3.java** as shown below:

```
public class StackDriver3 {
    public static void main(String[] args) {
        java.util.Stack<Integer> s = new java.util.Stack<>();
        s.trimToSize();
        // push 1000 elements to the stack (size = 0, capacity = 0)
        for (int i = 0; i < 1000; i++) s.push(i);
        System.out.println("size:" + s.size() + "  capacity:" + s.capacity());

        // pop 1000 elements from the stack and observe how the capacity changes
        for (int i = 0; i < 1000; i++) s.pop();
        System.out.println("size:" + s.size() + "  capacity:" + s.capacity());
    }
}
```

The capacity does not decrease even though its size became zero as shown below:

```
size:1000  capacity:1024
size:0  capacity:1024
```

After you implement the pop() method in **StackOfInts** as instructed, then the capacity becomes smaller as its size becomes smaller as shown below:

```
public class StackDriver31 {
    public static void main(String[] args) {
       StackOfInts s = new StackOfInts();
       s.trimToSize();
       // push 1000 elements to the stack (size = 0, capacity = 0)
       for (int i = 0; i < 1000; i++) s.push(i);
       System.out.println("size:" + s.size() + "  capacity:" + s.capacity());

       // pop 1000 elements from the stack and observe how the capacity changes
       for (int i = 0; i < 1000; i++) {
        int before = s.capacity();
        s.pop();
        int after = s.capacity();
        if (before != after)
           System.out.println("NEW  size:" + s.size() + "  capacity:" + s.capacity());
       }
       System.out.println("LAST size:" + s.size() + "  capacity:" + s.capacity());
    }
}
```

Checkpoint: The expected output is as shown below:

```
size:1000  capacity:1024
NEW  size:255  capacity:512
NEW  size:127  capacity:256
NEW  size:63  capacity:128
NEW  size:31  capacity:64
NEW  size:15  capacity:32
NEW  size:7  capacity:16
NEW  size:3  capacity:8
NEW  size:1  capacity:4
NEW  size:0  capacity:2
LAST size:0  capacity:2
```

Step 5: Printing elements in Stack

Quite often we would like to print elements in the Stack. It is a good idea to provide such a capability when you create a class. In Java it usually handles by overriding toString() method that returns a string representation of the object. The java print methods invoke toString method() to get a string representation of the object before printing. For example,

```
public class StackDriver4 {
    public static void main(String[] args) {
       java.util.Stack<Integer> s = new java.util.Stack<>();
       // StackOfInts s = new StackOfInts();
       s.push(2);
       s.push(8);
       s.push(1);
       s.push(9);  System.out.println("stack:" + s);
       System.out.println("stack:" + s.toString());
    }
}
```

The StackDriver4 program produces the following output as shown below:

```
stack:[2, 8, 1, 9]
stack:[2, 8, 1, 9]
```

Override toString() method such that it prints the stack just like java.util.Stack.

1. Use either StringBuilder or StringBuffer class, **not String**.
2. Use @Override annotation in Java. It is used when we override a method in subclass. Generally novice developers overlook this feature as it is not mandatory to use this annotation while overriding the method. It is considered as a best practice in java coding since it checks for its correctness with the method signature you intend to override.

Checkpoint: Run StackDriver4.java with StackOfInts instead of java.util.Stack, then it produces the following output:

```
stack:[2, 8, 1, 9]
stack:[2, 8, 1, 9]
```

Step 6: Throwing `EmptyStackException` class object

Let's run the following code twice while alternating the first and the second line in the code.

```
public class StackDriver5 {
    public static void main(String[] args) {
       java.util.Stack<Integer> s = new java.util.Stack<>();
       // StackOfInts s = new StackOfInts();
       s.push(2);
       s.push(8);
       s.push(1);
       s.push(9);
       System.out.println(s);

       // pop more elements than its size to see an Exception
       for (int i = 0, n = s.size() + 1; i < n; i++)
        s.pop();
    }
}
```

Since it tries to pop elements more than the stack has, the program generates error messages. If you compare two messages, the error message from java.util.Stack shows EmptyStackException object, but the error message from StackOfInts shows about an array index out of bound which is undesirable.

```
[2, 8, 1, 9]
Exception in thread "main" java.util.EmptyStackException
    at java.util.Stack.peek(Stack.java:102)
    at java.util.Stack.pop(Stack.java:84)
    at StackDriver5.main(StackDriver5.java:14)
```

Modify pop() method such that it throws an EmptyStackException if pop() is called when the stack is empty.

Checkpoint: Run StackDriver5.java with StackOfInts. Your output should be the same Exception object thrown.

Part II: Creating a generic and iterable stack

While the StackOfInts class can stack only int data type, but java.util.Stack can work with different data type objects. Run the warming-up program, StackDriver6.java, and observe the output.

```
public class StackDriver6 {
    public static void main(String[] args) {
      Stack<Integer> is = new Stack<>();
      is.push(new Integer(3));
      is.push(1);
      is.push(6);
      System.out.println(is);

      Stack<String> ss = new Stack<>();
      ss.push("Why Not");
      ss.push("Change");
      ss.push("World?");
      System.out.println(ss);

      Stack<Object> as = new Stack<>();
      as.push(new Integer(3));
      as.push("Hello");
      as.push("World");
      System.out.println(as);

      for (Object o : as)
       System.out.print(o + " ");
      System.out.println();

      Iterator<String> it = ss.iterator();
      while (it.hasNext())
       System.out.print(it.next() + " ");
    }
}
```

It produces the following output:

```
[3, 1, 6]
[Why Not, Change, World?]
[3, Hello, World]
3 Hello World
Why Not Change World?
```

Let us write a new stack called **StackGeneric** that may accept different data type objects like java.util.Stack eventually.

Step 1: Using java.util.ArrayList

The StackOfInts class used the int type array to store elements. Let us ArrayList class to store elements in the StackGeneric. Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.

- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

You will realize that the implementation of StackGeneric class is much simpler than that of StackOfInts. The ArrayList does not reveal about the 'capacity' of the ArrayList object. Don't try to implement it. This is a simple class diagram and the beginning of the class code.

```
StackGeneric
-list: ArrayList<Integer>
+StackGeneric()
+empty(): boolean
+peek(): int
+push(value: int): void
+pop(): int
+size(): int
+clear(): void
```

```
public class StackGeneric {

    private List<Integer> list = new ArrayList<>();

    /** Construct a stack with the default capacity */
    public StackGeneric() {

    }

    /** Returns the top of stack, no change in size */


    /** Push a new integer into the top of the stack */


    /** Return and remove the top element from the stack */


    /** Make the stack empty */


    /** Return true if the stack is empty */


    /** Return the number of elements in the stack */


    @Override
    public String toString(){

    }
```

Implement StackGeneric.java, and run it with **StackDriver7.java**. Expect the following output.

```
public class StackDriver7 {
    public static void main(String[] args) {
       StackGeneric s = new StackGeneric();
       s.push(3);
```

```
        s.push(1);
        s.push(s.peek() + 5);
        s.push(s.peek());

        System.out.println("peek:" + s.peek());
        System.out.println("size:" + s.size());
        System.out.println("stack:" + s);

        if (!s.empty()) s.clear();
      }
  }
```

```
peek:6
size:4
stack:[3, 1, 6, 6]
```

Checkpoint: Once you implement StackGeneric successfully, it should run with StackDriver7.java. Your output should be the same as shown above.

Step 2: Java Generics – use of generic types and methods

Now we want to enhance the StackGeneric class such that it accepts different types of data objects. This can be implemented by using Java Generics or parameterizing the types for variables and methods.

Let's make a new class called StackGeneric2 from StackGeneric.

When you define the StackGeneric2, let it take a **generic type (i.e. use <E> or <T>)** and use it when you define instance variables and methods. The caller (i.e. user of StackGeneric) instantiate to form parameterized type by providing actual arguments that replace the formal type parameters.

As you see the first line of StackGeneric class definition, a parameterized type or a generic type <T> is passed into the class. During instantiation, or when you use the class first, you provide it with a specific type such as ArrayList<String> or a ArrayList<Integer>, where String and Integer are the respective actual type arguments. **The motivation using Java generics is to detect errors at compile time rather than at runtime.**

You must complete the following program such that it uses a formal generic type <T>.

```
public class StackGeneric2 <T> {
    private List<T> list = new ArrayList<>();

    /** Construct a stack with the default capacity */
    public StackGeneric2() {
    }

    /** Push a new integer into the top of the stack */
    public void push(T item) {
       list.add(item);
    }
. . .
. . .

}
```

Checkpoint: Complete the StackGeneric2.java, run it with **StackDriver8.java** and pass the actual type arguments. Then expect the following output.

```
public class StackDriver8 {
    public static void main(String[] args) {
       StackGeneric2<Integer> is = new StackGeneric2<>();
       is.push(new Integer(3));
       is.push(1);
       is.push(6);
       System.out.println(is);
       StackGeneric2<String> ss = new StackGeneric2<>();
       ss.push("Why Not");
       ss.push("Change");
       ss.push("World?");
       System.out.println(ss);

       StackGeneric2<Object> as = new StackGeneric2<>();
       as.push(new Integer(3));
       as.push("Hello");
       as.push("World");
       System.out.println(as);
    }
}
```

```
[3, 1, 6]
[Why Not, Change, World?]
[3, Hello, World]
```

[For your information] Generic method examples

The following two examples show how to write generic methods to provide the method with more versatile argument types.

```
    /**
     * prints all items in inputArray, separated by a space in a line.
     */
    public static <E> void printArray(E[] inputArray) {
          for(E element : inputArray)
             System.out.print(element + " ");
          System.out.println();
    }
```

```
/**
     * Returns the number of times that itemToCount occurs in list.
     * Items in the list are tested for equality using itemToCount.equals(),
     * except in the special case where itemToCount is null.
     */
    public static <T> int countMatches(T[] list, T itemToCount) {
       int count = 0;
       if (itemToCount == null) {
        for (T listItem : list)
           if (listItem == null)
               count++;
       } else {
        for (T listItem : list)
           if (itemToCount.equals(listItem))
               count++;
       }
       return count;
```

```
      }
```

The countMatches method operates on an array. We could also write a similar method to count matches of an object in any collection. The next countMatches() method is very general since Collection<T> is itself a generic type. It can operate on an ArrayList of Integers, a TreeSet of Strings, a LinkedList of JButtons, etc.

```
public static <T> int countMatches(Collection<T> collection, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T item : collection )
         if (item == null)
            count++;
   }
   else {
      for ( T item : collection )
         if (itemToCount.equals(item))
            count++;
   }
   return count;
}
```

Checkpoint: Using the following program you may check the generic methods defined above.

```
public static void main(String[] args) {
       Integer[] ia = {2, 4, 6, 8, 2};
       String[] sa = {"Why not", "Change", "the World"};
       printArray(ia);
       printArray(sa);

       ArrayList<Integer> al = new ArrayList<>(Arrays.asList(ia));

       System.out.println("Count:" + countMatches(ia, 2));
       System.out.println("Count:" + countMatches(sa, "Change"));
       System.out.println("Count:" + countMatches(al, 2));
}
```

It produces the output:

```
2 4 6 8 2
Why not Change the World
Count:2
Count:1
Count:2
```

Step 3-1: Iterable interface using anonymous inner class

Let's run the following code twice **while alternating** the first and the second line in the code.

```
public class StackDriver9 {
    public static void main(String[] args) {
       java.util.Stack<Integer> s = new java.util.Stack<>();
       // StackOfInts s = new StackOfInts();
       s.push(3);
       s.push(1);
       s.push(6);
```

```
        System.out.println(s);

        // using for-each loop
        for (int i : s)
            System.out.print(i + " ");
    }
}
```

The program with the first line above produces the following output.

```
[3, 1, 6]
3 1 6
```

With the second line, however, the second line would have a syntax error. Why? Because the object s used in for-each loop is not iterable.

For-each is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

This kind of capabilities including linear search, sort, and iteration can be done using Iterator interface. Any class that wishes to allow search, sort and iteration can do so by implementing Iterable (i.e. by providing an Iterator). There are two Iterator interface predefined in Java.

- java.util.Iterator
- java.util.Iterable

There are three ways to create an Iterator for class X for example, we can

- Use a separate class
- (Use an inner class within X)
- Use an anonymous inner class within X while overriding iterator() method

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove(); // Optional operation
}
```

To make a class iterable, the class has to implement the **iterable** interface which requires to **override the iterator method()**. Notice that the iterator() method returns an Iterator object. This means that we have to

a. Declare "implements Iterable<Character>" at the class header

b. Override iterator() method that returns an Iterator object.

```
interface Iterable {
    Iterator iterator();
}
```

Before we challenge to make the StackGeneric class iterable, let us go through a simple example.

StringIterable: An iterable interface example

A string, a String class object in Java, is not iterable. To iterate one character at a time in a String, you need to convert it to a character array by using toCharArray() method provided with String class.

```
public static void main(String[] args) {
       String s = "Hello World";
       for(char c: s.toCharArray())
         System.out.print(c + " ");
}
```

Let us define a **StringIterable** class which is iterable. It has an instance variable s to store a string. To make a class iterable, the class has to implement the **iterable** interface which requires to **override the iterator method()** that must return an Iterator object.

To make an Iterator object used in iterator method() once, we have to define our own class which implements the **Iterator interface**.

Method I: Using an own separate class

1) To make StringIterable iterable, it must implement the Iterable interface.
   This means that we have to

   A. Declare "implements Iterable<Character>" at the class header

   B. Override iterator() method that returns an Iterator object.

2) To return an Iterator object, we define our own class (e.g. StringIterator in this case) that implements iterator() interface.
   This means that we have to

   A. Declare "implements Iterator<Character>" at the StringIterator class header

   B. Implement hasNext(), next(), remove() as well as a constructor.

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

public class StringIterable implements Iterable<Character>{
    private String s;

    public StringIterable(String string) {
         s = string;
    }

    @Override
    public Iterator<Character> iterator() {
         return new StringIterator(this.s);
    }
}

class StringIterator implements Iterator<Character> {
    private String str;
    private int count;

    public StringIterator(String s) {    // constuctor
       str = s;
    }
```

```
        @Override                                     // The next three methods implement Iterator.
        public boolean hasNext() {
           return count < str.length();
        }

        @Override
        public Character next() {
           if (count == str.length())
             throw new NoSuchElementException();
           count++;
           return str.charAt(count - 1);
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
```

Checkpoint: Check the StringIterable class using the **StringIterableDriver.java** program.

```
    import java.util.Iterator;

    public class StringIterableDriver {
        public static void main(String[] args) {
           StringIterable x = new StringIterable("Hello World");
           // StringIterable2 x = new StringIterable2("Hello World");
           for (char ch : x)
              System.out.print(ch + " ");
           System.out.println();

           StringIterable2 y = new StringIterable2("Why not");
           Iterator<Character> it = y.iterator();
           while (it.hasNext()) {
              char ch = it.next();
              System.out.print(ch + " ");
           }
        }
    }
```

Method II: Using anonymous inner class

An **anonymous class** is an inner class with the usual class body, but

- has no name
- can be instantiated only once
- is usually declared inside a method or a code block, a curly braces ending with semicolon.
- is accessible only at the point where it is defined.
- does not have a constructor simply because it does not have a name

```
new classOrInterfaceName {
    ... body ...

};
```

To make StringIterable iterable, it must implement the Iterable interface. This means that we have to

1) Declare "implements Iterable<Character>" at the class header
2) Override iterator() method that returns an Iterator object. Instead of creating our own class, use anonymous inner class and implement hasNext(), next(), remove() as well as a constructor.

Let us name this implementation of the class **StringIterable2**.

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

public class StringIterable2 implements Iterable<Character>{
    private String s;

    public StringIterable2(String string) {
       s = string;
    }

    @Override      // using anonymous inner class
    public Iterator<Character> iterator() {
       return new Iterator() {
         private int count;

         // The next three methods implement Iterator.
         public boolean hasNext() {
           return count < s.length();
         }

         public Character next() {
           if (count == s.length())
                  throw new NoSuchElementException();
           count++;
           return s.charAt(count - 1);
         }

           public void remove() {
              throw new UnsupportedOperationException();
        }
       };
    }
}
```

Checkpoint: Check **StringIterable2.java**, using the same program **StringIterableDriver.java** shown previously and expect the following output.

```
public class StringIterableDriver {
    public static void main(String[] args) {
       // StringIterable x = new StringIterable("Hello World");
       StringIterable2 x = new StringIterable2("Hello World");
 . . .
 . . .
```

```
H e l l o   W o r l d
W h y   n o t
```

Once you have experienced in implementing the iterable interface, let us apply it for making the StackGeneric2 class iterable.

Step 3-2: Iterable interface using anonymous inner class

Method I: Using a separate class;

Let us name this class implementation as **StackGeneric3**.

```
public class StackGeneric3 <T> implements Iterable<T>{
    private List<T> list = new ArrayList<>();
    /** Construct a stack with the default capacity */
    public StackGeneric3() {
    }
. . .
. . .
    @Override
    public Iterator <T> iterator() {
        return new MyIterator<T>(this.list);
    }
```

Create our own custom class, MyIterator, and make it implement Iterable so that it returns an Iterator using which we can iterate the elements.

```
class MyIterator<T> implements Iterator<T> {

    //  your code here


    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Checkpoint: The program StackDriver10.java program now should work with StackGeneric3.

```
import java.util.Iterator;

public class StackDriver10 {
    public static void main(String[] args) {
       StackGeneric3<String> s = new StackGeneric3<>();
       // StackGeneric4<String> s = new StackGeneric4<>();
       // StackGeneric5<String> s = new StackGeneric5<>();
       s.push("Why not");
       s.push("Change");
       s.push("the World");
       System.out.println(s);

       // using for-each loop
       for (String it : s)
            System.out.print(it + " ");
       System.out.println();

       // using iterator
       Iterator<String> it = s.iterator();
       while (it.hasNext())
            System.out.print(it.next() + " ");
       System.out.println();
    }
    }
```

Method II: Using an anonymous inner class while overriding iterator() method

Let us name this class implementation as **StackGeneric4**.

```
public class StackGeneric4 <T> implements Iterable<T>{
    private List<T> list = new ArrayList<>();

    /** Construct a stack with the default capacity */
    public StackGeneric4() {
    }
. . .
. . .
    /*********************************************
    @Override
    public Iterator <T> iterator() {
       return new MyIterator<T>(this.list);
    }
    **********************************************/
    @Override
    public Iterator <T> iterator() {
       return new Iterator<T>() {
       // your code here
       };
    }
}
```

Checkpoint: The StackDriver10.java program should also work with StackGeneric4..

Step 4: The simplest solution for this particular case^^

There is the simplest solution to make it iterable in this particular example. We don't need to make our own class to implement the iterator interface **since the instance variable list uses ArrayList which already provides iterator() method. Why don't we just use it? Wow… it is a grace of Java^^.**

Notice that we have to define our own class that implements the iterator interface or an inner class when we use either String or int array which don't provide with an iterator() method.

Let us name this class implementation as **StackGeneric5**.

```
public class StackGeneric5 <T> implements Iterable<T>{
    private List<T> list = new ArrayList<>();

    /** Construct a stack with the default capacity */
    public StackGeneric5() {
    }
. . .
. . .
    @Override
    public Iterator <T> iterator() {
      return ____________ your code here _____________________
    }
. . .
```

Fill the blank with an appropriate code. Then StackGeneric5 should work without any other inner class or anonymous inner class definitions.

Checkpoint: StackDriver10.java should also work with StackGeneric5.

Part III: Make StackOfInts class iterable

Create **StackOfIntsIterable** class which is iterable by overriding iterator method using **an anonymous inner class**. Basically, you implement the iterable interface in StackOfInts class which is a result of Part I (StackOfInts.java), not Part II(StackGeneric.java).

[FYI: For your information] If you want to make a new class StackOfIntsIterable from StackOfInts, refer to [FYI] in page 2 of this document.

Checkpoint: The following code now should work if you code it right.

```
import java.util.Iterator;

public class StackDriver11 {
    public static void main(String[] args) {
       StackOfIntsIterable s = new StackOfIntsIterable();
       // push() and print stack
       s.push(3);
       s.push(1);
       s.push(6);
       System.out.println(s);

       for (Integer it: s)
            System.out.print(it + " ");
       System.out.println();

       Iterator<Integer> it = s.iterator();
       while (it.hasNext())
            System.out.print(it.next() + " ");
       System.out.println();
    }
}
```

Run StackOfIntsIterable with the **StackDriver11.java** program and expect the following output.

```
[3, 1, 6]
3 1 6
3 1 6
```

If your output looks something like this, then you make sure that you the size instead of the element length in your iterator code.

```
3 1 6 0 0 0 0 0 0 0
```

Honor code and Submitting your solution

- This is not a group project. But I encourage you to share your knowledge or to get help each other. You may discuss or debug the code together, but never share your code with others. You are responsible for submitting your work by yourself.
- Include the following line at the top of your source files with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment. Signed: ______________
- Include your name, student number and email address.
  Include your group partner's name and student number.
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the homework problem partially before the due, you still need to turn it in. However, don't turn it in if it does not compile and run.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

Submit the following source files on time.
Use **lab9** folder in Piazza

- For Part I: StackOfInts.java
- An example: StringIterable.java, StringIterable2.java
- For Part II; StackGeneric.java, StackGeneric2.java, StackGeneric3.java, StackGeneric4.java, StackGeneric5.java
- For Part III: StackOfIntsIterable.java
- ProjectStackScores.docx with **Warming-up questions and scores recorded.**

Due and Grade points

- Due: 11:55 pm, Saturday, Nov 24, 2018
- Grade points:
  - Warming-up: 0
  - Part I: 2 points, Part II: 3 points, Part III: 2 points
  - An example - StringIterable : 1 point
  - Max penalty for wrong or inflated grading: -2.0
- My total score: ______________ by myself