Lab 2

Dynamic array and Iterators

Dynamic array

A Dynamic array (`Vector` in C++, `ArrayList` in Java) automatically grows when we try to make an insertion and there is no more space left for the new item. Usually by at least twice the original size;

Dynamic array (Cont'd)

```java
import java.util.Arrays;

public class DynamicArray<E> {
    private int size;
    private E[] elements;

    // ... constructor and etc

    public void add(E e) {
        if (size == elements.length) {
            ensureCapacity();
        }
        elements[size++] = e;
    }

    private void ensureCapacity() {
        int newSize = elements.length * 2;          // double the size
        elements = Arrays.copyOf(elements, newSize);
    }
}
```

Type *Erasure*

In short the underlying compiled classes are not actually generic. They compile down to Object and casts. In effect Java generics are a compile time artifact and can easily be subverted at runtime.

Type *Erasure* (Cont'd)

type erasure vs. type-passing semantics

- Code with generics:

```
class Foo<T> {
    T Bar(T item) { return item; }
}

// Use:
Foo<String> f = new Foo<String>();
```

Type *Erasure* (Cont'd)

type erasure vs. type-passing semantics

- Compiled code via type erasure, generic type is *erased*

```
class Foo {
    Object Bar(Object item) { return item; }
}

// Use:
Object f = new Foo();
```

Type *Erasure* (Cont'd)

type erasure vs. type-passing semantics

- Compiled code via type-passing semantics, generic type is kept:

```
class Foo`1 {
    String Bar(String item) { return item; }
}

// Use:
Foo`1 f = new Foo`1();
```

P.S.: "1" indicates that `Foo`1` is the first generic class definition of `Foo`

Example

```
class Foo { }
class Bar { }
class Baz { }

public static void main(String[] args) {
  // All share the same compile time code because type annotations are removed!
  DynamicArray<Foo> foo = new DynamicArray<Foo>();
  DynamicArray<Bar> bar = new DynamicArray<Bar>();
  DynamicArray<Baz> baz = new DynamicArray<Baz>();
}
```

As a result of *type-erasure*:

```
// Constructor
public DynamicArray() {
    this.size = 10;
    this.elements = (E[]) new Object[this.size]; // Instantiate generic array!
}
```

Class Invariant

The invariant must hold to be true after the constructor is finished and at the entry and exit of all public member functions.

```
private boolean wellFormed() {
    // Example of an invariant?
}
```

```
public void add(Integer element) {

    assert wellFormed() : "Failed at the start of add";

    // Add to element to Data-Structure

    assert wellFormed() : "Failed at the end of add";
}
```

Invariant vs. Unit test

- Overview:
    - An invariant is a condition that is a pre-condition *and* a post-condition
    - Unit tests have data and may follow certain steps to test a behavior
- Internal vs. External
    - Invariant validate the internal state of ADT
    - Unit test validate the external behavior of ADT
- Different goals
    - For invariants, the goal is to make debugging easier by observing invalid program states as soon as they occur
    - For unit tests, the goal is to find bugs

Sequence ADT

An ordered collection of items, one of which is the "current" item.

Sequence ADT (Cont'd)

- `size` : return the number of items in the sequence
- `addBefore` : add a given item just before the current item, or at the front of the sequence if there is no current item; make the new item the current one
- `addAfter` : add a given item just after the current item, or at the end of the sequence if there is no current item; make the new item the current one
- `removeCurrent` : remove the current item (error if there is no current item); if the current item is the last item in the sequence, then after the remove operation there is no current item; otherwise, make the next item the current item
- `start` : make the first item in the sequence be the current item.
- `getCurrent` : return the current item advance advance the current item
- `hasCurrent` : return true if there is a current item; otherwise, return false
- `lookup` : return true if a given item is in the sequence; otherwise, return false

Bag ADT

Bags are containers, they hold things. They are not ordered.

Bag ADT (Cont'd)

- `add` : Put something in
- `remove` : Take an item out
- `clear` : Take everything out
- `getFrequencyOf` : Count how many things are in it
- `isEmpty` : See if it is empty
- `contains` : Check to see if something is in it
- `getCurrentSize` : Count the items in it
- `display` : Look at all the contents

Exercise

Let's implement `ensureCapacity`

```
this.data = new int[] { 1, 3, 5, ... };

public void ensureCapacity(int minimumCapacity) {

    // Lets implement this method ...

}
```

Lab assignment #2: