Outline

- reminder of invariants and circular lists
- doubly-linked lists
- iterators
- the Java `foreach` statement
- iterator implementation
- the `ListIterator` interface

Using Invariants (reminder)

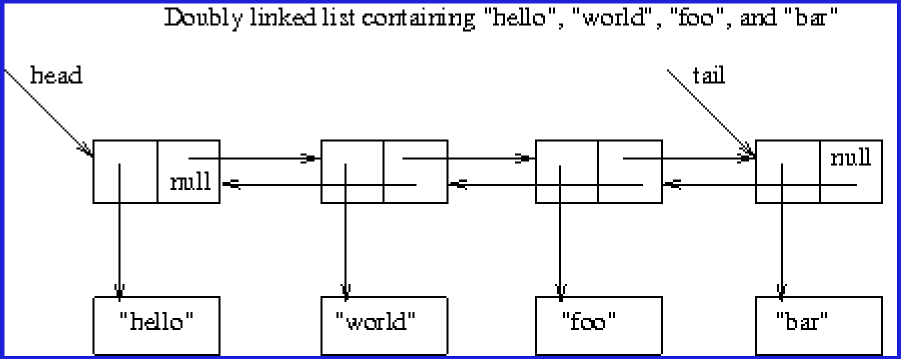
- invariants are useful for reasoning about the program
- some invariants can be checked by the program itself
- if an invariant is ever detected to not be true, the program should provide enough information to track down the bug (and should either crash, or re-establish the invariant)
- in ICS 211, if your classes have any invariants, it will be easier to debug your code if you check them at the beginning and at the end of each public method
- this may help you improve your understanding of your code, and may also help you find bugs
  - see also the linked list invariants
- if checking is slow, you may have to remove the invariant checking (e.g., add a return statement at the beginning of the check method) when doing performance testing

Circular Linked Lists reminder

- instead of the last `next` field pointing to `null`, it points to the `head` of the linked list
- then the head can be reached as `tail.next`, and does not need to be kept explicitly
- looping must end when `current == tail`, rather than when `current == null`

Doubly-Linked Lists reminder

- the linked lists so far have the limitation that it is only possible for code to follow references in one direction in the list, that is, forward
- node removal requires a reference to the node before the node to be removed
- if each node also keeps a reference to the node before it, both these problems can be solved


Doubly linked list containing "hello", "world", "foo", and "bar"
head
tail
null
null
"hello"
"world"
"foo"
"bar"

Nodes for Doubly-Linked Lists

```
private class DLinkedNode<E> {
  private E item;                // one element
  private DLinkedNode<E> prev;   // two references, one to the node before
  private DLinkedNode<E> next;   // and one to the node after

  private DLinkedNode(E value) {
    item = value;
    next = null;
    prev = null;
  }

  private DLinkedNode(E value, DLinkedNode<E> prev, DLinkedNode<E> next) {
    item = value;
    this.next = next;
    this.prev = prev;
  }
}
```

Doubly-Linked list add

- adding after a given node (node) means updating the previous and next node's next and prev references:

```
    DLinkedNode followingNode = node.next;
    node.next = new DLinkedNode (value, node,
                                 node.next);
    followingNode.prev = node.next;
```

- in-class exercise (alone or with a friend or two): draw the doubly-linked list after each of the lines of the above code
- the above code assumes that there is both a previous and next node
- if not, the code needs special cases
- a circular list, if coded correctly, needs fewer special cases

Doubly-Linked list `remove`

- removing a given node (`node`) means updating the node's predecessor's next field, and the node's successor's prev field:

```
   node.prev.next = node.next;
   node.next.prev = node.prev;
```

- here are the special cases for a doubly-linked list that is not circular:

```
 if ((node == head) && (head.next == null)) {
   head = null; tail = null;
 } else {
    if (node.prev != null)
       node.prev.next = node.next;
    if (node.next != null)
       node.next.prev = node.prev;
 }
```

Looping over the elements of a collection

- sometimes we want do something with all of the elements of a collection
- for example, we might want to print the values
- or we might want to add all the values in a collection of numbers
- we can do a loop with get:

```
for (int i = 0; i < List.size(); i++) {
  E element = List.get(i);
  ... // do something with element
}
```

- if `get` takes more than constant time, this is very inefficient: the outer loop is repeated `List.size` times, so if the inner loop is also linear in the list size, the entire loop takes time $\texttt{List.size}^2$.

Efficiently Looping over the elements of a collection

- for a linked list, `get` takes linear time
- but accessing a list element *if we have a reference to the node containing the element* only takes constant time
- it is not safe to let the user program directly have access to this reference
- instead, the reference is encapsulated in an object called an **iterator**, which only provides a small set of operations

using Java iterators

```
List<E> list = …
for (E element: list) {
    …
}
```

- Java internally re-writes the above loop as:

```
Iterator<E> it = list.iterator();
while (it.hasNext()) {
    E element = it.next();
    ….
}
```

Yes, but what is an iterator?

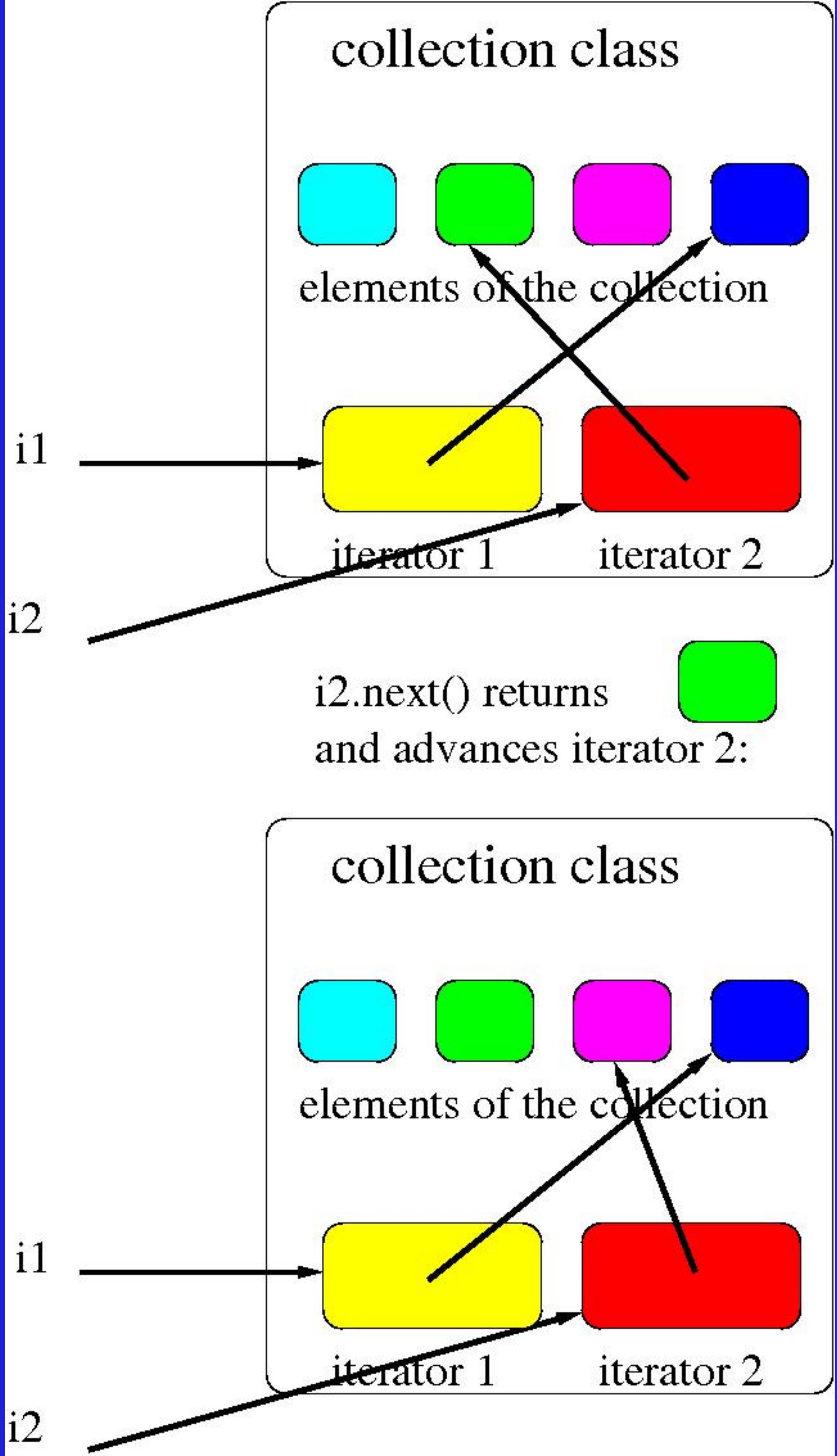
- an iterator is an object that supports the two methods `hasNext()` and `next()`
- `next()` provides access to the elements of a collection
- for example, an iterator for a linked list class would internally have a reference to the node containing the next object:

```
public class LinkedListIterator<E> … {
    LinkedNode<E> node;
```

- the iterator is a different object than the collection

example

- i1 and i2 are iterators for the same collection
- advancing i2 does not affect i1
- `next()` returns the next element **and** advances the iterator



Iterator methods

- a Java iterator only provides two or three operations:
- `E next()`, which returns the next element, and also advances the references
- `boolean hasNext()`, which returns whether there is at least one more element
- `void remove()`, which removes the last element returned by `next()` (this method is optional)
- using `remove` may invalidate any other existing (concurrent) iterators

Automatic use of iterators

- instead of having to use the while loop to use an iterator, the `for` loop has been specialized to call the iterator

```
LinkedList values = ...
int sum = 0;
for (Integer value: values) {
  sum = sum + value;
}
```

- Java creates and calls the iterator, but the iterator itself is not visible in the code
- the same code can loop over arrays

Java `for` and `foreach`

- this automated (and invisible) use of iterators with for loops is called the Java **enhanced for** statement or **for each** statement
- the **foreach** statement works on any expression that has a value that satisfies the **Iterable** interface
- The Iterable interface simply requires an **iterator** method:

```
class MyCollection<E>
      implements Iterable<E> {
// return a new iterator
Iterator<E> iterator();
```

Iterator Implementation

- a Java iterator may or may not be internal to the collection class
- every Java iterator must have sequential access to the elements of the collection
- every Java iterator must have at least one variable to keep track of where it is in the traversal, that is, which elements have not yet been returned
- See LinkedListIterator.java for a very simple iterator on linked lists.
- in-class exercise (everyone together): design the code for the `iterator()` method of the LinkedList class

ListIterator

- the Java `Iterator` interface is very general and reasonably powerful
- sometimes it is useful to be able to move backwards and forwards, and add or replace as well as remove elements
- the `ListIterator` interface adds these operations to the basic Iterator interface
- it also keeps track of the position and can return the index of the next or previous item