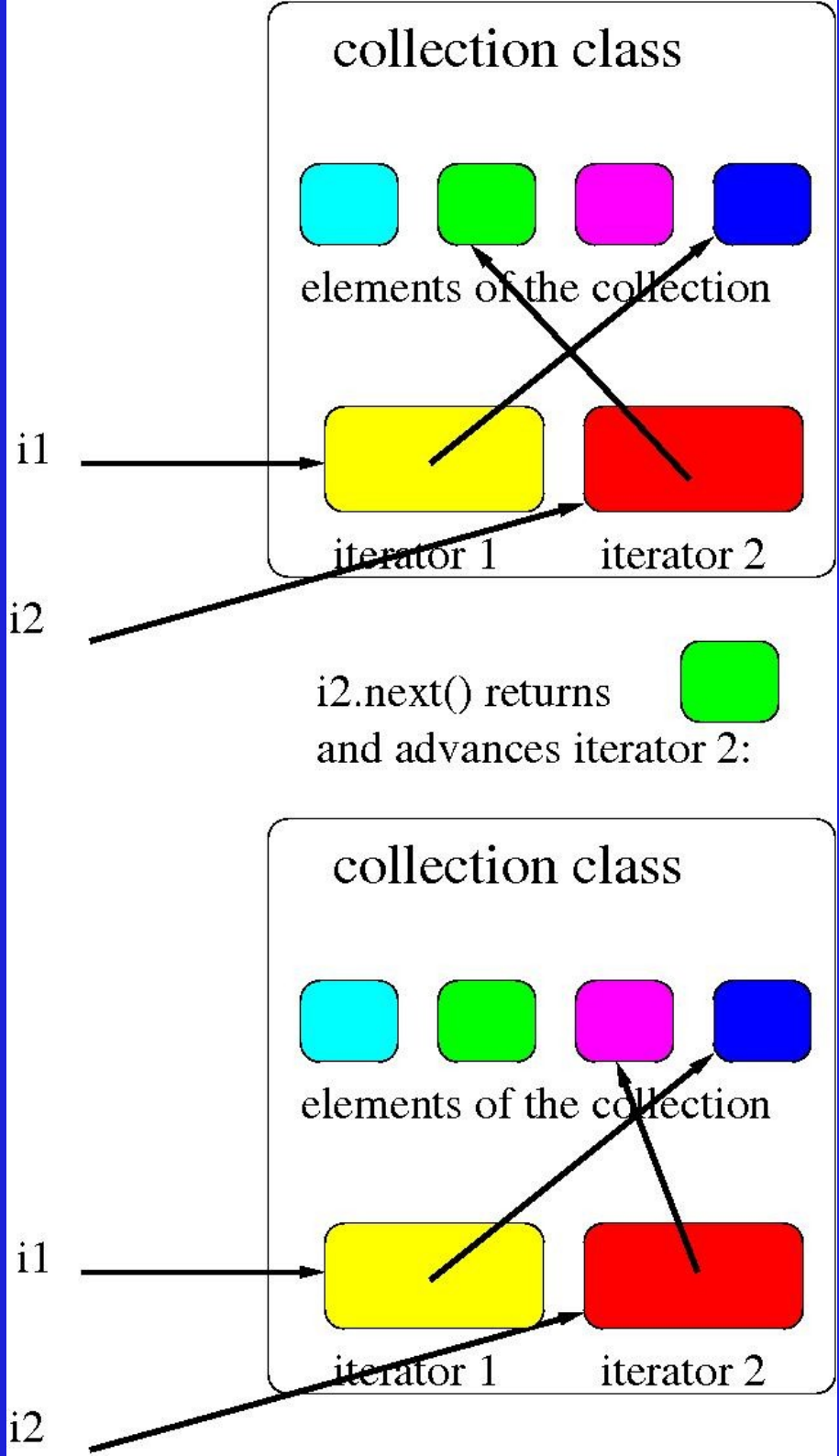
Outline

- iterators
- the Java `foreach` statement
- iterator implementation
- the `ListIterator` interface

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
  - but should not invalidate this iterator

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();
    ….
}
```

Automatic use of iterators: another example

- as stated, instead of having to use an iterator in a while loop 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

Java `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
- the current position is defined to be in-between the `previous` element and the `next` element

ListIterator Interface

```
public interface ListIterator<E> {
  void add(E e);
  boolean hasNext();
  boolean hasPrevious();
  E next();
  int nextIndex();
  E previous();
  int previousIndex();
  void remove();
  void set(E e);
}
```

Testing

- unless code is written correctly from the start, errors are found by testing
- but writing correct code from the start is hard, so most programmers use testing to make their code as close to correct as possible
- test routines can be included in the `main` method of any class that doesn't already have one, or in a separate test program. This is a *unit test*. This main method is a *driver program*. The driver program can also be defined separately.
- the unit test should call all the methods of the class, with as many combinations of parameters as possible
- if the writer of the test code doesn't study the code under test, this is *black box testing*
- at the very least, the goal of testing is *full coverage*: making sure that every path through the code has been used at least once, and has produced an acceptable result
- to produce full coverage, the programmer of the test program must study the code being tested: this is *white box testing*

A few common types of errors

- off-by-one (fencepost): how many fence posts are needed for a fence that is 20 feet long and has a post every 2 feet?
- not initializing data correctly. Sometimes this causes null pointer access
- using different variables as if they were one, or using one variable as if it were two variables
- assumptions that don't turn out to be true (misconceptions), not establishing and maintaining invariants
- not checking things that should be checked, e.g.

```
if (x == null)
```

strategies for testing

- print/show all method invocations and their parameters and return values
- write code to check that the invariants are established and maintained
- write test cases to not only provide full coverage, but also check all boundary conditions, where the result should change (make sure it changes where it should)
- some common special cases:
  - less than 0, 0, 1, greater than 1
  - first and last elements of an array, collection, linked list
  - elements and values that are null
  - desired element is not in the collection, or is in the collection more than once
  - collection has size 0, 1, or larger
- for example, when testing adding on a linked list, can test adding at the beginning of a linked list, at the end, in the middle, and adding into an empty linked list, and in both positions of a 1-element linked list. Also, adding an element that has the value `null` (is the behavior of your program defined in that case? Should it be?)
- if code to be tested needs to call a method that is not yet implemented, a *stub* of that method can do only what is needed for the test

reasoning about programs

- a **precondition** must be true before a method is called
- the code in the method is designed assuming that the precondition is true
- the caller of a method must guarantee (be sure) that the precondition holds
- a **postcondition** will be true after a method is called
- the code in the method must guarantee that the postcondition is true
- preconditions and postconditions are a little bit like a contract or any other agreement: if the caller provides the preconditions, the method will provide the postconditions
- preconditions and postconditions are documented in Javadoc
- invariants are postconditions of every method, **including** the constructors
- invariants are preconditions of every method **except** the constructors
- invariants are usually documented for the entire class rather than for each method

proof of program correctness

- given a mathematical specification
- it might be possible to prove that a program implements that specification
- specification: `add1` adds 1 and returns the result

```
private static int add1(int parameter) {
   return parameter + 1;
}
```

- seems obvious
- but fails if parameter is `Integer.MAX_VALUE`