Implementing an Iterator
Loose Ends

• In implementing several kernel interfaces so far, you have been given code in the skeletons for the *iterator* method

• The code for this method is stylized and *sometimes* easy to adapt to a new situation, even if the code itself is hardly transparent!
  – Several new Java issues arise ...
Iterators

• Recall: iterators offer a special way of getting \textit{sequential access} to all elements/entries of a collection

• Because linked data structures are particularly appropriate for sequential access, the \texttt{List2} code is a good place to examine how iterators can be implemented
iterator Contract for List

Iterator<T> iterator()

• Returns an iterator over the elements.
• Ensures:

\[
\sim \text{this.seen} \ast \sim \text{this.unseen} = \text{this.left} \ast \text{this.right}
\]
iterator Contract for List

Iterator<T> iterator()

• Returns an iterator over the elements.
• Ensures:
  \(~this.seen \times \simthis\) 
  \(\simthis.left \times \simthis.right\)

Iterator is an interface in the Java libraries (in the package java.util).
iterator()

Returns an iterator over the elements.

Ensures:

\(~\text{this.seen} \times \sim\text{this.unseen} = \text{this.left} \times \text{this.right}~

These two variables stand for the string of $T$ already seen and the string of $T$ not yet seen while using the iterator.
For-Each Loops

• Since List<T> extends the interface Iterable, you may write a for-each loop to “see” all elements of List<T>:

```java
for (T x : s) {
    // do something with x, but do not call methods on s or change the value of x
}
```
For-Each:

- Since `List<T>` extends the interface `Iterable`, you may write a `for-each` loop to “see” all elements of `List<T>` `s`:

```java
for (T x : s) {
    // do something with x, but do not call methods on s or change the value of x
}
```

This declares `x` as a local variable of type `T` in the loop; on each iteration, `x` is aliased to a different element of `s`. 
For-Each Loop

• Since List<T> extends the interface Iterable, you may write a for-each loop to “see” all elements of List<T>:

```java
for (T x : s) {
    // do something with x, but do not call methods on s or change the value of x
}
```

The restrictions on what you may do with x and s in the loop body are critical; do not forget about them!
How a For-Each Loop Works

• The for-each loop above is actually **syntactic sugar** for the following code:

```
Iterator<T> it = s.iterator();
while (it.hasNext()) {
    T x = it.next();
    // do something with x, but do
    // not call methods on s or
    // change the value of x
}
```
How a For-Each Loop Works

The for-each loop above is actually syntactic sugar for the following code:

```java
Iterator<T> it = s.iterator();
while (it.hasNext()) {
    T x = it.next();
    // do something with x, but do not call methods on s or change the value of x
}
```

The `iterator` method for `List<T>` returns a value of type `Iterator<T>`. The for-each loop is a more concise way to iterate over the elements of a collection.
How a For-Each Loop Works

- The for-each loop above is actually syntactic sugar for the following code:

```java
Iterator<T> it = s.iterator();
while (it.hasNext()) {
    T x = it.next();
    // do something with x, but do not call methods on s or change the value of x
}
```

The `hasNext` and `next` methods of this `Iterator<T>` variable are used in the iteration.
Iterating With `iterator`

- This code has the following properties:
  - It introduces aliases, so you must be careful to “follow the rules”; specifically, the loop body should not call any methods on `s`.
  - If what you want to do to each element is to change it (when `T` is a mutable type), then the approach does not work because the loop body should not change the value of `x`.
- With `List`, you could just use the kernel methods to visit the entries in the same order.
The `Iterator<T>` Interface

- For the `iterator` method, the kernel class returns a reference to an instance of a `nested class` (`List2Iterator`) that implements the `Iterator<T>` interface
- The code in that class implements these methods:
  - `boolean hasNext()`  
  - `T next()`  
  - `void remove()`
The `Iterator<T>` Interface

- For the `iterator` method, the kernel class returns a reference to an instance of a nested class (`List2Iterator<T>`) that implements the `Iterator<T>` interface.
- The code in that class implements these methods:
  - `boolean hasNext()`  
  - `T next()`  
  - `void remove()`
boolean hasNext()

- Returns **true** iff the iteration has more elements (i.e., there are any “unseen” elements).
- Ensures:

  hasNext = (~this.unseen /= < >)
next

T next()

- Returns the next element in the iteration (i.e., the next “unseen” element, which becomes a “seen” element).
- Aliases: reference returned by next
- Updates: ~this (i.e., the iterator, not the collection)
- Requires:
  ~this.unseen /= < >
- Ensures:
  ~this.seen * ~this.unseen =
  #~this.seen * #~this.unseen and
  ~this.seen = #~this.seen * <next>
Iterator for \texttt{List2}

\texttt{this} = (18, 6)
The object created by a call to `iterator` is an instance of the **nested class** `List2Iterator`...
... and it holds a reference to the node that has the first entry in ~this.unseen.
The `next` method returns the data in that node, also advancing `current` to the next node.
The `hasNext` method checks whether `current` is `null` (which, in this case, indicates that the last entry has been seen already).
A New Java Issue

• In the code inside the nested class `List2Iterator`, there are two references named `this`, so the name is ambiguous!
  – The name `this` denotes the object of type `List2Iterator` (the nested class)
  – The qualified name `List2.this` denotes the object of type `List2` (the enclosing class)

• See this line of code in the `List2Iterator` constructor:

  ```java
  this.current = List2.this.preStart.next;
  ```
The class `List2Iterator` has an instance variable named `current`, and this is it.

- In the code inside the nested class `List2Iterator`, there are two references named `this`, so the name is ambiguous!
  - The name `this` denotes the object of type `List2Iterator` (the nested class)
  - The qualified name `List2.this` denotes the object of type `List2` (the enclosing class)

See this line of code in the `List2Iterator` constructor:

```java
this.current = List2.this.preStart.next;
```
In the code inside the nested class `List2Iterator`, there are two references named `this`, so the name is ambiguous!

- The name `this` denotes the object of type `List2Iterator` (the nested class).
- The qualified name `List2.this` denotes the object of type `List2` (the enclosing class).

See this line of code in the `List2Iterator` constructor:

```java
this.current = List2.this.preStart.next;
```
Resources

• Java Libraries API: **Iterable** and **Iterator**
  – [http://docs.oracle.com/javase/8/docs/api/](http://docs.oracle.com/javase/8/docs/api/)