CSSE2002/7023

Semester 2, 2021

Programming in the Large

Week 4.1: Collections continued

In this Session

- List
- Iterators
- Set
- Map
- Sets with custom classes

Lists

Lists hold items in sequential order, much like an array.

- Grow and shrink automatically
- No fixed size limit
- Walk along list
- Insert an item anywhere in the list
- Remove an item anywhere in the list
- Is item in the list?

Types of Lists

`List` is an interface

- Can declare a reference to a list: `List<String>`
- Can't create a list object: ~~`new List<String>`~~

Some possible classes:

- LinkedList – better for ops which modify the middle of the list
  `LinkedList<String>`
- ArrayList – better for random access
  `ArrayList<String>`
- Vector – Mostly for protecting data in concurrent applications[1]
  `Vector<String>`

`ListDemo.java`

[1] `https://geeksforgeeks.org/vector-vs-arraylist-java/`

Iterators

Iterators are a more flexible way to move through a collection than a `for each` loop.

An iterator refers to a position/item in a collection without needing to use an index number.

Most collections support `.iterator()`.

`IteratorDemo.java`

Iterators

Iterators are a more flexible way to move through a collection than a `for each` loop.

An iterator refers to a position/item in a collection without needing to use an index number.

Most collections support `.iterator()`.

`IteratorDemo.java`

Items can be removed from collections via iterators.

`IteratorDemo2.java`

Iterators

Iterators are a more flexible way to move through a collection than a `for each` loop.

An iterator refers to a position/item in a collection without needing to use an index number.

Most collections support `.iterator()`.

`IteratorDemo.java`

Items can be removed from collections via iterators.

`IteratorDemo2.java`

Modifying one iterator, and then trying to use an older iterator, fails fast.

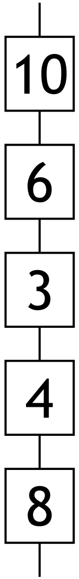
Types of Iterators

`Iterator` is an interface

- Subinterfaces, and their implementing classes, provide behaviour specfic to a context.
  - e.g. `ListIterator`
- Collection objects create and return an iterator to access their items.
  - We don't need to create the iterator for the collection.

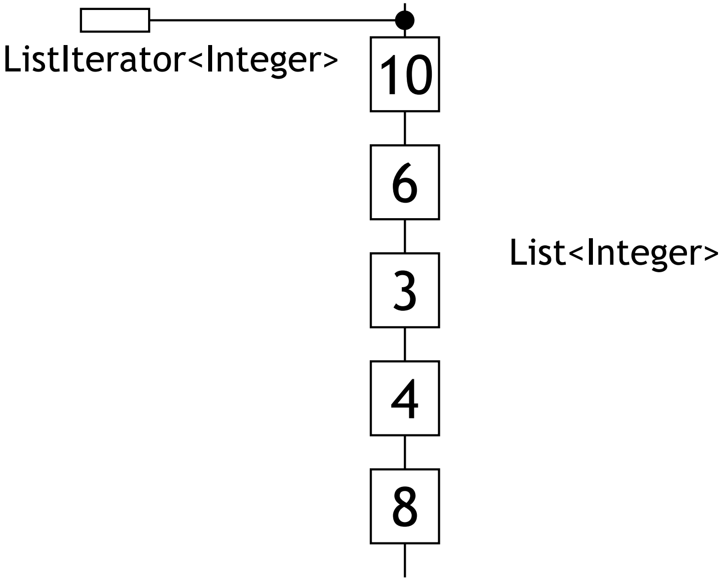
`ListIteratorDemo.java`

Iterators

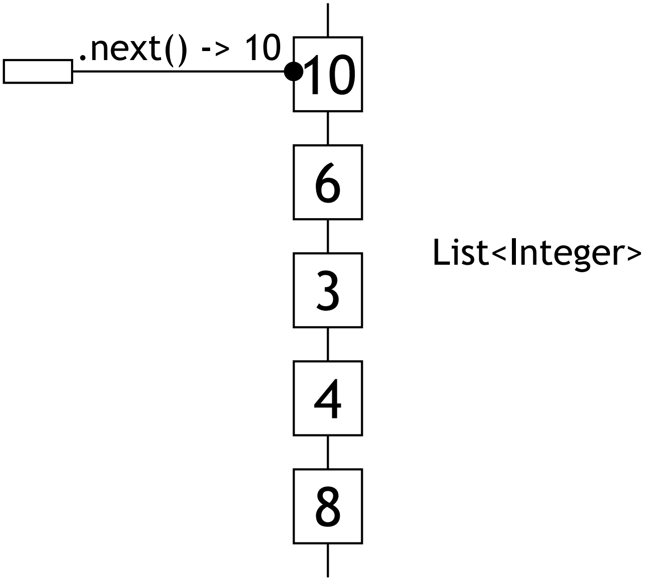


List<Integer>

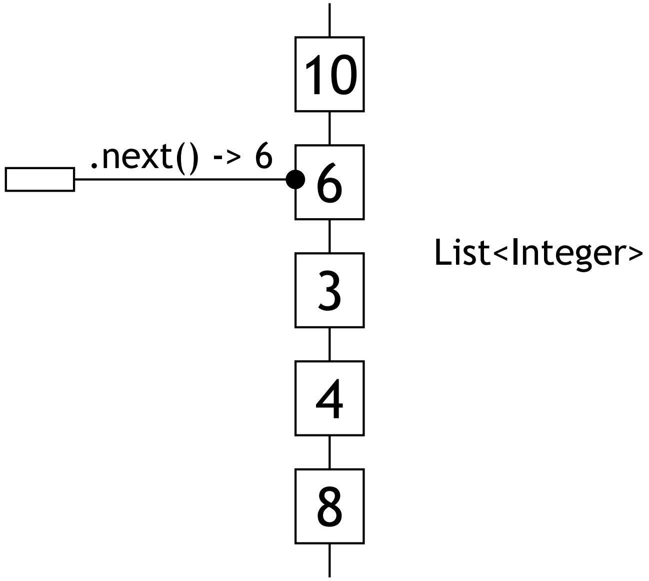
Iterators



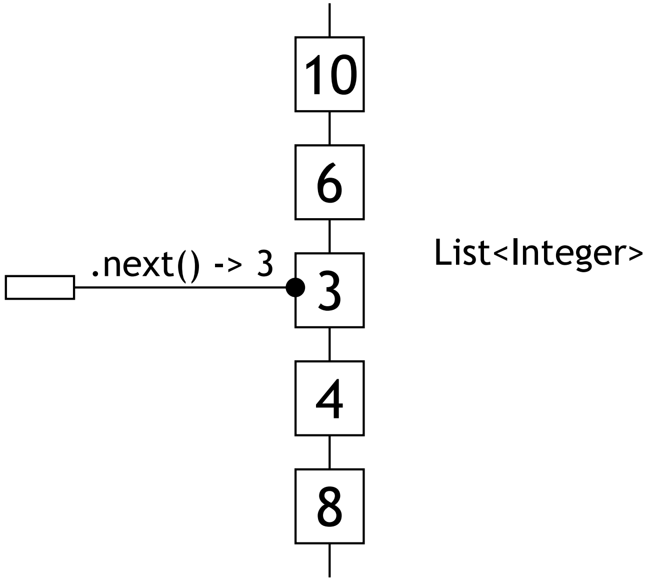
Iterators



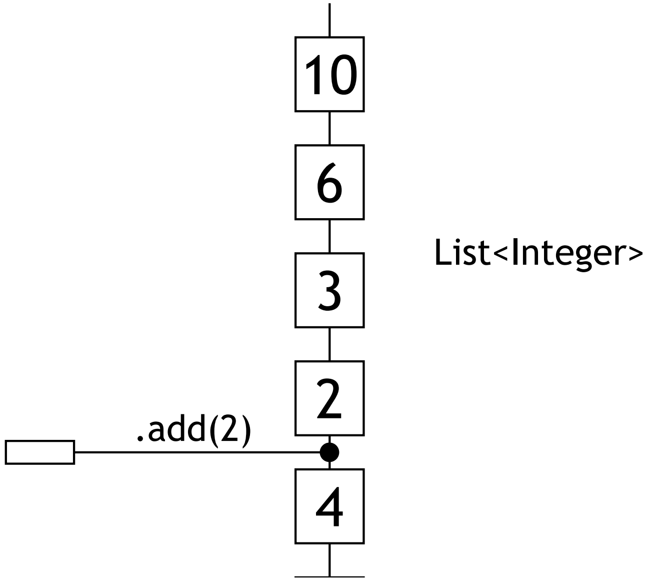
Iterators



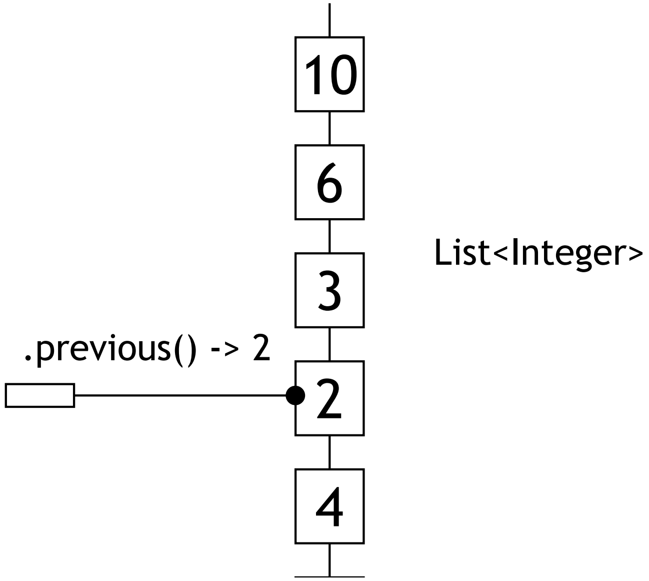
Iterators



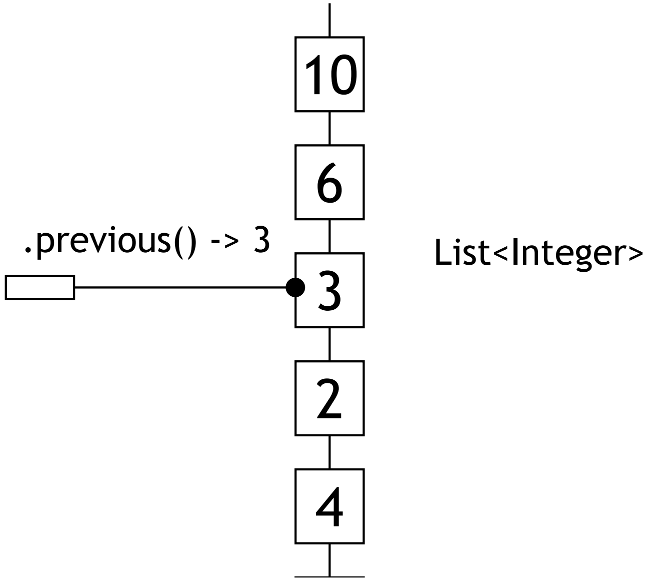
Iterators



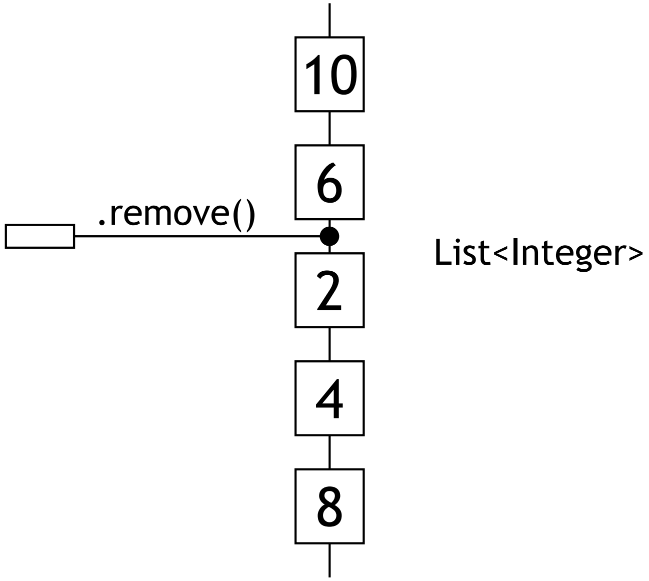
Iterators



Iterators



Iterators



Iterators

- The `ListIterator.remove()` method depends on the last element returned
- The `ListIterator.remove()` method cannot be called twice, or after `ListIterator.add()`
- The `ListIterator` interface is well documented
  - `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/ListIterator.htmll`
    - Get used to navigating and reading the JavaDoc for the standard libraries.

Dangerous Loops

```java
List<Integer> l = new ArrayList<>();
l.add(10);
l.add(8);
l.add(3);
```

1)

```java
for (int i = 0; i < l.size(); i++) {
  l.add(l.get(i) + 1);
}
```

2)

```java
for (Integer i : l) {
  l.add(i + 1);
}
```

3)

```java
int size = l.size();
for (int i = 0; i < size; i++) {
  l.add(l.get(i) + 1);
}
```

4)

```java
ListIterator<Integer> li =
 l.listIterator();
while (li.hasNext()) {
  l.add(li.next() + 1);
}
```

Sets

Sets store unique items (no duplicates).

- Iterate over the set (don't assume ordering)
- Add item to set
- Remove item from set
- Is item in set?

Types of Sets

- `TreeSet<E>` — `E` needs to implement `Comparable`
- `HashSet<E>` — `E` to have sensible `hashCode()` and `equals()`

`SetDemo.java`

Map

Maps store key:value pairs (similar to a Python dictionary).

- Need to specify a type for each

`Map<Integer, String>`

- `Integer` keys and `String` values.

`MapDemo.java`

- Note use of `.entrySet()` to iterate over map contents.

Sets with Custom Classes

(This is largly common with Maps, but Sets are simpler).
`CustSet1.java, ...`

Sets with Custom Classes

(This is largly common with Maps, but Sets are simpler).
`CustSet1.java, ...`

It is not enough that interface functions exist, they need to be consistent with what the code expects.

See the documentation for `Comparable` and `Map` for example.

In summary:

$$\begin{aligned} \text{x.equals(y)} &\Leftrightarrow \text{y.equals(x)} \\ \text{x.equals(y)} &\Rightarrow \text{x.hashCode()} == \text{y.hashCode()} \\ \text{x.compareTo(y)} < 0 &\Leftrightarrow \text{y.compareTo(x)} > 0 \end{aligned}$$

Some Things Not to Do

- **If** you use mutable objects as keys (this should be avoided), do not *change* them once they are in a `Set` or `Map`.
- Do not add a `Map` as a value inside itself.