Java Application Design
Fall 2016 - Week 3

Containers

Weng Kai
http://fm.zju.edu.cn

A personal notebook

- It allows notes to be stored.
- It has no limit on the number of notes it can store.
- It will show individual notes.
- It will tell us how many notes it is currently storing.

Collection

- Collection objects are objects that can store an arbitrary number of other objects.

Introduction to containers

1. **Collection**:
    1. **List**: hold the elements in a particular sequence
    2. **Set**: cannot have any duplicate elements
2. **Map**:a group of key-value object pairs

Printing containers

- toString()
- Example: PrintingContainers.java

Printing containers

- toString()
- Example: PrintingContainers.java

```
[dog, dog, cat]
[cat, dog]
 {cat=Rags, dog=Spot}
```

Container disadvantage: unknown type

- Example: CatsAndDogs.java

Making a type-conscious ArrayList

- Example: MouseList.java

Iterators

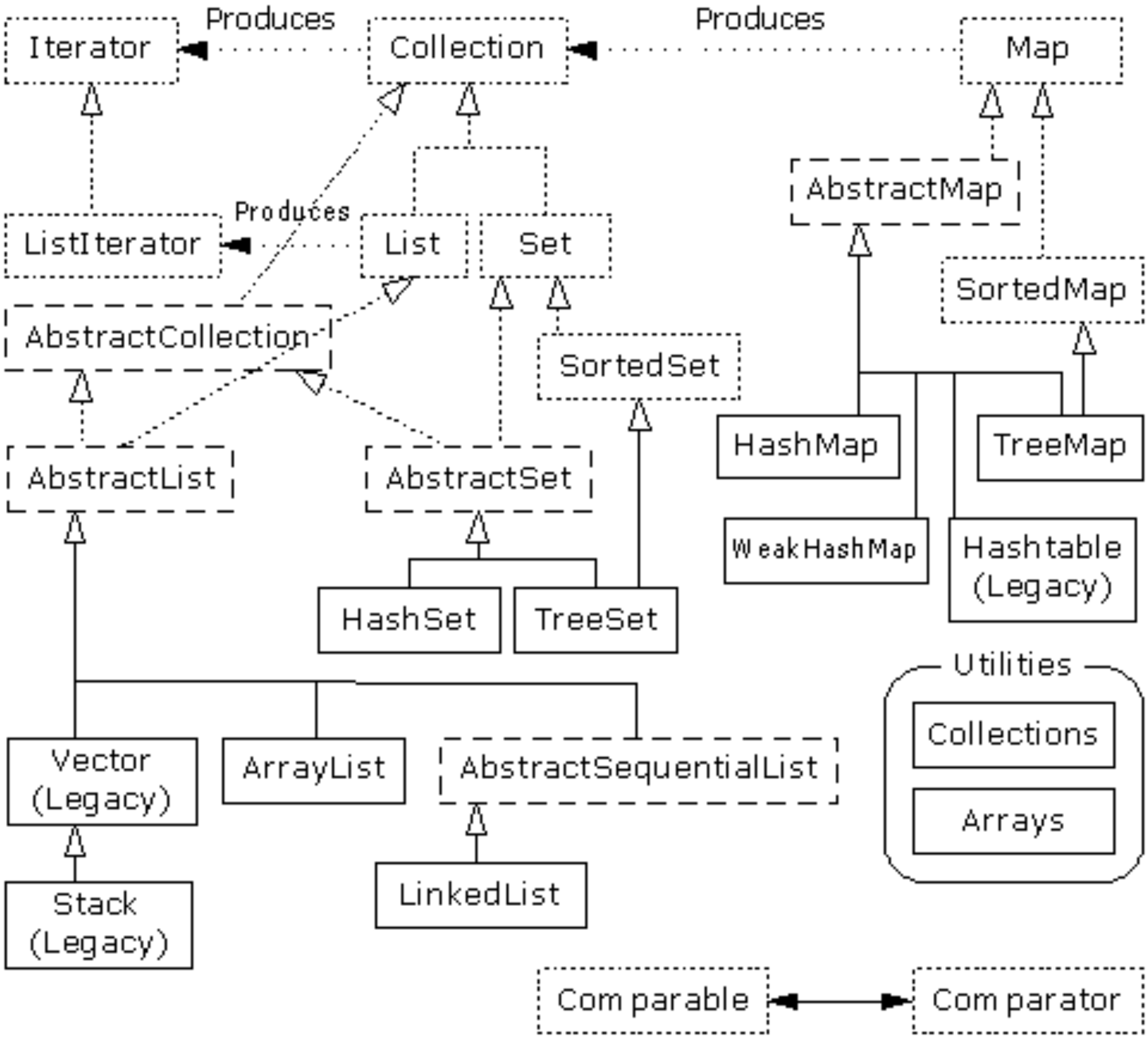
- An iterator is an object whose job is to move through a sequence of objects and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence.

How an iterator acts

1. Ask a container to hand you an **Iterator** using a method called **iterator( )**. This **Iterator** will be ready to return the first element in the sequence on your first call to its **next( )** method.
2. Get the next object in the sequence with **next( )**.
3. See if there *are* any more objects in the sequence with **hasNext( )**.
4. Remove the last element returned by the iterator with **remove( )**.

Example: CatsAndDogs2.java

Container taxonomy



Collection functionality I

- boolean add(Object)
- boolean addAll(Collection)
- void clear( )
- boolean
  contains(Object)
- boolean containsAll(Collection)
- boolean isEmpty( )

Collection functionality II

- Iterator iterator( )
- boolean remove(Object)
- boolean removeAll(Collection)
- boolean retainAll(Collection)
- int size( )
- Object[] toArray( )
- Object[] toArray(Object[] a)

no **get( ) function for random-access element selection*

List

- **List** (interface): maintain elements in a particular sequence
- **ArrayList**: implemented with an array, slow for inserting and removing
- **LinkedList**: inexpensive insertions and deletions, slow for random access

Example: List1.java

Making stack and queue from LinkedList

- Stack: StackL.java
- Queue: Queue.java

Set

- **Set** (interface): Each element that you add to the **Set** must be unique
- **HashSet**: **Object**s must also define **hashCode( )**
- **TreeSet**: An ordered **Set** backed by a tree

Example: Set1.java

Map

- **Map** (interface)
- **HashMap**
- **TreeMap**

Example: Map1.java

Hashing and hash codes

- Key object should have hashCode()

Example: SpringDetector.java

- to use your own classes as keys in a **HashMap**, you must override both **hashCode( )** and **equals( )**,

Example: SpringDetector2.java

Choosing between Lists

Type	get	iteration	insert	Remove
array	1430	3850	Na	Na
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Choosing between Sets

Type	Size	Add	Contains	Iteration
	10	138	115	187
TreeSet	100	190	151	207
	1000	151	177	407
	10	55	82	192
HashSet	100	46	90	202
	1000	36	107	394

Choosing between Maps

Type	size	put	get	Iteration
	10	143	110	186
TreeMap	100	201	188	280
	1000	223	205	407
	10	66	83	197
HashMap	100	81	136	279
	1000	48	106	414
	10	61	93	302
Hashtable	100	91	143	329
	1000	54	111	473

Utilities

- **Static members of Collections**
  - **max(Collection) , min(Collection)**
  - **reverse( )**
  - **copy(List dest, List src)**
  - **fill(List list, Object o)**

generic classes

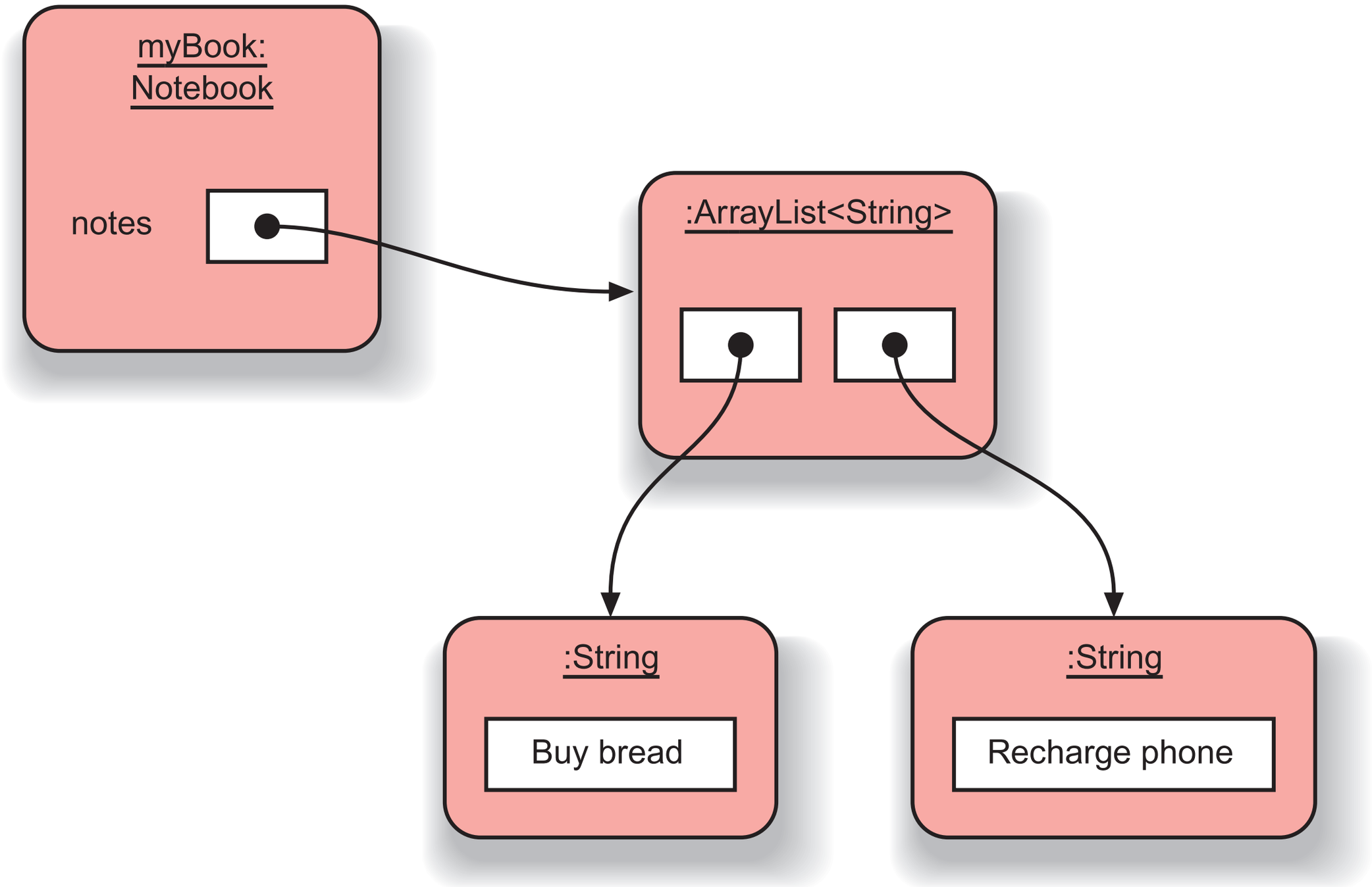
```
private ArrayList<String> notes;
```

- Have to specify two types: the type of the collection itself (here: ArrayList) and the type of the elements that we plan to store in the collection (here: String)

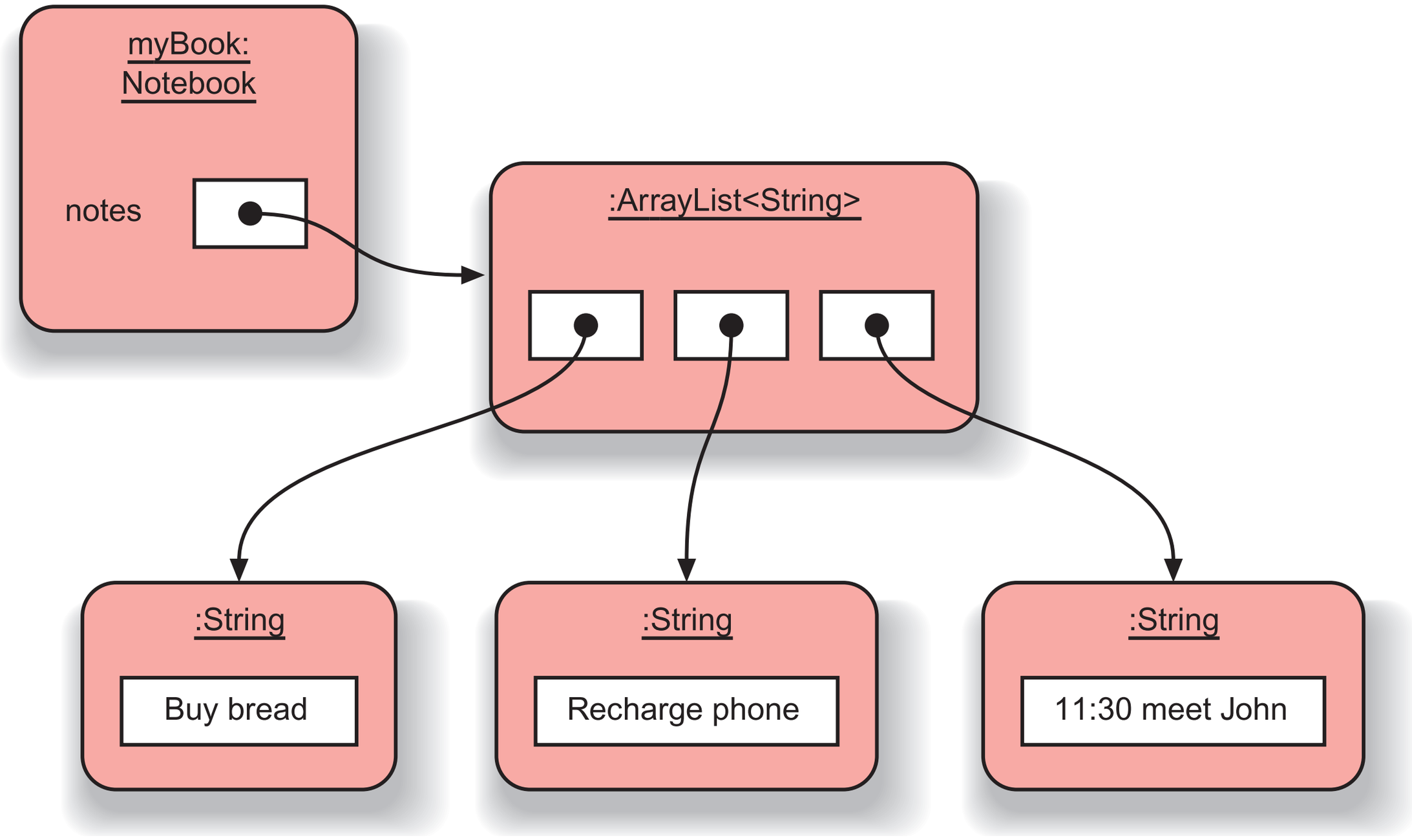
NoteBook.java

```
import java.util.ArrayList;

public class Notebook
{
    private ArrayList<String> notes;

    public Notebook()  {
        notes = new ArrayList<String>();
    }
}
```

Object structure



Object structure



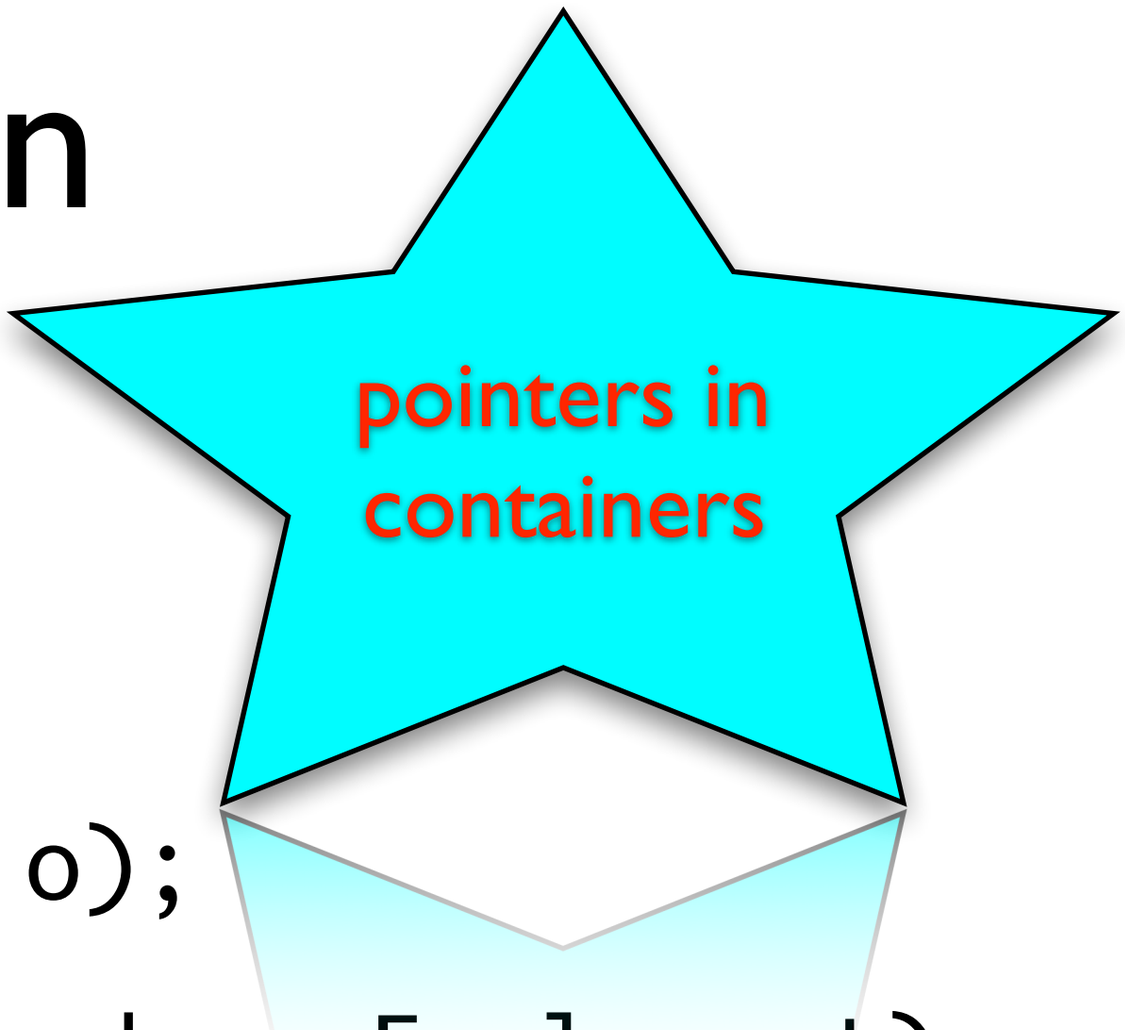
ArrayList

- It is able to increase its internal capacity as required: as more items are added, it simply makes enough room for them.
- It keeps its own private count of how many items it is currently storing. Its size method returns the number of objects currently stored in it.
- It maintains the order of items you insert into it. You can later retrieve them in the same order.

put in

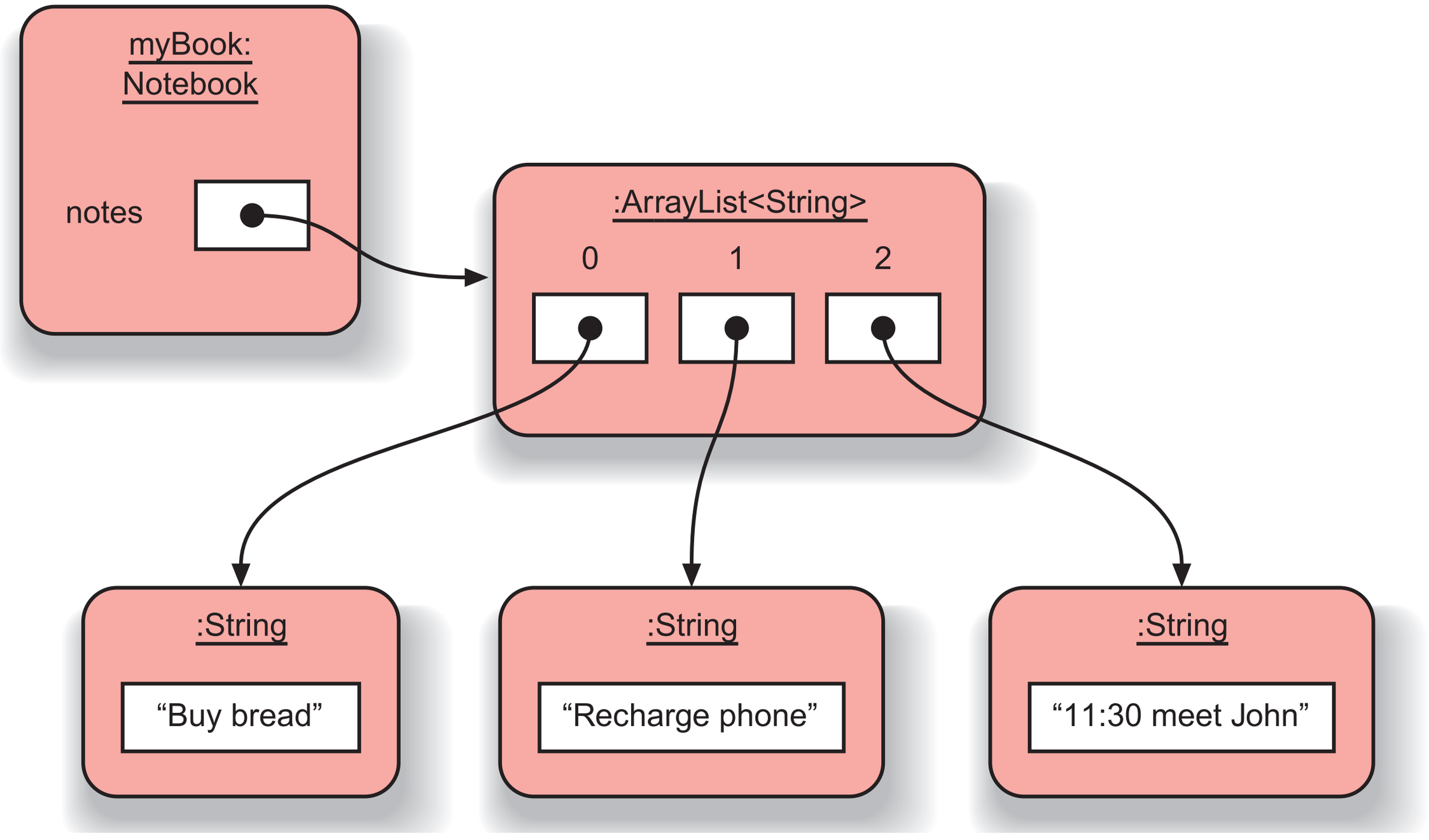
- `public boolean add(E o);`
- `public void add(int index, E element);`

put in

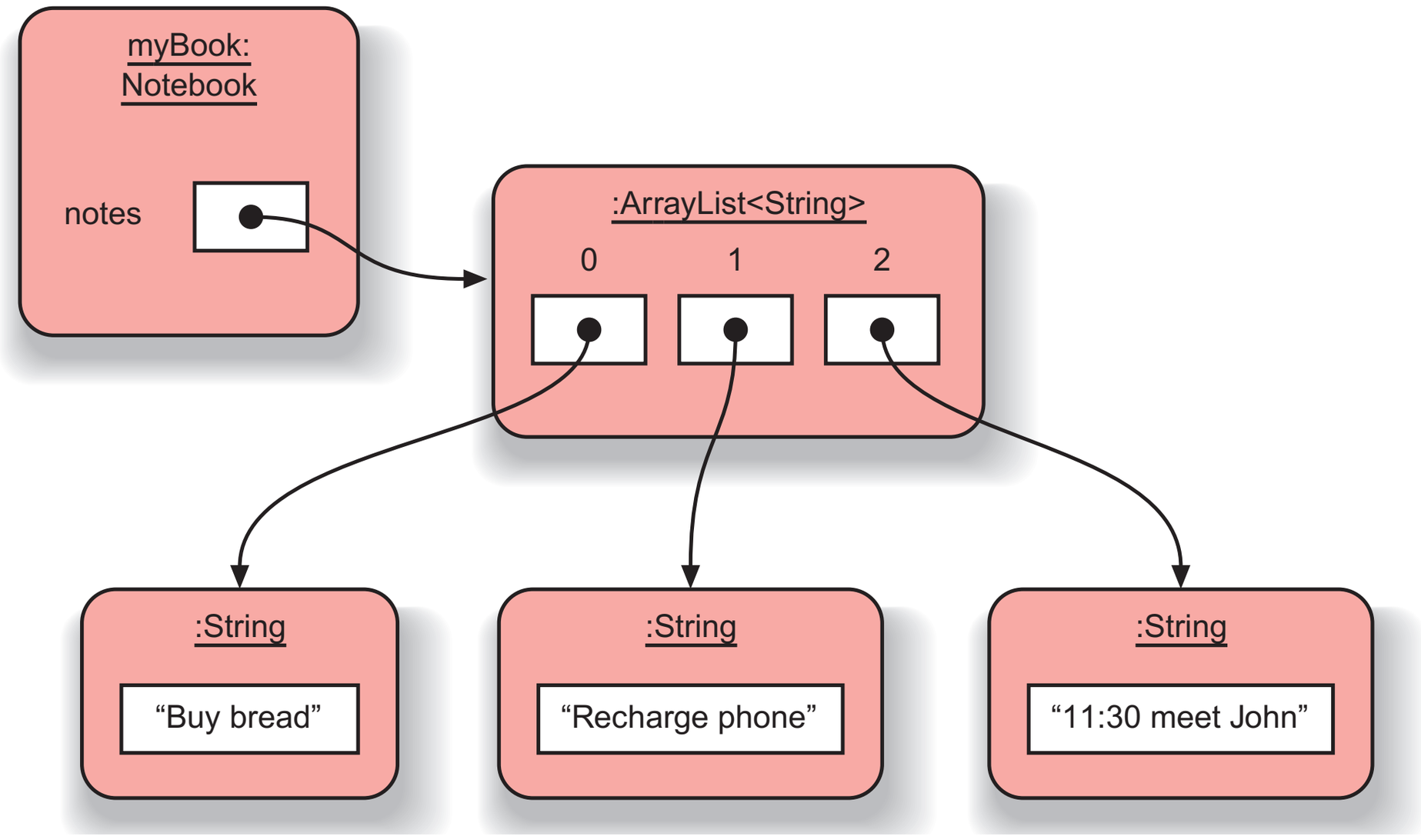


- `public boolean add(E o);`
- `public void add(int index, E element);`

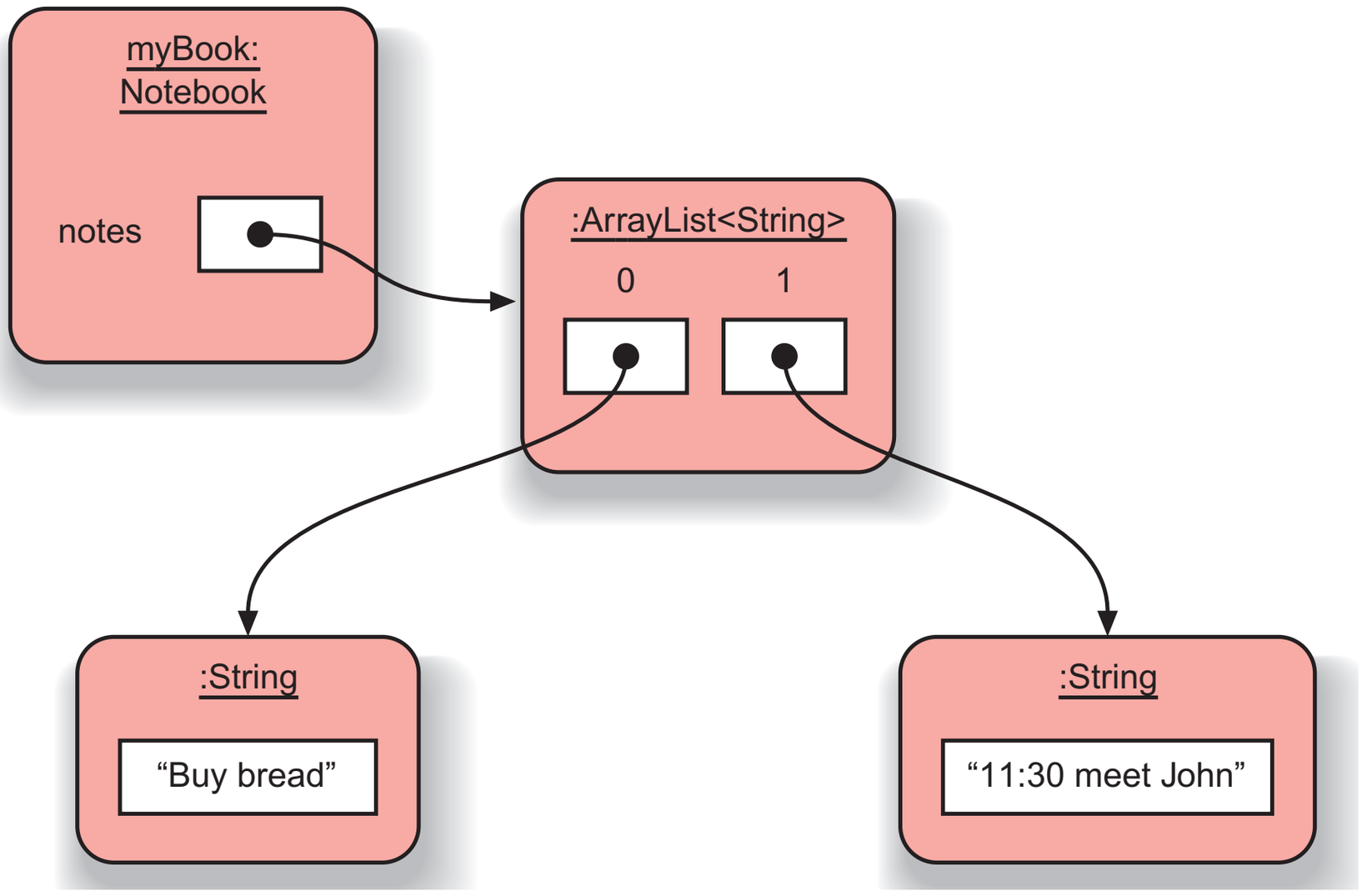
index


myBook:
Notebook
notes
:ArrayList<String>
0
1
2
:String
“Buy bread”
:String
“Recharge phone”
:String
“11:30 meet John”

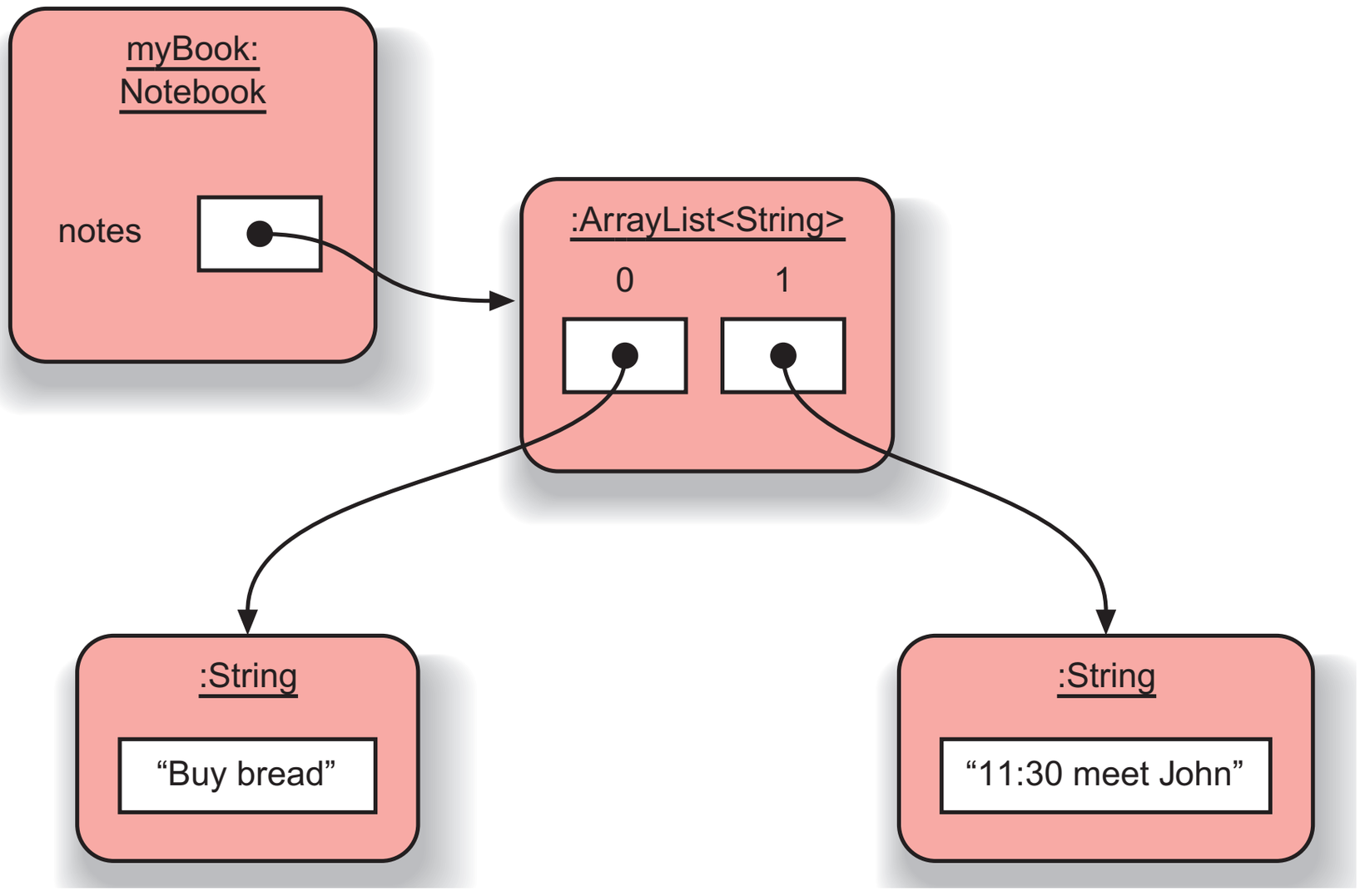
Removing


myBook:
Notebook
notes
:ArrayList<String>
0
1
2
:String
“Buy bread”
:String
“Recharge phone”
:String
“11:30 meet John”

Removing


myBook:
Notebook
notes
:ArrayList<String>
0
1
:String
“Buy bread”
:String
“11:30 meet John”

Removing



- `public boolean remove(Object o);`
- `public E remove(int index);`

Processing a whole collection

- for-each loop

```
for (ElementType element : collection)) {
    loop body
}
```

Processing a whole collection

- for-each loop

```
public voidlistNotes() {
for(String note : notes) {
for (ElementType element :collection){
    System.out.println(note);
}loop body
}
}
```

Processing a whole collection

- for-each loop

```
public voidlistNotes() {
for(String note : notes) {
    System.out.println(note);
}
}
```

for (ElementType element : collection) {

loop body

Can we change what in the container?

while loop

```
int index = 0;
while(index < notes.size()) {
    System.out.println(notes.get(index));
    index++;
}
```

while loop

```
int index = 0;
while(index < notes.size()) {
    System.out.println(notes.get(index));
    index++;
}
```

```
public E get(int index);
```

while loop

```
int index = 0;
    boolean found = false;
    while (index < notes.size() && !found) {
        String note = notes.get(index);
        if (note.contains(searchString)) {
            found = true;
        } else {
            index++;
    }
}
```

Iterator

- An iterator is an object that provides functionality to iterate over all elements of a collection.

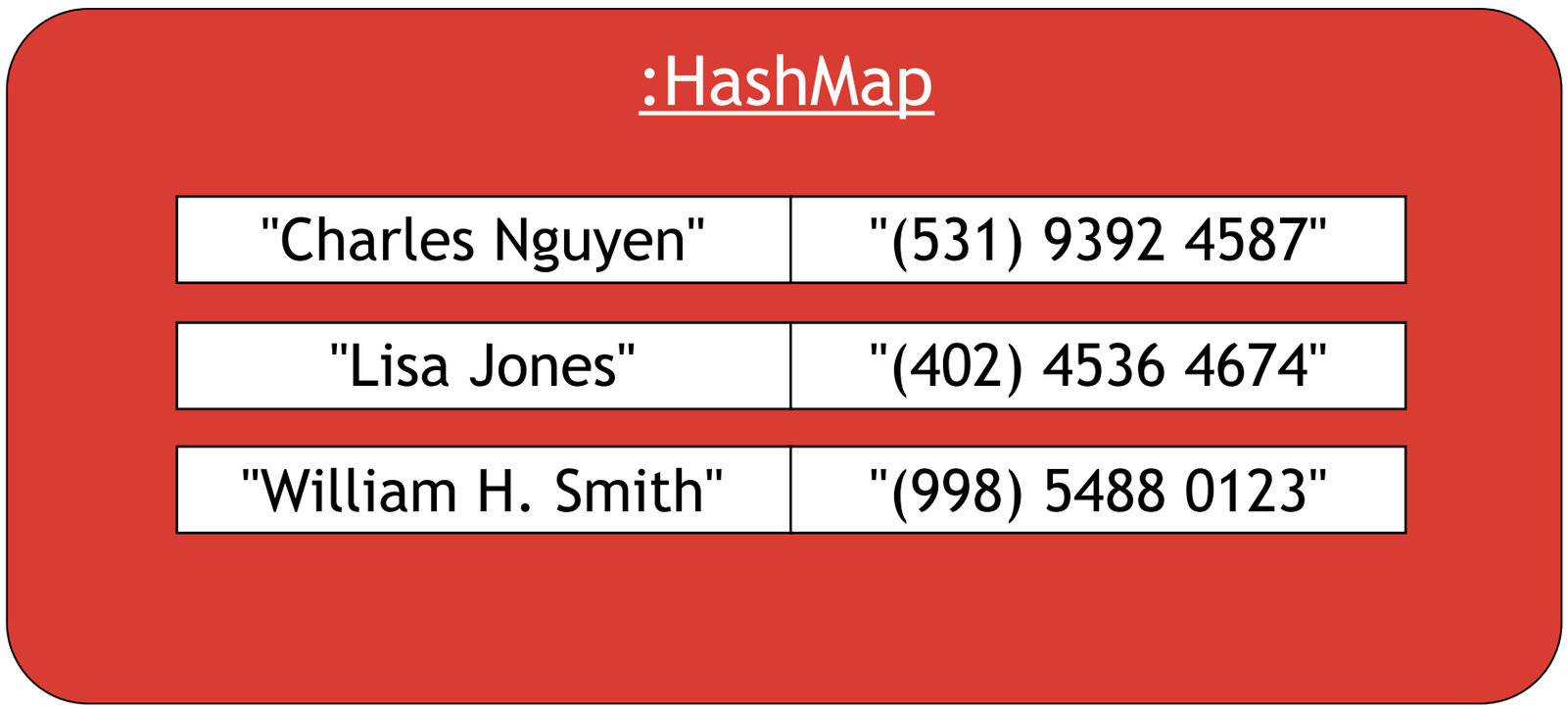
```
public void listNotes() {
    Iterator<String> it = notes.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

Maps

- Maps are collections that contain pairs of values.
- Pairs consist of a key and a value.
- Lookup works by supplying a key, and retrieving a value.
- An example: a telephone book.

Using maps

- A map with Strings as keys and values



Using maps

```
HashMap <String, String> phoneBook =
                new HashMap<String, String>();



phoneBook.put("Charles Nguyen", "(531) 9392 4587");

phoneBook.put("Lisa Jones", "(402) 4536 4674");

phoneBook.put("William H. Smith", "(998) 5488 0123");



String phoneNumber = phoneBook.get("Lisa Jones");

System.out.println(phoneNumber);
```

Using sets

```
import java.util.HashSet;
import java.util.Iterator;
...
HashSet<String> mySet = new HashSet<String>();
mySet.add("one");
mySet.add("two");
mySet.add("three");
Iterator<String> it = mySet.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}
```

Using sets

```
import java.util.HashSet;
import java.util.Iterator;
...
HashSet<String> mySet = new HashSet<String>();
mySet.add("one");
mySet.add("two");
mySet.add("three");
Iterator<String> it = mySet.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}
```

Compare this to ArrayList code!