Java Application Design
Fall 2020

Generic Containers

Weng Kai

Object array

- For object arrays, members are handles
- Case: ArrayClassObj.java

- Case: ArrayInit.java
  Case: VarArgs.java

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

generic classes

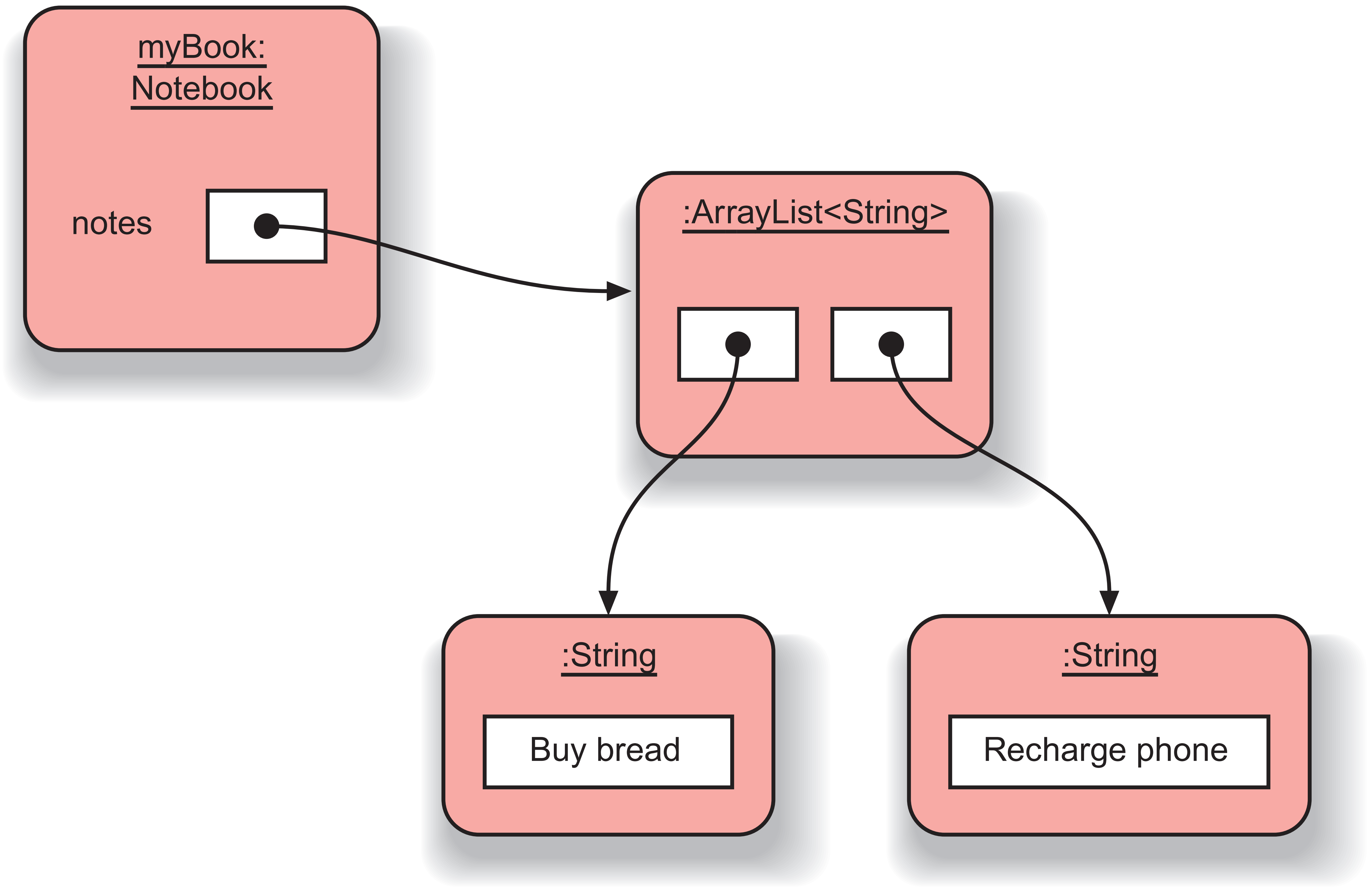
```
private ArrayList<String> notes;
```

- 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)

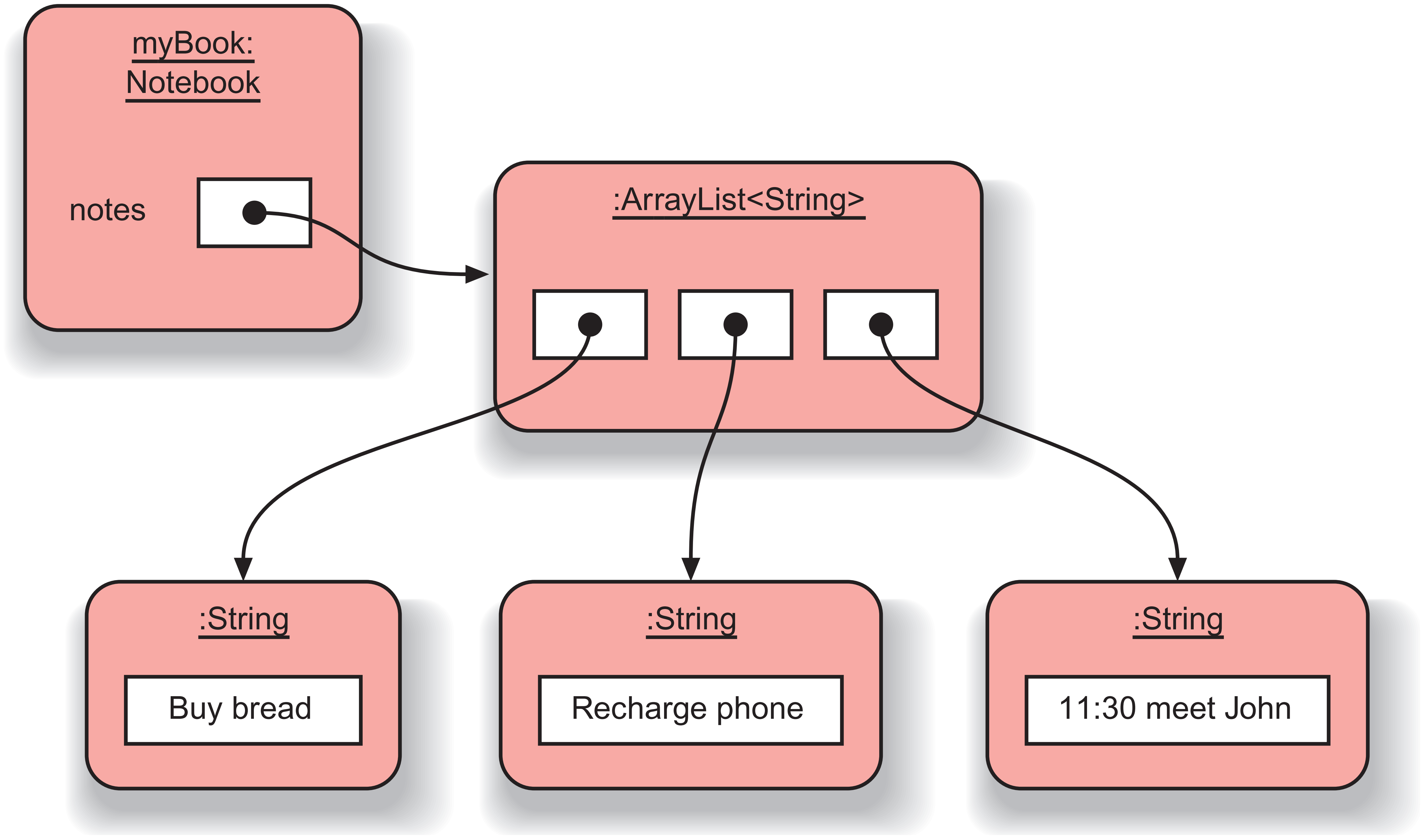
NoteBook.java

```
import java.util.ArrayList;

public class Notebook
{
   private ArrayList<String> notes;

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

Object structure



Object structure



ArrayList

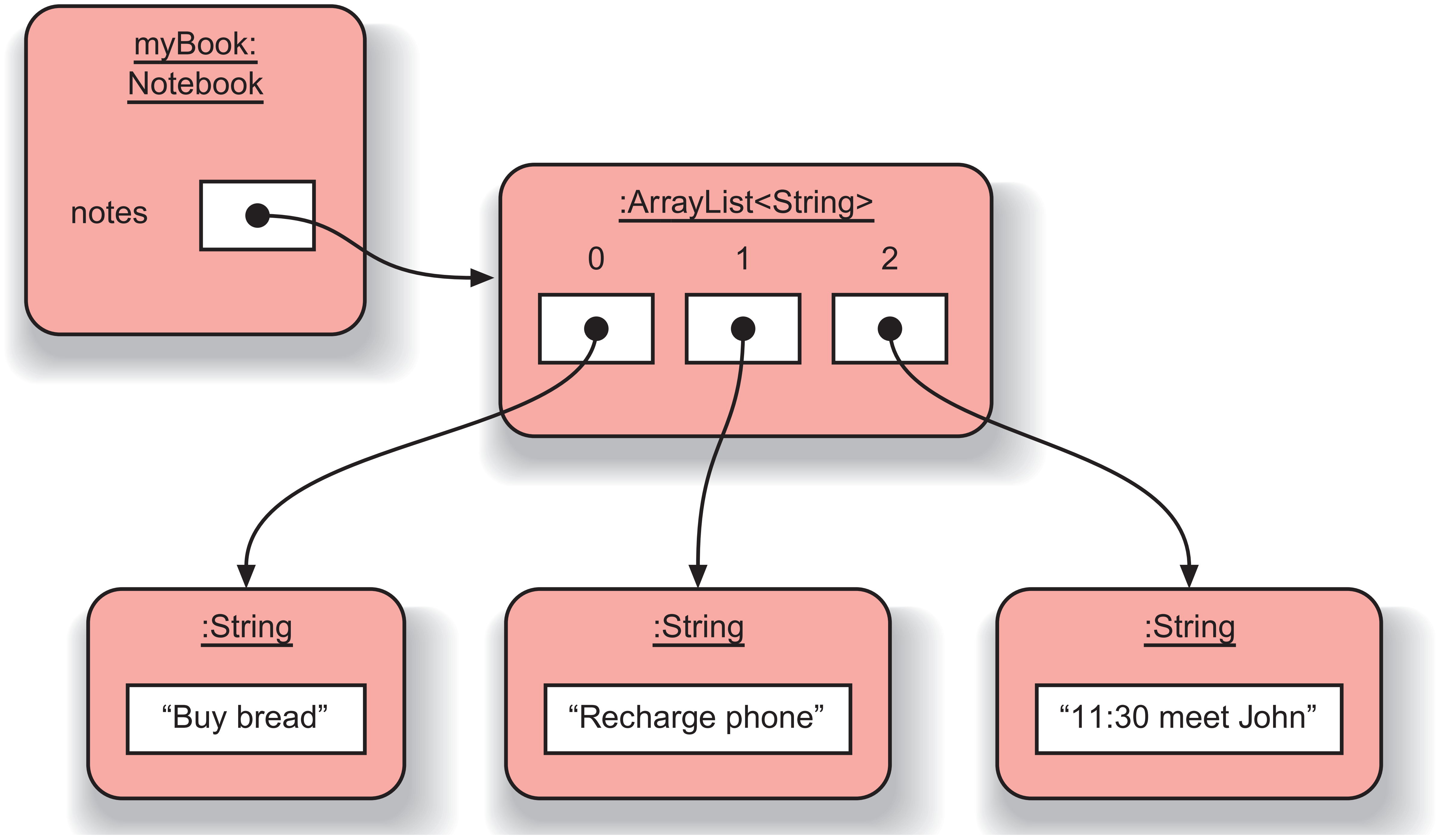
- It is able to increase its internal capacity as required: as more items are added, it simply makes enough room for them.
- It keeps its own private count of how many items it is currently storing. Its size method returns the number of objects currently stored in it.
- It maintains the order of items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

put in

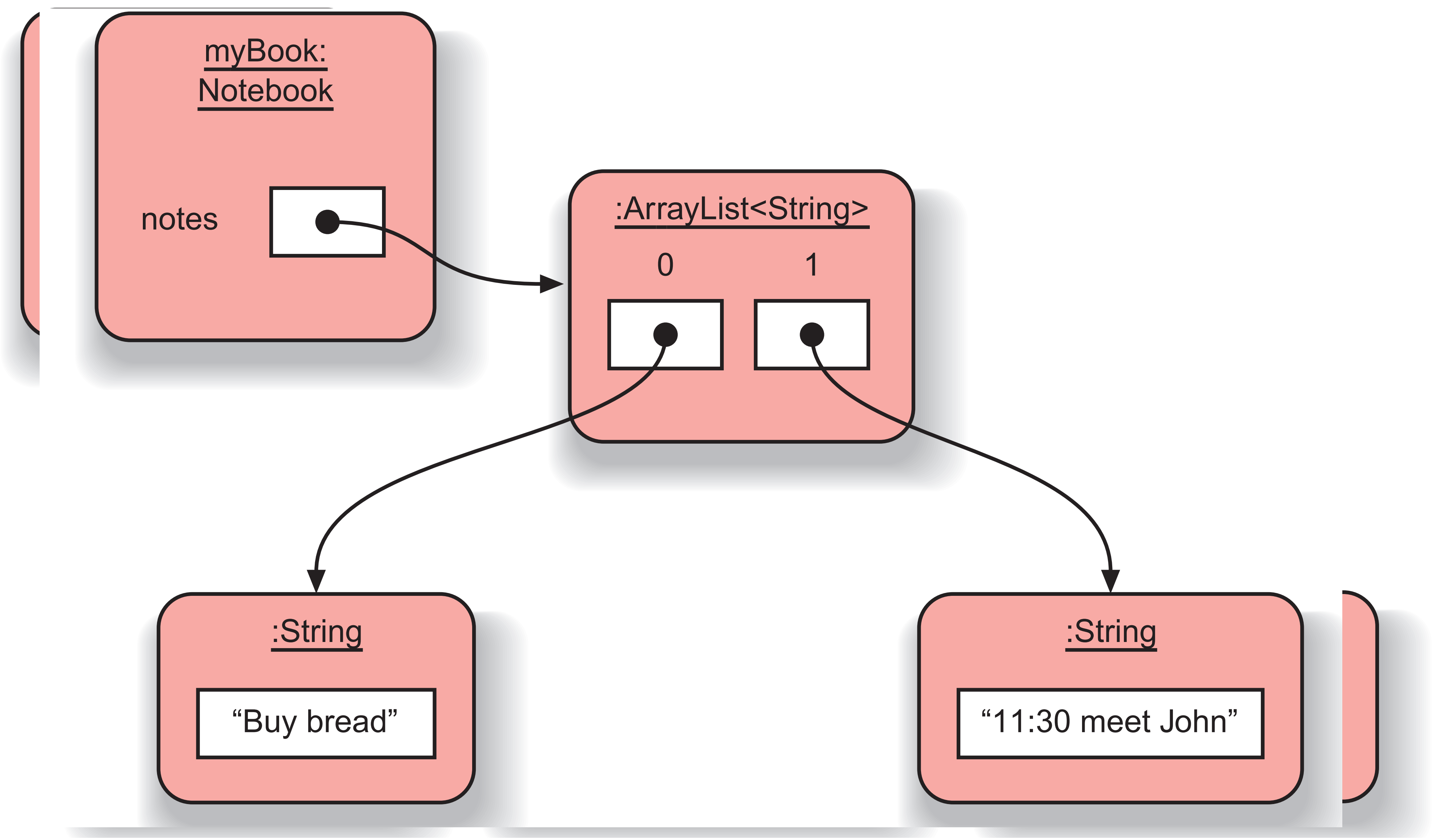
pointers in containers

- `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



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

Processing a whole collection

- for-each loop

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

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

Can we change what in the container?

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());
   }
}
```

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( )**.

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* —

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

Other Methods of ArrayList

- ArrayList(int initialSize);
- ArrayList(Collection<? extends E> c);

- Object[] toArray();
- <T> T[] toArray(T[] a);
  - will generate a new array if a is not capable.

- Array.asList()
  - returns an internal class object that is not ArrayList

LinkedList and Others

- LinkedList is a List that implements Queue and Deque
- LinkedList is a double-linked list internally
- Stack is another class inherited from Vector

Set

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

Example: Set1.java

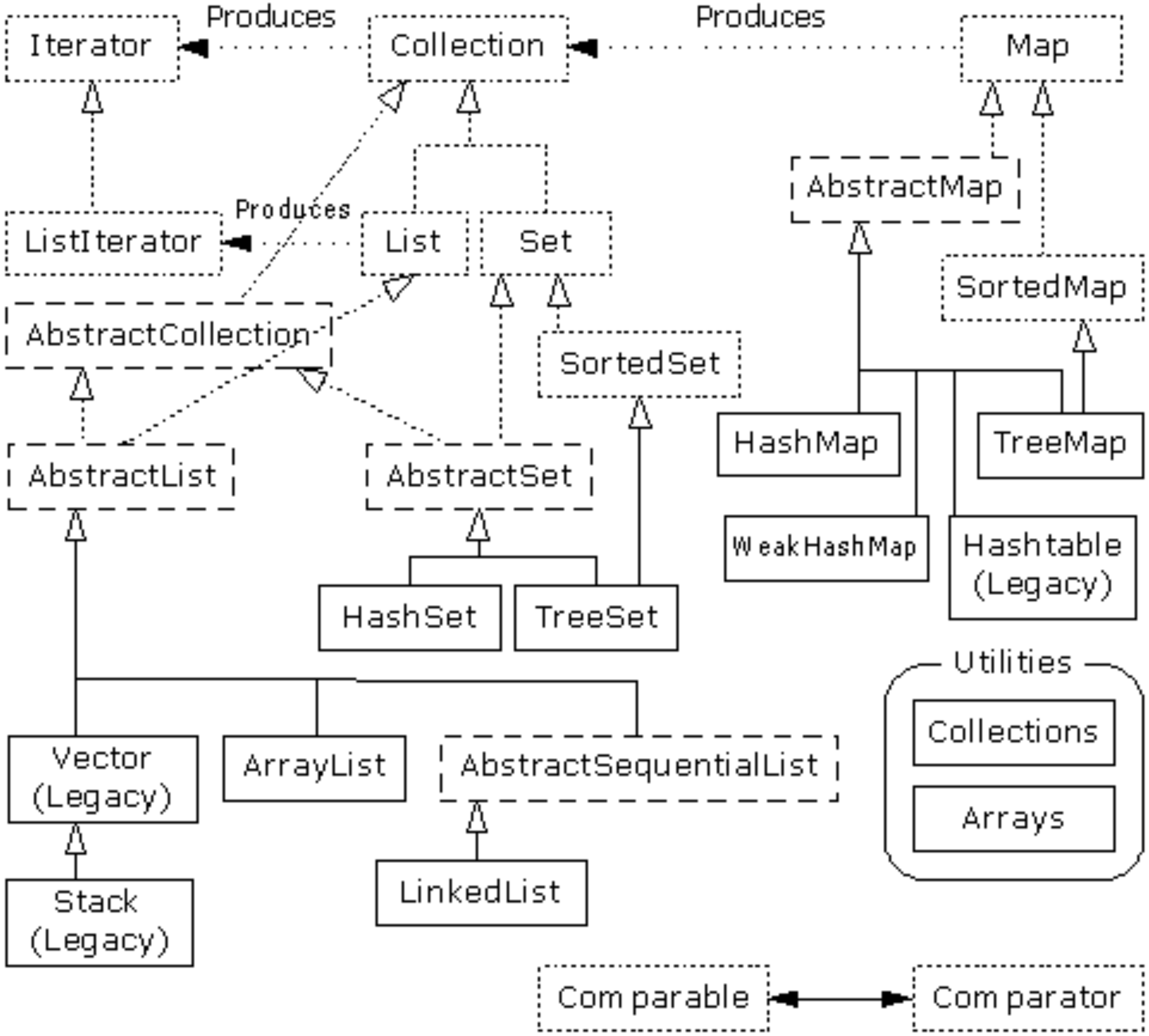
Choosing between Lists

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

Container taxonomy

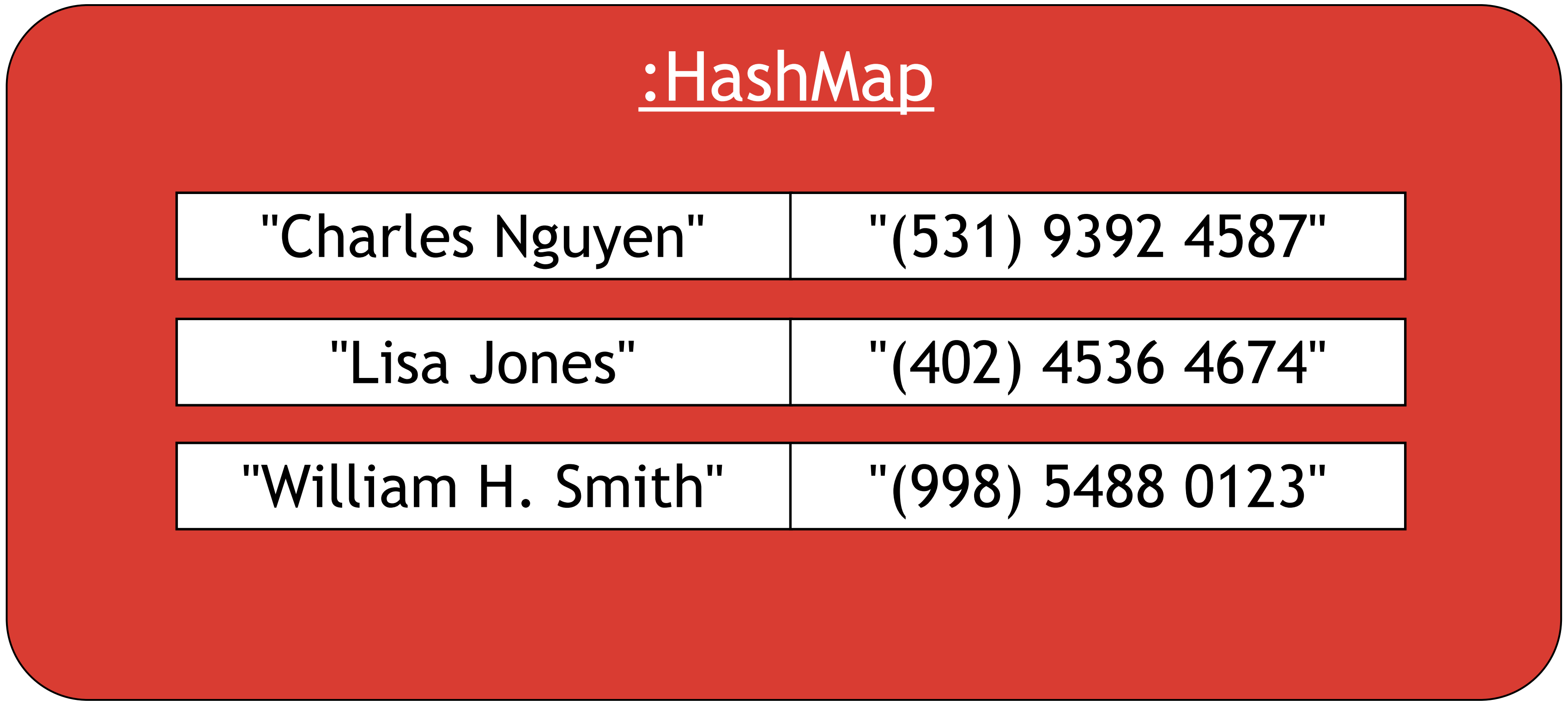


Maps

- Maps are collections that contain pairs of values.
- Pairs consist of a <u>key</u> and a <u>value</u>.
- 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



"Charles Nguyen"	"(531) 9392 4587"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

Map

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

Example: Map1.java

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
}
```

Compare this to ArrayList code!

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 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

Java vs. C++

- What has been put into the collections?

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)

Generics

- The compiler does not check that your cast is the same as the collection's type
- Generics provides a way for you to communicate the type of a collection to the compiler, so that it can be checked

http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

Generic

```
static void expurgate(Collection<String> c)
{
   for (Iterator<String> i = c.iterator();
   i.hasNext(); )
        if (i.next().length() == 4)
                i.remove();
}
```

Defining Simple Generics

```
public interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}
public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

Generic Function

```
public <T> int indexOf(T[] arr, T elm) {
….
}
```

- A generic function does not have to be a function of a generic class, and
- Can have a different generic type if it were a member of a generic class.

- There aren't multiple copies of the code.
- A generic type declaration is compiled once and for all, and turned into a single class file.
- A generic declaration has formal type parameters.
- To use pithy (single character if possible) yet evocative names for formal type parameters.

Generics and Subtyping

- List<String> ls = new ArrayList<String>();
- List<Object> lo = ls;
- lo.add(new Object());
- String s = ls.get(0);

- *In general, if* Foo *is a subtype (subclass or subinterface) of* Bar, *and* G *is some generic type declaration, it is **not** the case that* G<Foo> *is a subtype of* G<Bar>.

Wildcards?

```
void printCollection(Collection c) {
   Iterator i = c.iterator();
   for (k = 0; k < c.size(); k++) {
         System.out.println(i.next());
}}
```

```
void printCollection(Collection<Object> c) {
   for (Object e : c) {
         System.out.println(e);
}}
```

Wildcards

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
      System.out.println(e);
}}
```

Pitfall of wildcards

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());
```

A family of shapes

- **public abstract class** Shape { **public abstract void** draw(Canvas c);}
- **public class** Circle **extends** Shape { **private int** x, y, radius; **public void** draw(Canvas c) { ... }}
- **public class** Rectangle **extends** Shape { **private int** x, y, width, height;**public void** draw(Canvas c) { ... }}

To draw them on a canvas

- **public void** draw(Shape s) {

      s.draw(**this**);}

- **public void** drawAll(List<Shape> shapes) {

      for (Shape s: shapes) {

          s.draw(**this**);}}

Bounded Wildcards

- **public void** drawAll(List<? **extends** Shape> shapes) { ... }

Pitfall of bounded wc

```
public void addRectangle(List<? extends
   Shape> shapes) {
        shapes.add(0, new Rectangle());
}
```

Generic Methods

- **static void** fromArrayToCollection(Object[] a, Collection<?> c) {
```
      for (Object o : a) {
      c.add(o);
 }}
```
- **static** <T> **void** fromArrayToCollection(T[] a, Collection<T> c) {
```
      for (T o : a) {
            c.add(o);
 }}
```

Wildcard version

- **interface** Collection<E> {

  public boolean containsAll (Collection<?> c);

  public boolean addAll (Collection<? **extends** E> c);

  }

Generic version

-
  ```
  interface Collection<E> {
      public <T> boolean containsAll
      (Collection<T> c);
      public <T extends E> boolean
      addAll(Collection<T> c);
  }
  ```

Wildcards vs. Generic

- Wildcards are designed to support flexible subtyping.
- Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type.

Wildcards and Generic

- **class** Collections {

```
      public static <T> void copy (
      List<T> dest,
              List<? extends T> src)
      {…}
  }
```

public static <T, S **extends** T> **void** copy(List<T> dest, List<S> src){...}

Interoperating with Legacy Code

- Using legacy code from within generic code, and using generic code within legacy code.

Using Legacy Code in Generic Code

- **void** addAssembly(String name, Collection parts)
- Collection getParts();
- Collection<Part> c = new ArrayList<Part>();
- addAssembly(”thingee”, c);
- Collection<Part> k = getParts();

Using Generic Code in Legacy Code

- **void** addAssembly(String name, Collection<Part> parts) {...}
- Collection<Part> getParts();
- Collection c = new ArrayList();
- Inventory.addAssembly(”thingee”, c);
- Collection k = (”thingee”).getParts();

A Generic Class is Shared by all its Invocations

- List <String> l1 = new ArrayList<String>();
- List<Integer> l2 = new ArrayList<Integer>();
- System.out.println(l1.getClass() == l2.getClass());

Casts and InstanceOf

- Collection cs = new ArrayList<String>();
  if (cs **instanceof** Collection<String>) { ...} *// illegal*
- Collection<String> cstr = (Collection<String>) cs;
- <T> T badCast(T t, Object o) {**return** (T) o; *// unchecked warning*
- *Type variables don't exist at run time. This means that they entail no performance overhead in either time nor space, which is nice. Unfortunately, it also means that you can't reliably use them in casts.*

Arrays

- The component type of an array object may not be a type variable or a parameterized type, unless it is an (unbounded) wildcard type.

No to array

```
List<String>[] lsa = new List<String>[10]; // not really
    allowed
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // unsound, but passes run time store check
String s = lsa[1].get(0); // run-time error -
    ClassCastException
```

Yes to array

```
List<?>[] lsa = new List<?>[10]; // ok, array of unbounded wildcard type
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = (String) lsa[1].get(0); // run time error, but cast is explicit
```

Class Literals as Run-time Type Tokens

- java.lang.Class is generic.
- T stands for the type that the Class object is representing.
  - <T> Collection<T> select(Class<T>c, String sqlStatement) {

```
      T item = c.newInstance();
    }
```

Define a Generic Class

```
public class ObjectHolder<T> {
    private T obj;
    public T getObject() { return obj; }
    public void setObject(T obj) { this.obj = obj; }
}
```

- All types, except enum, anonymous inner classes and Exception classes, can have generic parameters.
- More than one: MyClass<S, U,V>
- T can not be used to create object or array, be as parent class, in instanceof...
- May not be specified a type