CSE 143
Lecture 4

Exceptions and `ArrayList`

slides created by Marty Stepp
http://www.cs.washington.edu/143/

Preconditions

- **precondition**: Something your method *assumes is true* at the start of its execution.
  - Often documented as a comment on the method's header:

```
// Returns the element at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
    return elementData[index];
}
```

  - Stating a precondition doesn't really "solve" the problem
    - Clients don't always follow directions.
  - Precondition is non-binding
    - What if we want to actually enforce the precondition?

2

Bad precondition test

- What is wrong with the following way to handle violations?

```
// Returns the element at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
    if (index < 0 || index >= size) {
        System.out.println("Bad index! " + index);
        return -1;
    }
    return elementData[index];
}
```

  - -1 could be a legal value
  - `println` is not a very strong deterrent to the client (esp. GUI)

3

Throwing exceptions (4.4)

```
throw new ExceptionType();
throw new ExceptionType("message");
```

- Causes the program to immediately crash with an exception.

- Common exception types:
  - ArithmeticException, ArrayIndexOutOfBoundsException, FileNotFoundException, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, IOException, NoSuchElementException, NullPointerException, RuntimeException, UnsupportedOperationException

- Why would anyone ever *want* the program to crash?

4

Exception example

```
public int get(int index) {
    if (index < 0 || index >= size) {
        String msg = "Out of list bounds: " + index;
        throw new IndexOutOfBoundsException(msg);
    }
    return elementData[index];
}
```

- Exercise: Modify the rest of `ArrayIntList` to state preconditions and throw exceptions as appropriate.

5

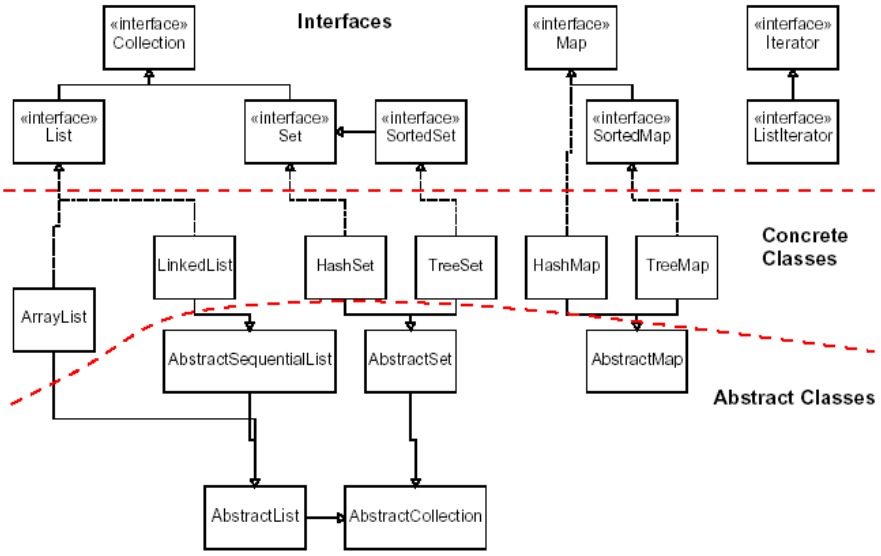
Collections

- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*
  - examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
  - all collections are in the `java.util` package

```
import java.util.*;
```

6

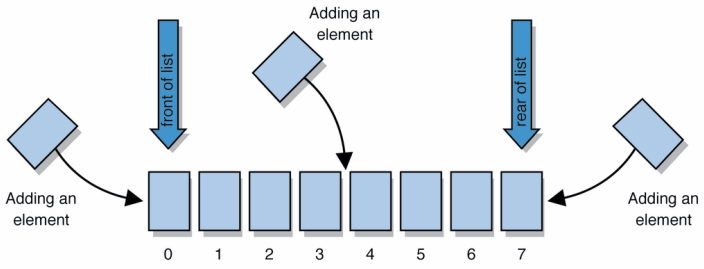
Java collection framework



7

Lists

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere



8

Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

```
[]
```

- You can add items to the list.
  - The default behavior is to add to the end of the list.

```
[hello, ABC, goodbye, okay]
```

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Internally, the list is implemented using an array and a size field.
  - Think of an "array list" as an automatically resizing array object.

9

`ArrayList` methods (10.1)

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

10

`ArrayList` methods 2

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list (seen later)
`lastIndexOf` `(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

11

Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `<` and `>`.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same `ArrayList` class to store lists of different types.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

12

Learning about classes

- The Java API Specification is a huge web page containing documentation about every Java class and its methods.
  - A link to the API Specs is on the course web site under "Links"

`ArrayList` vs. array

- construction

```
String[] names = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- storing a value

```
names[0] = "Jessica";
list.add("Jessica");
```

- retrieving a value

```
String s = names[0];
String s = list.get(0);
```

ArrayList vs. array 2

- doing something to each value that starts with "B"

```
for (int i = 0; i < names.length; i++) {
    if (names[i].startsWith("B")) { ... }
}
```

```
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}
```

- seeing whether the value "Benson" is found

```
for (int i = 0; i < names.length; i++) {
    if (names[i].equals("Benson")) { ... }
}
```

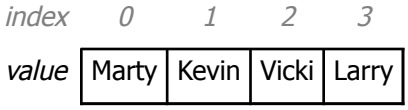
```
if (list.contains("Benson")) { ... }
```

15

Out-of-bounds

- Legal indexes are between **0** and the **list's size() - 1**.
  - Reading or writing any index outside this range will cause an IndexOutOfBoundsException.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty");    names.add("Kevin");
names.add("Vicki");    names.add("Larry");
System.out.println(names.get(0));        // okay
System.out.println(names.get(3));        // okay
System.out.println(names.get(-1));       // exception
names.add(9, "Aimee");                   // exception
```



16

Exercise

```
public static void main(String[] args) {
   ArrayList<String> animalList = new ArrayList<String>();
   animalList.add("aardvark");
   animalList.add("bears");
   animalList.add("cat");
   animalList.add("dogs");
   animalList.add("elephant");
   animalList.add("frogs");
   animalList.add("goats");

   System.out.println("List: " + animalList);

   // write code to remove Strings ending with 's'


   System.out.println("Modified list: " + animalList);
}
```

17

Solution

```
for (int i = 0; i < animalList.size(); i++) {
   String animal = animalList.get(i);
   if (animal.endsWith("s")) {
      animalList.remove(i);
      i--;
   }
}
```

18

`ArrayList` of primitives?

- The type you specify when creating an `ArrayList` must be an object type; it cannot be a primitive type.

```
// illegal -- int cannot be a type parameter
ArrayList<int> list = new ArrayList<int>();
```

- But we can still use `ArrayList` with primitive types by using special classes called *wrapper* classes in their place.

```
// creates a list of ints
ArrayList<Integer> list = new ArrayList<Integer>();
```

19

Wrapper classes

Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

- A wrapper is an object whose sole purpose is to hold a primitive value.
- Once you construct the list, use it with primitives as normal:

```
ArrayList<Double> grades = new ArrayList<Double>();
grades.add(3.2);
grades.add(2.7);
...
double myGrade = grades.get(0);
```

20

ArrayList "mystery"

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i = 1; i <= 10; i++) {
    list.add(10 * i);   // [10, 20, 30, 40, ..., 100]
}
```

- What is the output of the following code?

```
for (int i = 0; i < list.size(); i++) {
    list.remove(i);
}
System.out.println(list);
```

- Answer:

```
[20, 40, 60, 80, 100]
```

21

ArrayList "mystery" 2

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) {
    list.add(2 * i);   // [2, 4, 6, 8, 10]
}
```

- What is the output of the following code?

```
int size = list.size();
for (int i = 0; i < size; i++) {
    list.add(i, 42);   // add 42 at index i
}
System.out.println(list);
```

- Answer:

```
[42, 42, 42, 42, 42, 2, 4, 6, 8, 10]
```

22

`ArrayList` as parameter

```
public static void name(ArrayList<Type> name) {
```

- Example:

```
// Removes all plural words from the given list.
public static void removePlural(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        String str = list.get(i);
        if (str.endsWith("s")) {
            list.remove(i);
            i--;
        }
    }
}
```

- You can also return a list:

```
public static ArrayList<Type> methodName(params)
```


Exercise

- Write a method `addStars` that accepts an array list of strings as a parameter and places a `*` before each element.
  - Example: if an array list named `list` initially stores:

    `[the, quick, brown, fox]`

  - Then the call of `addStars(list);` makes it store:

    `[*, the, *, quick, *, brown, *, fox]`

- Write a method `removeStars` that accepts an array list of strings, assuming that every other element is a `*`, and removes the stars (undoing what was done by `addStars` above).

Solution

```
public static void addStars(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i += 2) {
        list.add(i, "*");
    }
}


public static void removeStars(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        list.remove(i);
    }
}
```

25

Exercise

- Write a method `intersect` that accepts two sorted array lists of integers as parameters and returns a new list that contains only the elements that are found in both lists.
  - Example: if lists named `list1` and `list2` initially store:

    `[1, 4, 8, 9, 11, 15, 17, 28, 41, 59]`

    `[4, 7, 11, 17, 19, 20, 23, 28, 37, 59, 81]`
  - Then the call of `intersect(list1, list2)` returns the list:

    `[4, 11, 17, 28, 59]`

26

Solution

```
public static ArrayList<Integer> intersect(ArrayList<Integer> list1,
                                           ArrayList<Integer> list2) {
   ArrayList<Integer> result = new ArrayList<Integer>();

   int index1 = 0;
   int index2 = 0;
   while (index1 < list1.size() && index2 < list2.size()) {
      if (list1.get(index1) < list2.get(index2)) {
         index1++;
      } else if (list1.get(index1) > list2.get(index2)) {
         index2++;
      } else {
         result.add(list1.get(index1));
         index1++;
         index2++;
      }
   }

   return result;
}
```


More exercises

- Write a method `reverse` that reverses the order of the elements in an `ArrayList` of strings.
- Write a method `capitalizePlurals` that accepts an `ArrayList` of strings and replaces every word ending with an "s" (or "S") with its uppercased version.

More solutions

```
public static void reverse(ArrayList<String> list) {
   for (int i = 0; i < list.size() / 2; i++) {
      int oppositeIndex = list.size() - 1 - i;
      String temp = list.get(i);
      list.set(i, list.get(oppositeIndex));
      list.set(oppositeIndex, temp);
   }
}
```

```
public static void capitalizePlurals(ArrayList<String> list) {
   for (int i = 0; i < list.size(); i++) {
      if (list.get(i).toLowerCase().endsWith("s")) {
         list.set(i, list.get(i).toUpperCase());
      }
   }
}
```

29