Lists

Chapter 12

Contents

Objectives

- Describe the ADT list
- Use the ADT list in a Java program

Lists

- A collection
  - Has order … which may or may not matter
  - Additions may come anywhere in list

Figure 12-1 A to-do list

Lists

- Typical actions with lists
    - Add item at end (or anywhere)
    - Remove an item (or all items)
    - Replace an item
    - Look at an item (or all items)
    - Search *for* an entry
    - Count how many items in the list
    - Check if list is empty

ADT List

- Data
  - A collection of objects in a specific order and having the same data type
  - The number of objects in the collection
- Operations
  - add(newEntry)
  - add(newPosition, newEntry)
  - remove(givenPosition)

    …

ADT List

- Operations (ctd.)
  - clear()
  - replace(givenPosition, newEntry)
  - getEntry(givenPosition)
  - contains(anEntry)
  - getLength()
  - isEmpty()
  - toArray()

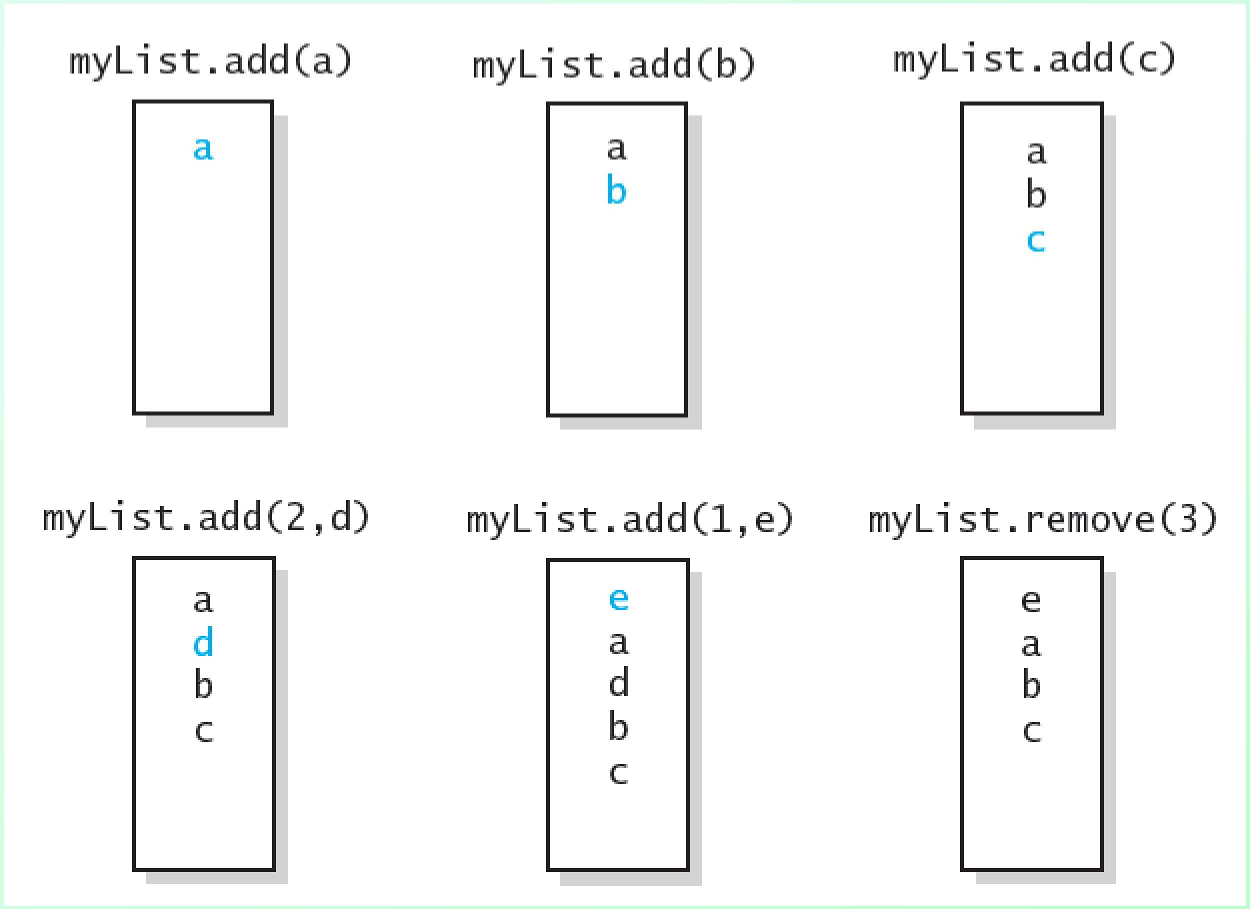

Figure 12-2 The effect of ADT list operations on an initially empty list

Question 1 Write pseudocode statements that add some objects to a list, as follows. First add c, then a, then b, and then d, such that the order of the objects in the list will be a, b, c, d.

Question 2 Write pseudocode statements that exchange the third and seventh entries in a list of 10 objects.

Question 1 Write pseudocode statements that add some objects to a list, as follows. First add c, then a, then b, and then d, such that the order of the objects in the list will be a, b, c, d.

```
myList.add(c)
myList.add(1, a)
myList.add(2, b)
myList.add(4, d)
```

Question 2 Write pseudocode statements that exchange the third and seventh entries in a list of 10 objects.

```
seven = myList.remove(7)
three = myList.remove(3)
myList.add(3, seven)
myList.add(7, three)
```

Another solution:

```
seven = myList.getEntry(7)
three = myList.getEntry(3)
myList.replace(3, seven)
myList.replace(7, three)
```

List

- View list interface, Listing 12-1
- Using the ADT List
  - Don't need to know *ho*
  - Only need to know *wha*

Note: Code listing files must be in same folder as PowerPoint files for links to work

on

- Consider keeping list of finishers of a running race
  - View client code, Listing 12-2
  - Output

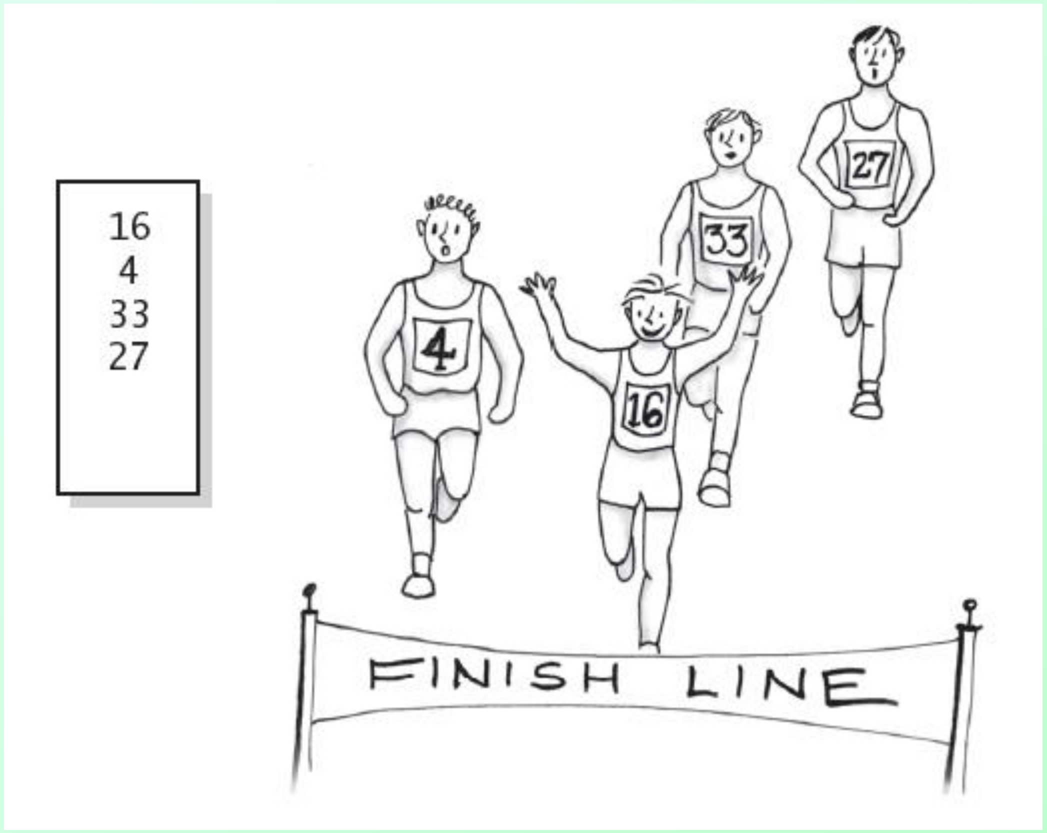

Figure 12-3 A list of numbers that identify runners in the order in which they finished a race

Question 3 In the previous example, what changes to testList are necessary to represent the runner's numbers as Integer objects instead of strings?

```
ListInterface<String> runnerList = new  AList<String>();
// runnerList has only methods in ListInterface
runnerList.add("16"); // winner
runnerList.add(" 4"); // second place
runnerList.add("33"); // third place
runnerList.add("27"); // fourth place
displayList(runnerList);
```

Question 3 In the previous example, what changes to testList are necessary to represent the runner's numbers as Integer objects instead of strings?

```
ListInterface<String> runnerList = new  AList<String>();
// runnerList has only methods in ListInterface
runnerList.add("16"); // winner
runnerList.add(" 4"); // second place
runnerList.add("33"); // third place
runnerList.add("27"); // fourth place
displayList(runnerList);
```

```
ListInterface<Integer> rList =  new  AList<Integer>();
rList.add(16);
rList.add(4);
rList.add(33);
rList.add(27);
rList.displayList();
```

Java Class Library: The Interface **`List`**

- Method headers
  - `public T remove(int index)`
  - `public void clear()`
  - `public boolean isEmpty()`
  - `public boolean add(T newEntry)`
  - `public void add`
    ```
       (int index, T newEntry)
       . . .
    ```

Java Class Library: The Interface **`List`**

- Method headers (ctd.)
  - **`public T set(int index, T anEntry)`**
    `// like replace`
  - **`public T get(int index)`**
    `// like getEntry`
  - **`public boolean contains`**
    `(Object anEntry)`
  - **`public int size()`**
    `// like getLength`

Java Class Library: The Interface **`ArrayList`**

- Implementation of ADT list with resizable array
  - Implements java.util.list
- Constructors available
  - **`public ArrayList()`**
  - **`public ArrayList`**
    `(int initialCapacity)`

Exceptions

What Will I Learn?

Objectives

In this lesson, you will learn how to:

- Use exception handling syntax to create reliable applications
- Use try and throw statements
- Use the catch, multi-catch, and finally statements
- Recognize common exception classes and categories
- Create custom exception and auto-closeable resources

Why Learn It?

Purpose

The user experience and programming functionality are very important components to a well designed program.

Imagine requesting a software from a major company, paying lots of money for it, and it breaks every time you enter the wrong input. Would that product be very successful? Probably not. The user would prefer a handler that addresses the exception and prompts the user with the issue and continues functioning.

Exceptions allow for an elegant, consistent
way of handling errors that may occur throughout execution.

Exceptions

Exceptions, or run-time errors, should be handled by the programmer prior to execution.

Handling exceptions involves one of the following:

- Try-catch statements
- Throw statement

ORACLE Academy

Try-Catch Statements

Try-Catch statements are used to handle errors and exceptions in Java.

In the following code example, the program will run through the try code block first. In no exception occurs, the program will continue through the code without executing the catch block.

```
System.out.println("About to open a file");
try {
InputStream in =
        new FileInputStream("missingfile.txt");
    System.out.println("File open");
} catch (Exception e) {
    System.out.println("Something went wrong!");
}
```

Try-Catch Statements

If an exception is found, the program will search for a catch statement that catches the exception.

In the code segment below, an exception can be expected if the file “missingfile.txt” does not exist (a reference is being made to a non-existent object). When the exception occurs, the catch statement is prepared to handle it by noting the user with “something went wrong”.

```
try {
    System.out.println("About to open a file");
    InputStream in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
} catch (Exception e) {
    System.out.println("Something went wrong!");
}
```

Note: If no catch statement is found, and the exception is not handled in any other way, your program will crash during run-time.

Try-Catch Statements

The action that occurs when the catch statement is reached is up to the programmer and how s/he decides the program should operate.

For example, rather than displaying “something went wrong” the programmer could prompt the user with “File not found, please provide file name.” With this information, the program will be able to use another file and attempt to open that one.

```
try {
    System.out.println("About to open a file");
    InputStream in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
} catch (Exception e) {
    System.out.println("File not found, please provide file
      name");
    //read file name from user input
}
```

Using Multiple Catch Statements

You may find that using multiple catch statements is very effective in making catch statements more specific to the certain exception that occurs.

Multiple catch statements can be used for one try statement in order to catch more specific exceptions.

```
try {
    //try some code that may possibly cause an exception
} catch (FileNotFoundException e) {
    //code that executes when a FileNotFoundException
occurs
} catch (IOException e) {
    //code that executes when an IOException occurs
}
```

Using Multiple Catch Statements

In the code segment below, the try statement is executed first. There are 2 possible threats for exceptions:

1. FileNotFoundException - this may occur if “missingfile.txt” does not exist
2. IOException - this may occur if no data is found in “missingfile.txt” when the code attempts to access it.

```
try {
    System.out.println("About to open a file");
    InputStream in = new
             FileInputStream("missingfile.txt");     Threat #1
    System.out.println("File open");
    int data = in.read();                 Threat #2
    in.close();
}
//continued on next slide
```

Using Multiple Catch Statements

If FileNotFoundException occurs, the first catch statement is triggered.

If this exception does not occur, the program will continue to execute in the try statement until it reaches the IOException threat. If the IOException occurs, the second catch statement is triggered.

If no exceptions occur, the program will skip over the catch statements and continue executing the rest of the program.

```
//continued from previous slide...
 catch (FileNotFoundException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
} catch (IOException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
}
```

Finally Clause

Try-Catch statements can optionally include a finally clause that will always execute if an exception was found or not.

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(in != null) in.close();
    } catch(IOException e) {
        System.out.println("Failed to close file");
    }
}
```

Auto-closeable Resources

There is a “try-with-resources” statement that will automatically close resources if the resources fail. In the following example, “missingfile.txt” will close if the try statement completes normally, or if a catch statement is executed.

```
System.out.println("About to open a file");
try (InputStream in =
    new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Multi-Catch Statement

There's a multi-catch statement that allows you to catch multiple exception types in the same catch clause.

- Each type should be separated with a vertical bar: |

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new
     ObjectInputStream(is)) {
     cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " +
    cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Declaring Exceptions

Another way to handle an exception is to declare that a method *throws* an exception.

- A try statement will go in the method declaration.
- If the try fails, the method will throw the declared exception.

```
public static int readByteFromFile() throws
IOException {
    try (InputStream in = new
FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    }
}
```

Handling Declared Exceptions

Method-declared exceptions must still be handled, but can be handled inside the method declaration OR when the method is called. Here is an example of handling the exception when the method from the previous slide is called.

```
public static void main(String[] args) {
   try {
        int data = readByteFromFile();
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Creating Custom Exceptions

If you find that no existing exception adequately describes the exception, you can create your own, custom exceptions by extending the Exception class or one of its subclasses.

The code may look something like this:

```
public class MyException extends Exception {
    public MyException() {
        super();
        //MyException specific code here...
    }
    public MyException(String message) {
        super(message);
        //MyException specific code here...
    }
//MyException specific code here...
}
```