Introduction to Programming

6: Program Control Structures:

Part 3 – Exceptions

(plus – A First Look at Arrays)

Exceptions

- The normal flow of control of a program involves sequence, selection and iteration
- If it is possible that some exceptional event may occur, not expected in the normal flow of the program, it can make the code more difficult to understand if the logic that deals with the unusual event has to be included in the normal program flow
- Modern programming languages such as Java include a separate control flow mechanism for this situation – exception handling

Introduction to Exceptions

- An *exception* signals that an unexpected event has occurred that disrupts the normal flow of control of the program
- Normal flow of control is abandoned - control passes to an exception handler if there is one
- Can arise through
  - Programming errors – e.g. trying to access the 6th character in a String that is only 5 characters long
  - Runtime environment situation – e.g. a network link the program uses is down

RuntimeException

- When unexpected event occurs, the exception is said to be *thrown*
- Exceptions in Java are classes and the exception that is thrown is an instance of an exception class
- There are many different kinds of exception, each kind is associated with a different kind of event or error
- Most exceptions that we will be concerned with in this module are kinds of RuntimeException
  - Some examples of runtime exceptions are on the next slide

Example RuntimeExceptions

- StringIndexOutOfBoundsException
  - Thrown when attempting to use an index that is not between 0 and one less than the length of a String inclusive to access a character in the String
- NullPointerException
  - E.g. thrown when attempting to use a String variable to access a String but the variable does not refer to any String and has the value, **null**.

Example Program

```
public class Example1 {

    public static char getInitial(String aName) {
        return aName.charAt(0);
    }

    public static void main(String[] args) {
        String name = null;
        char choice;
        while (true) {
            TextIO.put("Type 1 to enter a name or q to quit: ");
            choice = TextIO.getlnChar();
            switch (choice) {
            default :  // if user does not type a '1' or a 'q' treat input as a '1'
            case '1' :
                TextIO.put("Enter a name: ");
                name = TextIO.getln(); // no break, so falls through
            case 'q' :
                break;  // oops, this break is supposed to break the loop, not the switch
            }
            TextIO.putln("The name's initial is " + getInitial(name));
        }  // end while loop
    }
}
```


Comments on program

- In Example1, the programmer has forgotten that a **break** in a switch statement exits the switch statement and not the enclosing loop
- If the user types 'q' at the first prompt the program crashes and the JVM displays:

```
Exception in thread "main" java.lang.NullPointerException
    at Example1.getInitial(Example1.java:4)
    at Example1.main(Example1.java:20)
```


Comments on program

- Note that two line numbers are listed in the JVM's message.
  - The NullPointerException is thrown in getInitial() when the method attempts to call the charAt() method of the String, but there is no String (this is line 4).
  - getInitial() was called in main() on line 20; it was this call, passing over a String variable whose value was **null**, that resulted in getInitial() throwing the NullPointerException. The exception has now reached main() and the program crashes.

Fixing the program

- Error that the programmer made was to omit including a break for the loop
  - also omitted the break (from switch) in case ‘1’
- How does one break out of an enclosing loop from a switch statement (or an inner loop)?
- One approach is to *label* the loop
  - Java allows you to give a loop a name
  - You can use the name in a break statement to indicate what the break applies to


```
public class Example2 {

    public static char getInitial(String aName) {
       return aName.charAt(0);
    }

    public static void main(String[] args) {
         String name = null;
        char choice;
        mainLoop: while (true) {  // mainLoop is a label
            TextIO.put("Type 1 to enter a name or q to quit: ");
            choice = TextIO.getlnChar();
            switch (choice) {
            default :  // if user does not type a '1' or a 'q' treat input as a '1'
            case '1' :
                TextIO.put("Enter a name: ");
                name = TextIO.getln(); break;  // break now included
            case 'q' :
              break mainLoop;  // this break statement exits the loop
            }
            TextIO.putln("The name's initial is " + getInitial(name));
        }  // end mainLoop
    }
}
```


Summary of Example2

- Now if the user types ‘q’ to quit the application does not throw any exception and the program just terminates normally

Handling (catching) exceptions

- Given that Java includes exception handling as a mechanism for handling errors or unusual events, how does it work, how does one prevent the program crashing and handle the exceptional situation?
- This is done using something called a *try-catch statement*

try-catch statements I

- An exception that has been thrown can be caught (handled) using a *try-catch statement*
- Allows the programmer to define what the program should do when the unexpected event occurs

```
try {
    int index = TextIO.getlnInt();
    char chosen = myString.charAt(index);
    TextIO.putln("The character is " + chosen);
} catch (StringIndexOutOfBoundsException obe) {
    TextIO.putln("There is no such character!");
    TextIO.putln(obe.getMessage());
}
```


try-catch statements II

- Only exceptions thrown in the **try** block can be caught in the **catch** following the **try** block.
- Any statements in the **try** block occurring after the place where the exception is thrown are skipped and control transfers to the **catch** block.
- Once the exception is caught, execution continues as normal and the program does not crash.

try-catch statements II

- In the code on slide 13, suppose that myString refers to "hello"
- If the user types a 1, the output is

```
The character is e
```

- If the user types a 6, the output is

```
There is no such character!
String index out of range: 6
```


try-catch statements III

- The getMessage() method of the Exception class returns a String that contains information about the error
  - In this case, “String index out of range: 6”
- If you want to display the exception name and line numbers you can call the method printStackTrace().
  - Note that this does not print to the TextIO console but to where System.out.print() would display.

try-catch statements IV

- Can have more than one **catch** following a **try** block
- First catch that matches the exception is executed
- Duplicate catch statements (i.e. for same exception) are not allowed

```
try {
    int index = TextIO.getlnInt();
    char chosen = myString.charAt(index);
    TextIO.putln("The character is " + chosen);
} catch (StringIndexOutOfBoundsException obe) {
    TextIO.putln("There is no such character!");
} catch (NullPointerException npe) {
    TextIO.putln("That was a null String!");
}
```


Example2 revisited

- We can use a try-catch statement to deal with the exception that occurred in Example2
- Code on following page also deals with a NullPointerException, though this not expected
- Code for both versions is on Moodle if you want to try running them

try-catch statements V

- From Java SE7 onwards can name several exceptions in the same catch statement
- In code below, e could be either exception

```
try {
    int index = TextIO.getlnInt();
    char chosen = myString.charAt(index);
    TextIO.putln("The character is " + chosen);
} catch (StringIndexOutOfBoundsException |
         NullPointerException e) {
    TextIO.putln(e.getMessage());
}
```


Exceptions in this lecture

- Have seen that modern programming languages include a mechanism for dealing with errors and unusual events
- Allows the logic that deals with handling or recovering from these events to be kept separate from the normal flow of control
- That mechanism is exception handling
- We have looked at how runtime exceptions can be thrown and how to catch them

A first brief look at Arrays

- Arrays will be the subject of a separate lecture but will follow the example of the textbook (section 3.8) and briefly introduce them now.

Arrays in Java

- Most programming languages provide array types – Java is no exception
- In Java, arrays are classes (reference types)
- An array object is an *indexed list* of elements of the same type (a primitive type or a reference type)
- The array type name is written as

  elementType[]
- The type name can be used to declare variables including parameters to methods and constructors

Declaring an array variable

- To declare an array variable you use the array type name

```
int[] list;  /* can refer to arrays of
int values /*

String[] names;  /* can refer to arrays
of String values */
```


Array objects

- The element type of an array can be
  - a primitive type *or*
  - a reference type (*eg* the parameter to main() is an array of String, which is a reference type)
- When an array object is created, every element of the array is given a default value
  - 0 for numeric types
  - **false** for **boolean**
  - **null** for reference types
- The syntax to call an array constructor is:

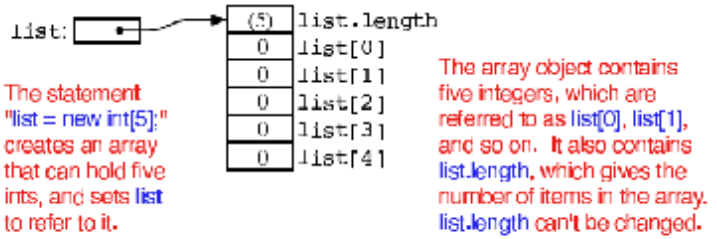
  new *ElementType* [*arrayLength*];

Creating an array object

- To create an array object, call the constructor for the array type and specify how many elements the array should contain

```
int[] list;
list = new int[5]; /* create an array
object with five int elements */
```


Array of primitive types (Eck, page 319)



Arrays in Java continued

- The first array element has index 0
  - An array of 5 elements has elements [0] to [4]
- Each instance of an array type has a field called *length* = the number of elements
- Array instances (objects) are created using an array constructor, once created an array object cannot change length
- An array reference variable refers to an array object or has value **null**

Accessing an array element

- Can think of each array element as a variable of the element type
- Use its index number in square brackets to access the element

```
int[] list = new int[5];
list[0] = 1;  // regard list[0] as an int variable
for (int i = 1; i < list.length; i++) {
    list[i] = list[i-1]*2;  // list[i] as a variable
}
```

Notes on accessing elements

- So you can think of an array as a collection of variables whose type is whatever the element type of the array is
- You access the element using its index
  - To get its value
  - To update its value
- The loop on the previous slide set the value of each array element to be twice that of the element before it

Next week

- Reading
  - For a longer first look at arrays, read section 3.8 of the book
- Will start to look at subroutines
  - Discussed in chapter 4 of Eck
    - Read sections 4.1 and 4.2 for a preview
  - Uses static methods in Java

Questions?