Additional Programming Concepts

Exceptions

Exceptions

- When an error occurs during runtime that occurs due to some exceptional event, an exception occurs.
- In Java, an exception is an object that contains information about the runtime condition that has occurred.
- Normally, exceptions will cause your program to terminate unless they are caught and handled with special code.

Exceptions we've seen

```
ArithmeticException
NumberFormatException
StringIndexOutOfBoundsException
ArrayIndexOutOfBoundsException
NullPointerException
IOException
```

Handling exceptions

- Exceptions don't have to crash our programs!
- We can do two things when an exception is caught:
  - Catch the exception and run a sequence of instructions to "handle" the error in some way.
  - Throw the exception back to the method that called this method and let it deal with the exception.

Catching an exception

- To catch an exception, we determine which instruction(s) can cause an exception.
- We then enclose the instruction(s) in a **`try`** block.
- The **`try`** block is followed immediately by a **`catch`** block with code to execute if the exception occurs.
- If the exception occurs during execution of the **`try`** block, control moves immediately to the **`catch`** block for that exception.
- If the exception does not occur, the **`catch`** block is not executed.

Example

```
do {
   System.out.println("Please input your age:");
   userInput = scan.nextLine();
   age = Integer.parseInt(userInput);
   if (age < 0)
        System.out.println("ERROR: Invalid input.");
} while (age < 0);
```

assume `userInput` is a `String` variable

If the user inputs anything other than a valid **`int`**, **`Integer.parseInt`** will throw the **`NumberFormatException`** and the program will crash.

Example Revised

```
do {
  System.out.println("Please input your age:");
  userInput = scan.nextLine();
  try {
      age = Integer.parseInt(userInput);
  }
  catch (NumberFormatException e) {
      age = -1;
  }
  if (age < 0)
      System.out.println("ERROR: Invalid input.");
} while (age < 0);
```

e is a reference to the exception; we could call methods on e to find out more about the exception

7

Another Example

```
public static double findAverageMileage(Car[][] lot)
{
    int sum = 0;
    int numCars = 0;
    for (int row = 0; row < lot.length; row++)
      for (int col = 0; col < lot[row].length; col++)
         if (lot[row][col] != null) {
            sum += lot[row][col].getMileage();
            numCars++;
         }
    double result = (double)sum/numCars;
    return answer;
}
```

this statement can throw an exception

8

Using try/catch

```
public static double findAverageMileage(Car[][] lot)
{
  // calculation of sum and numCars not shown here
  ...
  double result;
  try {
    result = (double)sum/numCars;
  }
  catch (ArithmeticException e) {
    result = 0.0;
  }
  return result;
}
```

replacement for statement from previous example

9

A better way

```
public static double findAverageMileage(Car[][] lot)
{
  // calculation of sum and numCars not shown
  double result;
  if (numCars != 0)
     result = (double)sum/numCars;
  else
     result = 0.0;
  return result;
}
```

Use exception handling only for those runtime cases that you can't correct yourself without the program crashing.

10

Throwing an exception back

```
public static int countLines
  (String filename) throws IOException
{
     Scanner fileScan = new Scanner(
          new File(fileName));
     ...
```

If the input file is not found, an IOException is thrown by the File constructor. Instead of catching the exception, this method throws it back to whatever method called it. The calling method must either catch the exception or throw the exception as well toits caller, etc.

11

Using `throws` vs. `try/catch`

- An exception may occur in some method due to illegal data passed to it by its caller.
- So this method won't catch the exception itself.
- Instead, it will use **`throws`** to throw it back to the caller to **`catch`** it.
  - Example: When **`parseInt`** detects an error, it doesn't deal with it itself; it throws the exception back to us.
- Determining which method is responsible for dealing with an exception is part of software design and engineering.

12

Different kinds of exceptions

- In Java there are two kinds of exceptions:
  - Checked - these exceptions must either be caught or thrown to a calling method
    - Examples: **`IOException`** **`InterruptedException`**
  - Unchecked - these exceptions are not required to be caught or thrown to a calling method
    - Examples: **`NullPointerException`** **`ArrayIndexOutOfBoundsException`** **`NumberFormatException`**

Exceptions and the Java API

- In order to determine if you must explicitly catch an exception/throw it to your method's caller or not, you can look at the Java API.
- If you call a method that can throw an exception, and this exception is not **`RunTimeException`** nor one of its subclasses, then you must either catch this *checked exception* or throw it to your method's caller.
- If you call a method that can throw an *unchecked exception*, it is up to you whether you will deal with it or not. (e.g. `Integer.parseInt` does not require an explicity try/catch or throws statement)