Lecture April 6th - More Exceptions

Bentley James Oakes

April 5, 2018

This Lecture

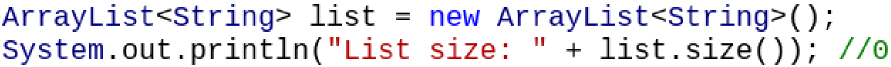
1. Recap
2. Writing to a File
3. Exception Handling
4. Exception Types
5. File IO Exceptions
6. Wrapper Classes
7. Immutable Classes
8. Storing Objects

Section 1

Recap

ArrayList

- Let's create an `ArrayList` which stores `Strings`
- We'll indicate the type the `ArrayList` stores with `<String>`



```java
ArrayList<String> list = new ArrayList<String>();
System.out.println("List size: " + list.size()); //0
```

ArrayList

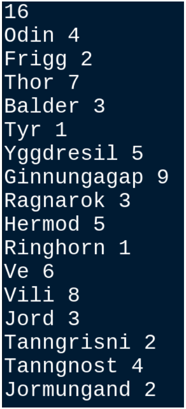
Some useful methods for an `ArrayList`

- `add(Object element)` ← this method appends the element to the end of the list
- `get(int index)` ← this method returns the element at the specific index
- `isEmpty()` ← returns *true* if there are no elements in the list and *false* otherwise
- `indexOf(Object o)` ← returns the index of `o` in the list, or -1 is `o` is not in the list
- `size()` ← returns the number of elements in the list
- `contains(Object o)` ← returns *true* if `o` is in the list and *false* otherwise

This documentation will be on the final

The File to Read

Here's the file we're going to read
Has cat names and cat ages on each line



```
16
Odin 4
Frigg 2
Thor 7
Balder 3
Tyr 1
Yggdresil 5
Ginnungagap 9
Ragnarok 3
Hermod 5
Ringhorn 1
Ve 6
Vili 8
Jord 3
Tanngrisni 2
Tanngnost 4
Jormungand 2
```

Creating Cats

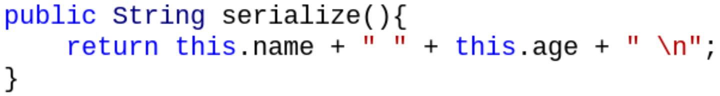
```
ArrayList<Cat> catList = new ArrayList<Cat>();
String filename = "cats.txt";

try{
    //don't forget: import java.io.*;
    FileReader fr = new FileReader(filename);
    BufferedReader br = new BufferedReader(fr);

    String currentLine = br.readLine(); //get a line in the file
    while (currentLine != null){ //while there are still lines
        String[] tokens = currentLine.split(" "); //split the line
        Cat c = new Cat(tokens[0], Integer.parseInt(tokens[1])); //create the Cat
        catList.add(c);//add to the ArrayList
        currentLine = br.readLine(); //get the next line
    }
    br.close();
}catch (FileNotFoundException e){
    System.out.println("Could not find file: " + filename);
}catch (IOException e){
    System.out.println("Problem with file: " + filename);
}
```

Section 2

Writing to a File

Writing to a File

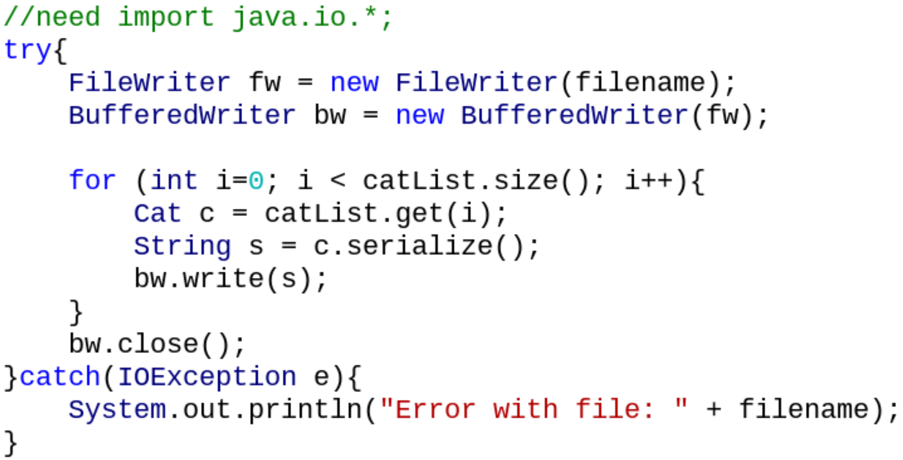
- Let's write some code to write the `ArrayList` of cats back to a file
- We'll write a method for `Cats` which defines how they should be written to a file
- Let's call this the `serialize` method
- Should give the same format as when we loaded the `Cats`



```
public String serialize(){
    return this.name + " " + this.age + " \n";
}
```

Writing to a File

- Let's save all the `Cats` in our `ArrayList`.
- This involves the objects `FileWriter` and `BufferedWriter`.



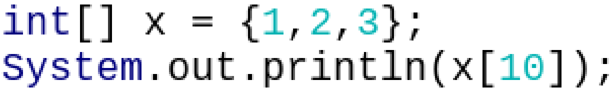
```java
//need import java.io.*;
try{
    FileWriter fw = new FileWriter(filename);
    BufferedWriter bw = new BufferedWriter(fw);

    for (int i=0; i < catList.size(); i++){
        Cat c = catList.get(i);
        String s = c.serialize();
        bw.write(s);
    }
    bw.close();
}catch(IOException e){
    System.out.println("Error with file: " + filename);
}
```

Section 3

Exception Handling

Exceptions Handling

- Let's review some concepts in exception handling
- Exceptions happen when there is an error

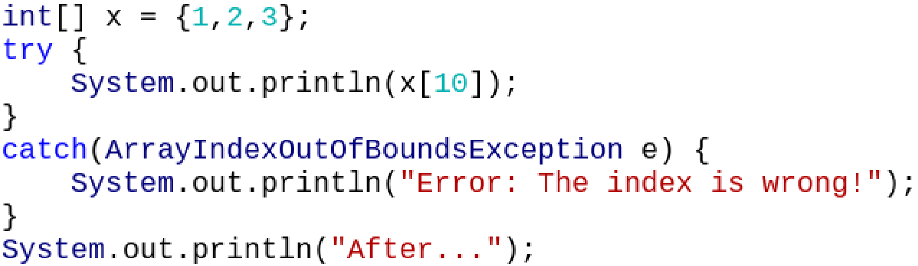


```
int[] x = {1,2,3};
System.out.println(x[10]);
```

The second line tries to access an invalid index. An `ArrayIndexOutOfBoundsException` will be thrown.

Exceptions Review

- Let's review how to *catch* these `Exceptions`
- We use a try/catch block to decide what code to run
- An exception that is caught will not cause the program to crash.

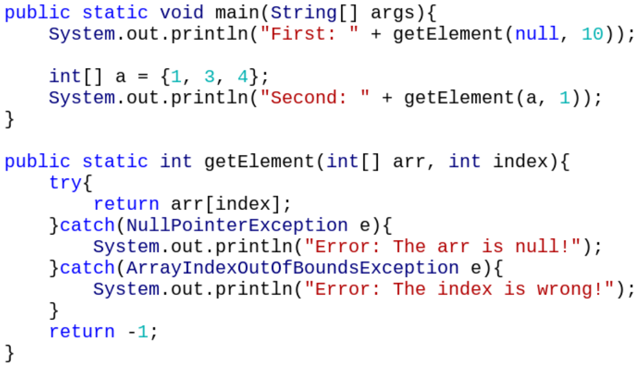


```java
int[] x = {1,2,3};
try {
    System.out.println(x[10]);
}
catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Error: The index is wrong!");
}
System.out.println("After...");
```

```
Error:  The index is wrong!
After...
```

Exceptions Review

- Let's look at handling multiple errors
- Java will execute the first catch block that matches the `Exception`

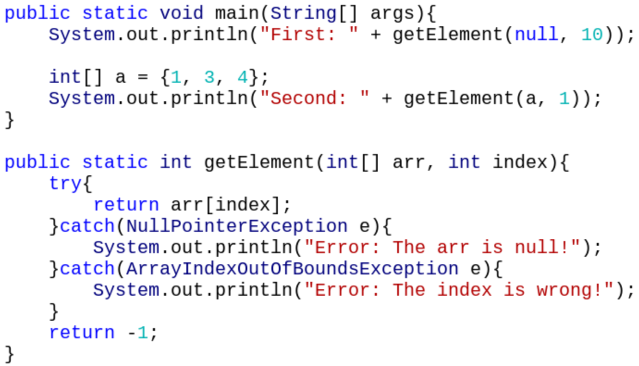


```
public static void main(String[] args){
    System.out.println("First: " + getElement(null, 10));

    int[] a = {1, 3, 4};
    System.out.println("Second: " + getElement(a, 1));
}

public static int getElement(int[] arr, int index){
    try{
        return arr[index];
    }catch(NullPointerException e){
        System.out.println("Error: The arr is null!");
    }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("Error: The index is wrong!");
    }
    return -1;
}
```

```
Error:  The arr is null!
First:  -1
Second:  3
```

Exceptions Review

- Let's look at handling multiple errors

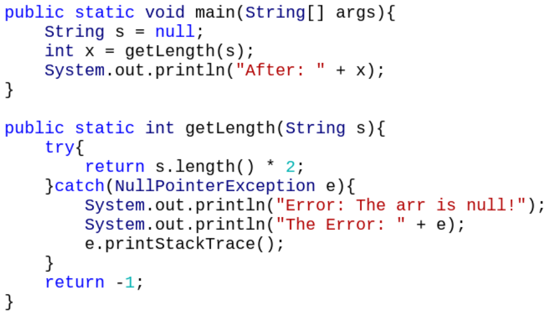


```java
public static void main(String[] args){
    System.out.println("First: " + getElement(null, 10));

    int[] a = {1, 3, 4};
    System.out.println("Second: " + getElement(a, 1));
}

public static int getElement(int[] arr, int index){
    try{
        return arr[index];
    }catch(NullPointerException e){
        System.out.println("Error: The arr is null!");
    }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("Error: The index is wrong!");
    }
    return -1;
}
```

```
Error:  The arr is null!
First:  -1
Second:  3
```

Exceptions Review

- Let's review how to print information about the `Exception`
- Note that the program still continues



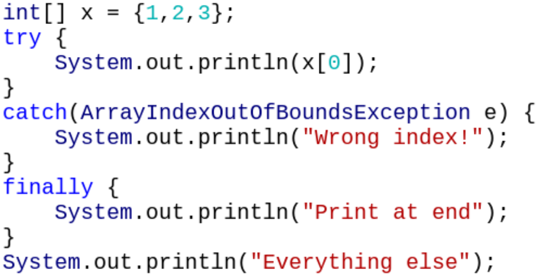
```java
public static void main(String[] args){
    String s = null;
    int x = getLength(s);
    System.out.println("After: " + x);
}

public static int getLength(String s){
    try{
        return s.length() * 2;
    }catch(NullPointerException e){
        System.out.println("Error: The arr is null!");
        System.out.println("The Error: " + e);
        e.printStackTrace();
    }
    return -1;
}
```

```
Error:  The arr is null!
The Error:  java.lang.NullPointerException
java.lang.NullPointerException
at ExceptionHandling.getLength(ExceptionHandling.java:13)
After:  -1
```

The Finally Block

- Let's see the *finally* block, just to finish off our discussion of `Exceptions`
- The finally block is attached to a try block, and it always executes.
- Even if one of the following happens:
  - an unexpected exception occurs in the try block
  - an exception occurs in the catch block
  - There's a return/continue/break statement in the try/catch block.
- You can have a finally even with just a try block (and no catch).

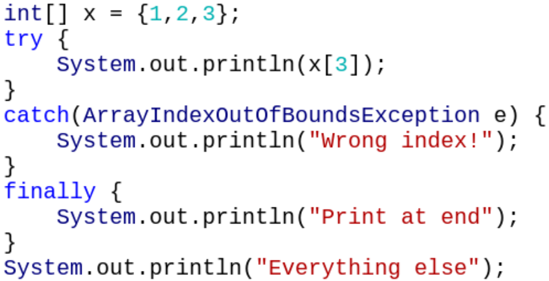
Exceptions Example



```
int[] x = {1,2,3};
try {
    System.out.println(x[0]);
}
catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Wrong index!");
}
finally {
    System.out.println("Print at end");
}
System.out.println("Everything else");
```

```
1
Print at end
Everything else
```

Exceptions Example 2



```java
int[] x = {1,2,3};
try {
    System.out.println(x[3]);
}
catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Wrong index!");
}
finally {
    System.out.println("Print at end");
}
System.out.println("Everything else");
```

```
Wrong index!
Print at end
Everything else
```

Finally Use

Why use `finally`?

- Close files readers/writers and scanners
- Close whether or not there was an Exception
- Good practice

Section 4

Exception Types

Checked vs Unchecked Exceptions

There are two kinds of exceptions in Java:

- Checked
  - `Exception`
  - `IOException`
- Unchecked
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `IllegalArgumentException`

Unchecked Exceptions

- These exceptions are not checked at compile-time
- Most exceptions are unchecked, and they can cause your code to crash at run-time
- You are not forced by the compiler to handle these exceptions
- It is up to the programmer to decide to catch the exceptions

Example: `int c = 12/0;` is a valid statement, and produces an unchecked `Exception`

Checked Exceptions

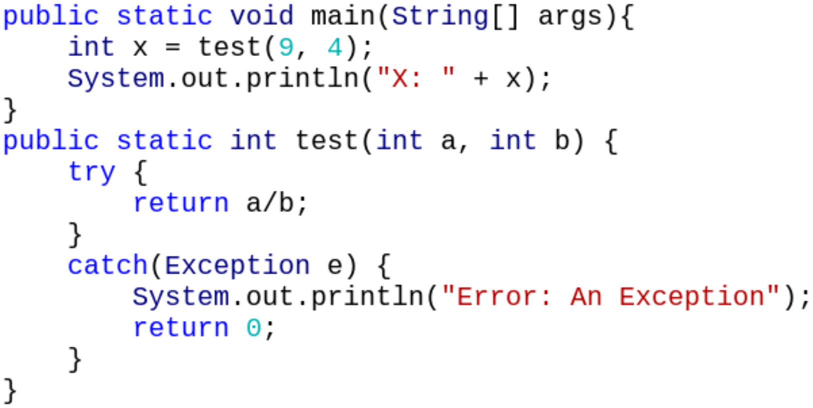
- These exceptions are checked at compile-time!
- The programmer is forced to handle these exceptions

There are two ways to handle these *checked exceptions*:

1. Use try/catch block to surround the code that might throw a checked exception
2. Specify that your method might throw a particular Exception
    - Force anyone using the method to handle the Exception

Option 1

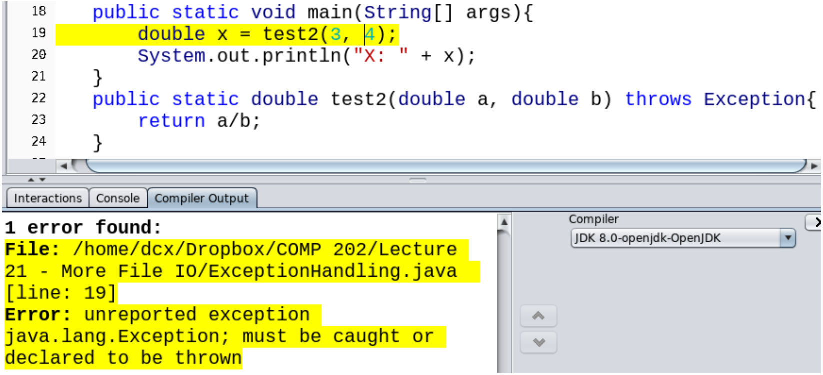
Surround the code that might throw an exception with a try/catch block.



```java
public static void main(String[] args){
    int x = test(9, 4);
    System.out.println("X: " + x);
}
public static int test(int a, int b) {
    try {
        return a/b;
    }
    catch(Exception e) {
        System.out.println("Error: An Exception");
        return 0;
    }
}
```

Option 2

Specify in the method header that there's an exception using the throws keyword followed by the type of the exception.
The method call then needs to be caught.

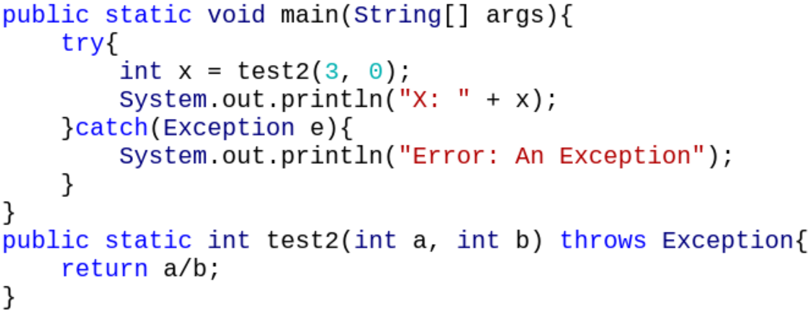


```
18    public static void main(String[] args){
19        double x = test2(3, 4);
20        System.out.println("X: " + x);
21    }
22    public static double test2(double a, double b) throws Exception{
23        return a/b;
24    }
```

```
1 error found:
File: /home/dcx/Dropbox/COMP 202/Lecture
21 - More File IO/ExceptionHandling.java
[line: 19]
Error: unreported exception
java.lang.Exception; must be caught or
declared to be thrown
```

Compiler
JDK 8.0-openjdk-OpenJDK

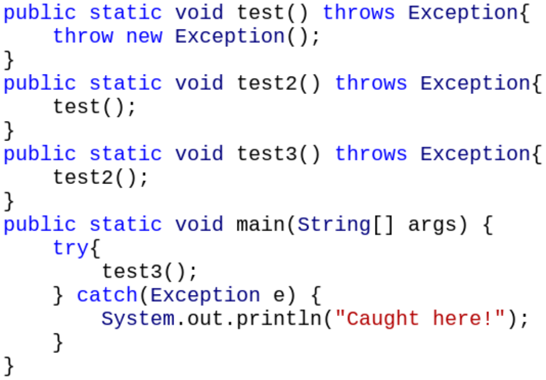
Option 2



```java
public static void main(String[] args){
    try{
        int x = test2(3, 0);
        System.out.println("X: " + x);
    }catch(Exception e){
        System.out.println("Error: An Exception");
    }
}
public static int test2(int a, int b) throws Exception{
    return a/b;
}
```

Chaining

- We can keep throwing the `Exception` to the calling method



```java
public static void test() throws Exception{
    throw new Exception();
}
public static void test2() throws Exception{
    test();
}
public static void test3() throws Exception{
    test2();
}
public static void main(String[] args) {
    try{
        test3();
    } catch(Exception e) {
        System.out.println("Caught here!");
    }
}
```

- In this course, you're not allowed to throw Exceptions from the main method

Why Pass the Exception?

- Why would we throw an `Exception` to the calling method?
- We force the programmer to decide what to do when there's an error
    - This is more related to design of your program

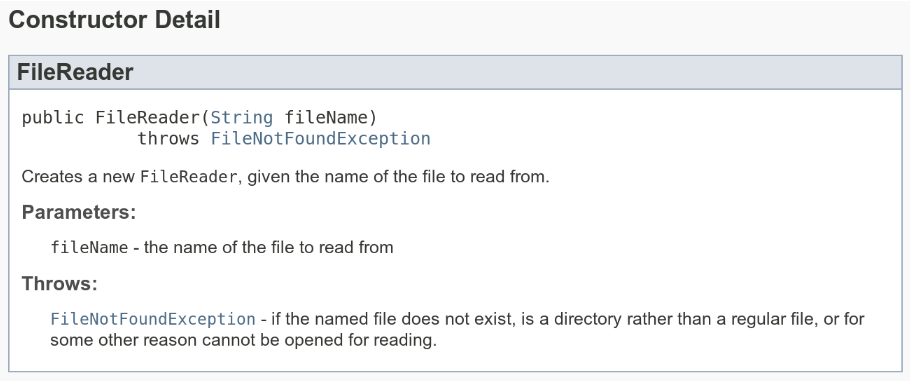
Section 5

File IO Exceptions

IOException and FileNotFoundException

- File IO operations are so error prone that the code containing them can throw an `IOException`

From the `FileReader` object:



Constructor Detail

FileReader

```
public FileReader(String fileName)
           throws FileNotFoundException
```

Creates a new `FileReader`, given the name of the file to read from.

Parameters:

`fileName` - the name of the file to read from

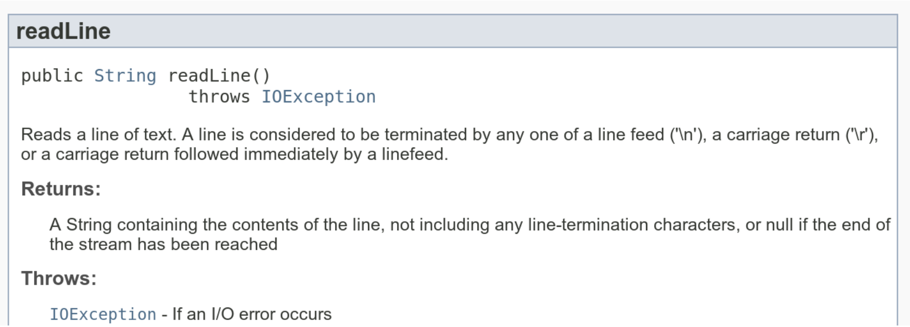
Throws:

`FileNotFoundException` - if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

IOException and FileNotFoundException

- The readline method of the BufferedReader object can throw an IOException

From the BufferedReader object:



readLine

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

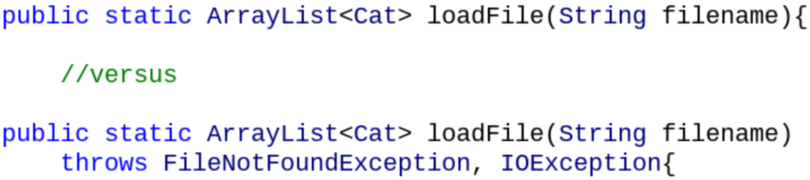
A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

Handling These Exceptions

- Both these exceptions are checked exceptions. Therefore, you must handle them or you will get a compile-time error
- Put all File IO operations inside a try-block and have a catch block for the exceptions
- Or, pass on the exceptions by using throws in the header of the method



```
public static ArrayList<Cat> loadFile(String filename){

    //versus

public static ArrayList<Cat> loadFile(String filename)
    throws FileNotFoundException, IOException{
```

File Reading With Throws

```java
public static void main(String[] args){
    try{
        readFile("abc.txt");
    }catch(FileNotFoundException e){
        System.out.println("The file was not found.");
    }catch(IOException e){
        System.out.println("Error with the file.");
    }
}
public static void readFile(String filename)
    throws FileNotFoundException, IOException{
    FileReader fr = null;
    BufferedReader br = null;
    try{
        fr = new FileReader(filename);
        br = new BufferedReader(fr);
        String s = br.readLine();
        while (s != null){
            System.out.println(s);
            s = br.readLine();
        }
    }finally{
        if (fr != null){
            fr.close();
        }
        if (br != null){
            br.close();
        }
    }
}
```

Section 6

Wrapper Classes

Wrapper Classes

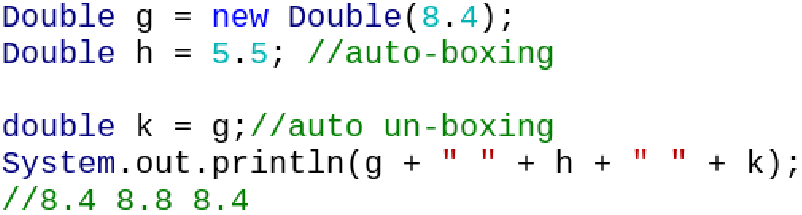
- What about storing ints and doubles in `ArrayLists`?
- We can't, because they are primitive types, and `ArrayLists` only store reference types/`Objects`

Wrapper Classes

- We use the classes `Integer` or `Double`
- These are very simple classes
- Check the documentation for useful attributes and methods
- Examples:
  - `MAX_VALUE` - A constant holding the largest positive finite value of type double
  - `SIZE` - The number of bits used to represent a double value.
  - `parseDouble(String s)` - Returns a new double initialized to the value represented by the specified String

Boxing

- Java does some magic behind the scenes to make these wrapper classes mostly act like the primitive types
- This is called *boxing* and *unboxing*



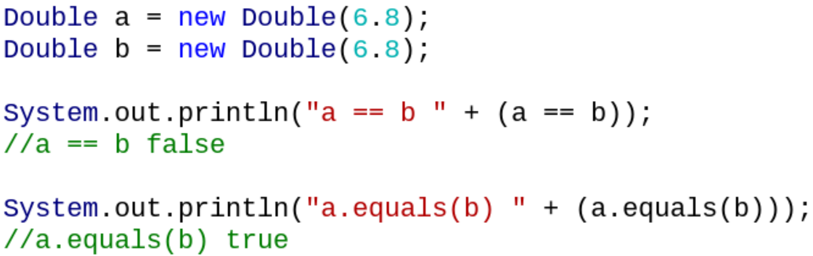
```
Double g = new Double(8.4);
Double h = 5.5; //auto-boxing

double k = g;//auto un-boxing
System.out.println(g + " " + h + " " + k);
//8.4 8.8 8.4
```

Wrapper Class Example

```
ArrayList<Double> dblList = new ArrayList<Double>();

Double dbl = new Double(4.5);

dblList.add(dbl);

System.out.println(dblList);//[4.5]

Double first = dblList.get(0);
System.out.println(first);//4.5

double x = first.doubleValue();
System.out.println(x);//4.5

double y = first + 6.7;
System.out.println(y);//11.2
```

Wrapper Class Reference Types

- But we can't forget that they are reference types
    - Specifically, *aliasing* and checking for *equality*



```
Double a = new Double(6.8);
Double b = new Double(6.8);

System.out.println("a == b " + (a == b));
//a == b false

System.out.println("a.equals(b) " + (a.equals(b)));
//a.equals(b) true
```

Wrapper Class Overview

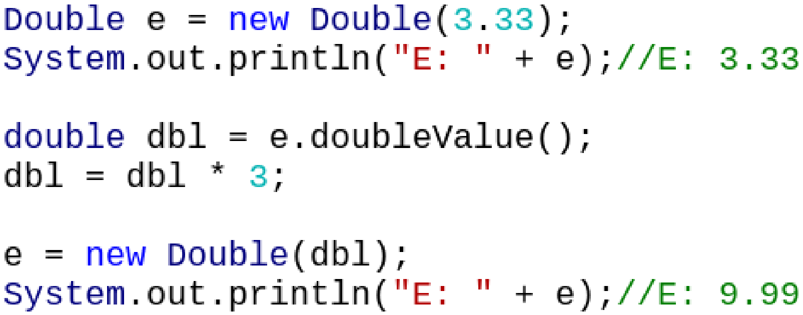
- Wraps primitive values in a class
- Only use wrapper classes if you have to place values in an `ArrayList`

Section 7

Immutable Classes

Immutable

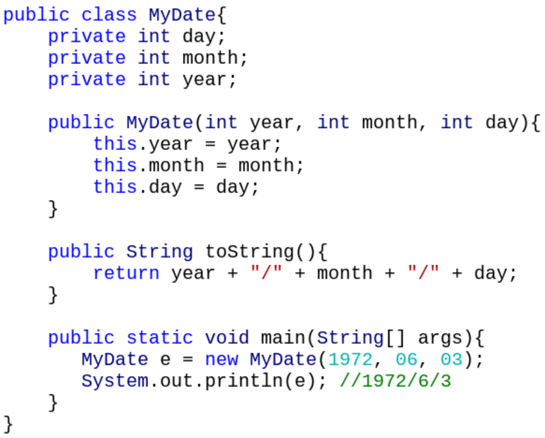
- Recall that Strings in Java are *immutable*
- After Strings are created, we can't change their value
- Another example is these wrapper classes
- There is no setter to change the value stored
- If you want a Double instance with a different value, you must create a new instance



```
Double e = new Double(3.33);
System.out.println("E: " + e);//E: 3.33

double dbl = e.doubleValue();
dbl = dbl * 3;

e = new Double(dbl);
System.out.println("E: " + e);//E: 9.99
```

Writing an Immutable Class

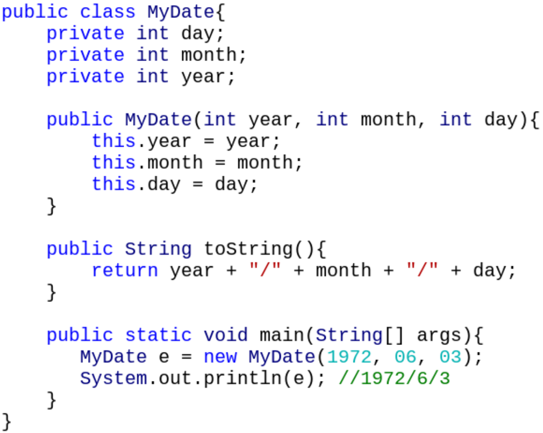
- We can write our own immutable classes
- Just make the variables private, and don't provide setter methods



```
public class MyDate{
    private int day;
    private int month;
    private int year;

    public MyDate(int year, int month, int day){
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public String toString(){
        return year + "/" + month + "/" + day;
    }

    public static void main(String[] args){
       MyDate e = new MyDate(1972, 06, 03);
       System.out.println(e); //1972/6/3
    }
}
```

Writing an Immutable Class

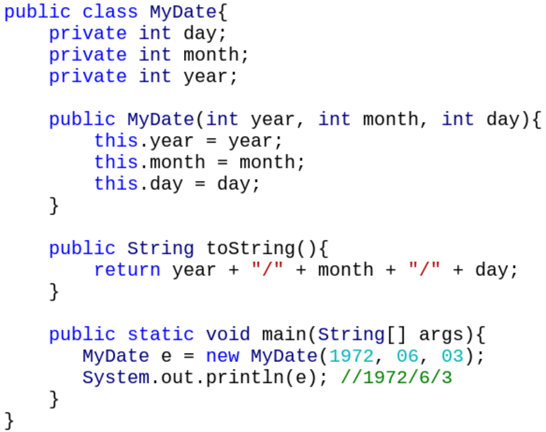
- Why are immutable classes useful?
- If we have an instance of `MyDate`, then we know that another part of our program can't accidentally change the information



```
public class MyDate{
    private int day;
    private int month;
    private int year;

    public MyDate(int year, int month, int day){
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public String toString(){
        return year + "/" + month + "/" + day;
    }

    public static void main(String[] args){
       MyDate e = new MyDate(1972, 06, 03);
       System.out.println(e); //1972/6/3
    }
}
```

Immutable Summary

- A class is immutable when:
- The attributes are private, and we haven't provided setter methods for the attributes
- Once we create the instance, we can't change the values of its attributes



```
public class MyDate{
    private int day;
    private int month;
    private int year;

    public MyDate(int year, int month, int day){
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public String toString(){
        return year + "/" + month + "/" + day;
    }

    public static void main(String[] args){
       MyDate e = new MyDate(1972, 06, 03);
       System.out.println(e); //1972/6/3
    }
}
```

Section 8

Storing Objects

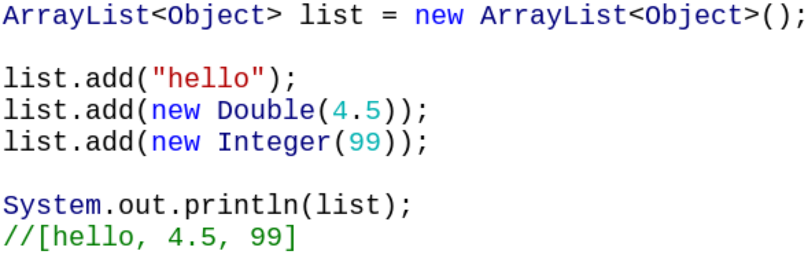
Creating ArrayLists

- We saw how to create `ArrayLists` to store various types
- `ArrayList<Integer> myList = new ArrayList<Integer>();`
- `ArrayList<String> otherList = new ArrayList<String>();`
- `ArrayList<Book> books = new ArrayList<Book>();`
- `ArrayList<Double> numbers = new ArrayList<Double>();`

What if we want to store both `Strings` and `Integers` in an `ArrayList`?

Storing Objects in an ArrayList

To store multiple object types in an `ArrayList`, we write
`ArrayList<Object> list = new ArrayList<Object>();`
Note that we can't store primitive types, only reference types!



```java
ArrayList<Object> list = new ArrayList<Object>();

list.add("hello");
list.add(new Double(4.5));
list.add(new Integer(99));

System.out.println(list);
//[hello, 4.5, 99]
```

Object Inheritance

- We've talked about *objects* in Java, but there's some missing background material
- We're ignoring the concept of inheritance, which means that all reference types (like `Strings` and `Students`) have some default methods
    - `toString` and `equals`
- We say that all classes *inherit* these methods from the parent class `Object`

Why is This Important?

- This is important because all reference types are guaranteed to have an `equals` method and a `toString` method.
- For example, the `toString` method of an `ArrayList` can just call `toString` on all the elements
- Or, we have the `contains` method on `ArrayLists`
  - `contains(Object o)` ← returns *true* if o is in the list and *false* otherwise
- The `contains` method will call the `equals` method to see if the element is the same as o
  - Note this means you will have to write your own `equals` method in your classes, but that is outside COMP 202