CS 151

Exceptions & Javadoc
slides available on course website
Announcements

Prelab 1 is due now. Please place it in the appropriate (Mon vs. Tues) box.

Please attend lab this week. There may be a lecture portion at the beginning.

If you are having trouble with your account, or want help getting your home machine set up, start by asking a lab helper during your lab time.

My office hours today are shortened to 1:30-2:30pm, and Tuesday’s are shortened to 12-1pm. Apologies! I need to get to the labs for the lecture!
You have all used the Java API to look up information about Java classes.

These pages define the behaviour of these classes (and leave out how they are implemented).

Wouldn’t it be nice if you could easily make these pages for your classes?
Java provides a tool called Javadoc that generates html web documentation.

Bonus: it’s easy to use, and you don’t have to do much. Just run

```
javadoc -d docs *.java
```

inside a directory with java files.

Let’s try it out...
So, this is where the Java web documentation comes from.

- Javadoc collects information on interfaces from source code directly
- Can supply extra info by using special comments inside source code:
  - /** javadoc stuff */
  - @author
  - @param
  - @return
  - @throws
  - html tags, such as <a></a>, <p>, <br>, etc.
  - and more
You should use javadoc documentation for your programs.

The editor you’ll be learning in lab (namely, eclipse) will automate a lot of the process.
Exceptions occur at runtime, when you really offend your program.

For example:

```java
int x = 5/y;  //will throw an ArithmeticException if y=0
int[] array = {0, 1, 2, 3};
x = array[4];  //will throw an IndexOutOfBoundsException
```

At compile-time, you cannot know that these errors will occur. As much as possible, you should anticipate and “handle” your exceptions. You can do this in two ways. When an exception is thrown, you can either

- **catch** the exception in order to handle it gracefully, or
- **throw** the exception to the caller, in the hopes that they will catch it.

In Java, exceptions are objects that store (useful) information.
When an exception is thrown, you can always just pass the buck.

That is, rather than deal with the exception yourself, you can just acknowledge to Java that it may happen, so that the next person can deal with it (or pass it on themselves).

To do this, just add `throws NameOfException` to the end of the method header. This tells the caller of the method that an exception may be thrown.

For example:

```java
public void read(String s) throws FileNotFoundException {
    Scanner input = new Scanner( new File("badFile.txt") );
}
```
Alternatively, you can try to deal with the exception yourself. In this case, you need to be alerted when an exception is thrown, so that you can perform special “handling” code.

To do this, you add a try-catch-finally block. For example:

```java
public void read(String s) {
    try {
        Scanner input = new Scanner( new File("badFile.txt") );
    } catch( FileNotFoundException e ) {
        System.exit(0);
    } catch( ArrayOutOfBoundsException e ) {
        System.out.println( e );
    } finally {
        // cleanup here, if you need it
        // the code in here is always run
    }
    // this code is run after the finally block
}
```
Catching the Exceptions

```java
public void read(String s){
    try {
        Scanner input = new Scanner(new File("badFile.txt"));
    } catch( FileNotFoundException e ) {
        System.exit(0);
    } catch( ArrayOutOf BoundsException e ) {
        System.out.println( e );
    } finally {
        // cleanup here, if you need it
    }
}
```

First, you place the code that may throw the exception in a try block. If an exception is not thrown, proceed as usual then go to finally block. If an exception is thrown, Java immediately looks for a matching catch block. If none is found, control goes to finally block, then exception propagates. If matching catch block is found, the catch block is executed, then control passes to the finally block and onwards.
In Java, Exceptions are just objects.

You can create your own Exception and throw it using the `throw` clause.

```java
throw new MyOwnException( "The problem was something." ) ;
```

Since throwing your own exception is a signal that something is amiss, you should probably not catch it immediately (in the same scope). Rather, you should have the containing method propagate the exception to the caller with a `throws` clause.
One of the main tasks in lab 1 is to implement an ArrayList from scratch!

As you will see in the lab, your MyArrayList class will look something like:

```java
public class MyArrayList<T> extends AbstractList<T> {
    T[] data;    // backing storage for this arraylist
    int size;    // the number of elements in this arraylist

    public int size() {
        // What goes here?
    }

    public boolean isEmpty() {
        // What goes here?
    }

    private void resize() {
        // What goes here?
    }

    public void add(int index, T element) throws IndexOutOfBoundsException {
        // What goes here?
    }
}
```
An Iterator is a structure that allows you to step through it sequentially.

The Iterator interface contains 3 methods:

- hasNext() // return true if another element exists
- next() // returns the next element
- remove() // optional. removes the

Where have you seen an iterator before? What class has these methods, and therefore likely implement this interface?

When you create a list structure, such as an arraylist, it is good practice to also provide an iterator class that iterates over those elements. Usually you just place it as another non-public class in the same class as your public data structure. This is called a nested class, fyi. This way, this class has access to all of the class members and class methods of the structure, but itself represents a specific iterating structure.
class MyArrayListIterator<T> implements Iterator<T> {
    private int index;   // what index will you iterate next

    public MyArrayListIterator() {
        index = 0;
    }
    public boolean hasNext() {
        return (index < size);
    }
    public T next() {
        if( hasNext() ) {
            return get(index++);
        }
        throw new NoSuchElementException();
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
Once you have an Iterator, you need to provide access to it!

The Iterable interface has a “factory method” `iterator()`

```java
public class MyArrayList<T> extends AbstractList<T> implements Iterable<T> {
    T[] data;
    int size;    // same as before
    ... <class methods as before>

    public Iterator<T> iterator() {
        return new MyArrayListIterator(this);
    }
}
```

You could modify your ArrayList class to implement Iterable<T> so that you can produce iterators to iterate over the elements of your arraylist.

```java
public class MyArrayList<T> extends AbstractList<T> implements Iterable<T> {
    T[] data;
    int size;       // same as before
    ... <class methods as before>

    public Iterator<T> iterator() {
        return new MyArrayListIterator(this);
    }
}
```

Now, your ArrayList is Iterable! And it can be used in those funky for-loops, automatically...