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.

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!

Javadoc

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).

You all also have

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.

• you want people to be able to use the classes you create
• you don’t want these people to know about the implementation details

Let’s try it out...

Javadoc -d docs *.java

Bonus: it’s easy to use, and you don’t have to do much. Just run Javadoc inside a directory with java files.
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

Examples of propagating the exceptions:

```java
{ Scanner input = new Scanner( new File( "badFile.txt" ) );

public void read( String s )
    throws FileNotFoundException
{
    Scanner input = new Scanner( new File( "badFile.txt" ) );
}
```

In Java, exceptions are objects that store useful information.

- how: the exception to the caller, in the hopes that they will catch it.
- catch: the exception in order to handle it gracefully. Or
catch the exception in order to handle it gracefully. Or
When an exception is thrown, you can either
- As much as possible, you should anticipate and "handle" your exceptions.
- In Java, exceptions are objects that store (useful) information.

```
exceptions occur at runtime, when you really offend your program.
```
Catching the Exceptions

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.

```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
}
```

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

```java
try {
    // cleaning here, if you need to
} catch (ArrayOutofRangeException e) {
    System.out.println("Array out of range: ", e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array Index Out Of Bound Exception: ", e.getMessage());
} catch (FileNotFoundException e) {
    System.out.println("File not found Exception: ", e.getMessage());
} catch (Exception e) {
    System.out.println("Generic Exception: ", e.getMessage());
}
```

First, you place the code that may throw the exception in a try block.

1) If matching catch block is found, the catch block is executed, then control passes to the finally block and onwards. If none is found, control goes to the finally block. If no exception is thrown, it will proceed as usual. If an exception is thrown, the try block is exited.

Throwing Your Own Exceptions

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 your methods propagate the exception to the caller with a `throws` clause.

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

ArrayList

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?
    }
}
```

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

- List is implemented as ArrayList from scratch.
- One of the main tasks in lab 1 is to implement an ArrayList from scratch.

Catching the Exceptions

A throws clause should throw the containing method propagates the exception to the caller with the error. You should probably not catch it immediately (in the same scope). Instead, you throw your own exception if something is wrong. You can create your own exception and throw it using the throw clause.

In Java, Exceptions are just objects.
An Iterator is a structure that allows you to step through it sequentially.

```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);
    }
}
```

The Iterable interface has a `factory method` `iterator()` that can produce iterators to iterate over the elements of your arraylist. You could modify your Arraylist class to implement the Iterator pattern so that you can provide an iterator to iterate over the elements of your arraylist.

The `Iterator` interface has a `factory method` `next()` that allows you to step through the elements of the arraylist.

```java
public 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!

```java
public class MyArrayList<T> extends AbstractList<T> implements Iterable<T> {
    ... <class methods as before>
    public Iterator<T> iterator() {
        return new MyArrayListIterator(this);
    }
}
```

The `Iterable` interface contains 3 methods:

- `void remove()`: Removes the element at the current position of the Iterator. This method is not implemented by the `Iterator` class.
- `T next()`: Returns the next element in the collection.
- `boolean hasNext()`: Returns true if the Iterator has more elements.

When you need a list of elements that you can traverse, you can use the `next()` method to access the next element in the list. Each time you call the `next()` method, it returns the next element in the list and advances the Iterator to the next position.

**Iterator Pattern**

Now, your Arraylist is Iterable and can be used in those funky for-loops.

```java
{ 
    return new MyArrayListIterator(this); 
} () } 
```