Java java java java java java……

Mobile Application Programming: Android

Matthew Murdock

- u0661237@utah.edu
- www.blog.mtmurdock.com
- Lab/Office Hours: hour following class
- Available on MW after class until 5PM by appointment.

Assignment 1

Progress? Questions?

Checklist

- Install eclipse
- Install Android SDK
- Download Android version (2.2 or higher)
- Create virtual device (or connect real device)
- Create empty Android Application Project
- Code code code code…

ASK FOR HELP!

Email me and come to office hours

Please participate in class.

Inheritance

Because out-heritance didn’t sound as cool.

Developing for Android

- Most of the code you'll write for Android will subclass objects defined by Google in the Android SDK.
- Inheritance is an essential principle of all OOP, and Android is no exception to that rule.

Inheritance

- Wikipedia defines Inheritance as such:

  "In object-oriented programming (OOP), inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both... In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes.... The resulting classes are known as… subclasses... The relationships of classes through inheritance gives rise to a hierarchy."

Extends vs. Implements

- There are two forms of inheritance in Java, sometimes referred to direct and indirect inheritance or subclassing.
- Class inheritance (extends) allows an object to extend a concrete or abstract class, bringing with it data and functionality.
- Interface inheritance (implements) guarantees an object has certain functionality, but doesn't specify how.

extends

- To create a subclass of Activity, you would declare your class like this:

```
public class MyActivity extends Activity {
        // implementation
}
```

implements

- To indicate that your class implements the Runnable interface, you would declare your class like this:

```
public class MyActivity extends Activity implements Runnable {
        public void run() {
                // implementation
        }
}
```

Inheritance

- This is pretty basic. Very similar to other languages.
- If you want to know more, Google it.
- Java has a special kind of inheritance that is used a lot in Android, which you will need to know and understand.

Anonymous Objects

- Need instance of an interface or abstract
- Only using it once, or only internally.
- Rather than making a new file, use an anonymous object.

Anonymous Objects

```
OnClickListener l = new OnClickListener() {
      public void onClick(View v) {
            // implementation
      }
};
```

- Has same scope as where it is declared.
- Can access class and final variables.
- Can be shared between views.

Questions?

Collections And Data Structures

Or how I stopped worrying and
learned to love Generics

Android and Data Structures

- No Android specific structures.
- If you are familiar with Java generics and data structures then you're set.

Arrays

- Most basic data structure.
- Statically sized.
- Capable of holding exactly one type of thing

```
int[] ints = new int[10];
Button[] buttons = new Button[10];
```

Arrays

- What can you do with arrays?
- So why don’t we just use arrays? Advantages? Disadvantages?

String List

- To overcome the disadvantages of the basic array, we will make list backed by an array of Strings that dynamically resizes as needed.

- What happens when we want a list of integers? Or a list of Buttons?

Object List

- Instead of using a String array to back the list, we'll just use an Object array, that way we can store anything, right?
- What's wrong with this?

Generics

- Generics allow a programmer to abstract away the type of object you are using and focus on functionality.
- Allows for code reuse.
- Works with inheritance/polymorphism.

Generics

Class declaration:

```
public class TestClass<T>
```

Method declaration:

```
public <K> String doAThing(K thing)
```

Generic Data Structures

- ArrayList<E> - array backed list
- LinkedList<E> - node backed list
- Queue<E> - FIFO
- Stack<E> - LIFO
- PriorityQueue<E> - highest priority first
- Set<E> - no order
- HashTable<K, V> - dictionary (key value pairs)

ArrayList<E>

- Array backed list. Ordered.
- Random access, fast adds, dynamically sized.
- Inserts/Removes slow (require copying and moving large portions of list).
- Items are stored sequentially in memory.

LinkedList<E>

- Node backed list. Ordered.
- Has “random access” (all indexed lookups require a list traversal).
- Fast adds, removes, and inserts.
- Takes up more memory than array, and nodes can be stored at separate locations in memory.

Queue<E> (interface)

- First in, first out “list” (often implemented as a linked structure, but not always).
- Implementers include LinkedList<E>, Deque<E>, and PriorityQueue<E>
- Used for holding elements before being processed (great for multithreading management).
- Offer (add), Poll (remove), and Peek (look).

PriorityQueue<E>

- Implements Queue<E>.
- NOT FIFO.
- As elements enter the queue they are assigned a priority based on some criteria.
- The element with the highest priority exits the queue first.
- Determined by a Comparator<E> object.

Stack<E>

- Last in, first out.
- Similar to Queue<E>, but only allows add/remove from the same end of the structure.
- Fast adds and removes.
- Useful when building state machines and interpreters.

HashTable<K, V>

- Dictionary of key value pairs.
- Values are stored in the structure using a key value which allows the data to be found quickly.
- Unordered (kind of).
- Great for storing json (common when using web services), or named settings values.

Mayhem

- All of the classes just covered are NOT THREAD SAFE (cannot be accessed from multiple threads without causing mayhem)

Blocking Collections

- Most of the collections discussed here have “Blocking” variants that are thread safe such as PriorityBlockingQueue<E>.

Questions?