Today …

- Assignments
  - “Tutorial” project
- Listener (observer) pattern
- Best practices
- Lab 2
- Readings:
  - Ch. 6 pages 255-259
  - Ch. 7 pages 281-285
Compiling on ada

- You need to use the -1.6 flag and set the classpath for multi-file compilation

```
javac -1.6 -cp . XYZ.java
```

- If you do not want the warning messages:

```
javac -1.6 -cp . -nowarn XYZ.java
```

Listener (aka observer) pattern

[Diagram of the listener (aka observer) pattern showing the relationships between Listener, Source, ConcreteListener1, ConcreteListener2, ConcreteSource1, and ConcreteSource2 with method signatures such as update(), notify(), action(), and annotations like “is-a” and “has-a” relationships.]
Listener (aka observer) pattern

- There is often an object passed to `update()`
  - Could be the subject object that the triggered the change
  - Or an event object that contains information about the change (including a reference to the source)
Listener (aka observer) pattern

public interface ActionListener {
    public void actionPerformed(ActionEvent event);
}

Public class ActionEvent extends AWTEvent {
    public ActionEvent(Object source, int id, String cmd) {...}
    ...
    public String getActionCommand() {...}
}

In EventObject (which AWTEvent extends):

    public Object getSource() {...}

Best Practices

• Here are a few best practices
• We’ll discuss more throughout the course
• This is just a start
Best Practices: Object Oriented Design

Minimize accessibility of classes and fields

• Instance variables should never be public

• Classes should be immutable unless there is a very good reason to make them mutable
  – Immutable classes have no setters!
  – Data members are final (const)

• If a class cannot be immutable, limit mutability as much as possible
  – Only provide setters if absolutely necessary

Best Practices: Always override toString()

A good toString() makes your class more pleasant to use

• When practical, toString() should return all of the interesting information in the object

  public String toString() {
    return getTitle() + " (Deadline Date = " + theDate + ")";
  }
  – Ideally, string returned should be self-explanatory

• Clearly document your conventions
  – E.g., give an example in your doc comments

• Provide getters for all information in the value returned by toString()
**Best Practices: Object creation**

*Avoid creating unnecessary objects*

• Never do this:
  ```java
  String s = new String("stringette");
  ```

• Creates a new object that has an identical value as “stringette”

• This is inefficient … just use “stringette”
  ```java
  String s = "stringette";
  ```

*Don’t keep unnecessary objects around*

• If you are done with an object **assign it to null**
  – Helps avoid “memory leaks” by removing references
  – Remember the garbage collector …

• For example:
  ```java
  public Object pop() {
      if(size != 0) {
          Object result = elements[--size];
          elements[size] = null;
          return result;
      }
      return null;
  }
  ```
Best Practices: Annotations

When overriding always use the @Override annotation

- Use the @Override tag to tell the compiler you are overriding a method
  ```java
  @Override public String toString() { ... }
  ```

- The compiler checks to see if you are overriding an actual method
  ```java
  @Override public Object equals(Todo otherObject) {...}
  ```
  - Oops, equals takes a parameter of type object
  - The compiler will warn you about this
  - This can save a lot of debugging time!

Best Practices: Static

Use static methods and fields sparingly

- As a rule of thumb, any class that could have multiple objects
  - should only use static for constants ...
  - and methods that are object-independent (“utility” methods)
  - i.e., a method that returns the same result, regardless of objects

- It is OK to use static in “launcher” classes
  - the application class that contains main(…)
  - but only if there would never by multiple instances of this class
  - Note some of your regular classes may have main(…) for testing
Lab 2