Today …

• The Object class

• Readings:
  – Same as last time: Ch. 5 pages 192-198
  – Ch. 6 pages 249-255
**Object Refresher**

- A class that does not extend another class *implicitly* extends `Object`
- Thus all classes are derived (inherit) from `Object`
- This allows us to do things like this in Java:
  ```java
  Object[] myList = new Object[10];
  myList[0] = new Car();
  myList[1] = new Fish();
  myList[2] = new Dog();
  myList[3] = new Guitar();
  etc.
  ```

**The Object class**

Inheriting from `Object` also means
- `Object` methods can be called on *all objects*

So what are these methods?
- What do they do?
- When should they be used?
- What are their default implementations?
- How should we override them?

We'll focus on the `Object` methods:
- `toString()`, `clone()`, `equals()`, and `hashCode()`

*Look at these today...*
The `toString()` Object method

```java
public String toString()
```

What does it do?
- Returns a string representation of an object

When should it be used?
- Whenever a string representation is needed
- Implicitly called with String `concatenation`
- For example, these are equivalent

```java
System.out.println("myObj = " + myObj);
System.out.println("myObj " + myObj.toString());
```

The `toString()` Object method

What is the default implementation (i.e., in `Object`)?
- Return the class name plus the object's "hashcode" value
  ```java
  return getClass().getName()
  + ' @'
  + Integer.toHexString(hashCode());
  ```
- Why do you think `getClass()` is called here?
- We'll talk about the `hashCode()` method next time …
- Recommended to always override `toString()`
The `toString()` Object method

How should we override `toString()`?

- Return all of the “interesting” information in the object

```java
public String toString() {
    return super.toString() + "[Deadline Date = " + theDate + "]";
}
```

- Return string should be “self explanatory”

- Provide “getter” functions for each value returned

- If possible, it is nice if it can follow the convention:

```java
return super.toString() + " " + ... objects important fields
```

What the book suggests

Example 1

Our Todo class without overridden `toString()`

```java
public class Todo {
    private String title;
    ...
    public static void main(String[] args) {
        System.out.println(new Todo("dishes"));
    }
}
```

```
$ java Todo
Todo@7f12f614
```

This doesn’t seem “self-explanatory”
Example 2

Our Todo class with overridden toString()

```java
public class Todo {
    private String title;
    ...
    public String toString() {
        return "Todo (title = \\
            + getTitle() + \\
    ”;
    }
    public static void main(String[] args) {
        System.out.println(new Todo("dishes"));
    }
}
```

This seems more "self-explanatory"

Using JList and overriding toString()

- When populating a list, e.g.:
  ```java
  DefaultListModel model = new DefaultListModel();
  model.addElement("Arlo"); // any object
  model.addElement("Cosmo");
  model.addElement("Elmo");
  ```

- Added items are shown via their toString() results
- But, in our TodoPlanner we only want to display titles

Our list would look like this if we simply called:
```java
model.addElement(task1);
model.addElement(task2);
```
Using JList and overriding toString()

• How can we solve this problem?

One Solution:

– Define toString() in Todo to only return the title
– Do not override toString() in subclasses of Todo
  • For example, declare it final in Todo
– This goes against our “best practices” for toString()
  • Subclasses can’t display their “important” state
  • One philosophy: toString() is really more for debugging, etc.
  • That is, a friendly way to print and see what the object is

Another Solution:

– Add the titles and not the todo objects themselves

        model.addElement(task1.getTitle());

– Works in this case since the planner requires unique titles

– We must use the titles to get associated Todo items from the planner

        Todo t = planner.getTodo(title);
Using JList and overriding toString()

Another similar solution:

- Create a “wrapper” class and add the wrapper objects

  public class TodoListItem {
      private Todo todo;
      TodoListItem(Todo theTodo) {todo = theTodo;}
      public Todo getTodo() {return todo;}
      public String toString() {return getTodo().getTitle();}
  }

- This is a slightly more general solution
  - Doesn’t rely on a certain toString() implementation

- Can be more efficient since we don’t need to lookup
todo’s in the planner by title … we have a direct reference

Using JList and overriding toString()

A more complicated solution:

- Write a custom list cell renderer

  list.setCellRenderer(new DefaultListCellRenderer() {
      public Component getListCellRenderer(JList l, Object o, int i,
          boolean sel, boolean focus)
      {
          JLabel label =
              (JLabel)super.getListCellRenderer(l, o, i, sel, focus);
          label.setText(((Todo)o).getTitle());
          return label;
      }
  });

- For details see ListCellRenderer and DefaultListCellRenderer
The clone() Object method

- We know that assignment \textit{does not} mean copy ...

  TodoPlanner \texttt{p1} = new TodoPlanner();
  TodoPlanner \texttt{copy} = \texttt{p1};

- In this case, both \texttt{p1} and \texttt{copy} refer to the \textit{same} object

- So \texttt{copy} isn’t really a “copy” of \texttt{p1}

The clone() method on Object can help here

---

The clone() Object method

protected Object clone()
    throws CloneNotSupportedException

What does it do?

- Returns a copy of an object

When should it be used?

- When a copy is needed
- Need to be sure the clone() method is implemented for the object you want to copy
The clone() Object method

What is the default implementation (i.e., in Object)?

– Creates a “shallow” copy of the object
– A shallow copy will create a new object s.t. \( p1 \neq \text{copy} \)
– However, it doesn’t (recursively) copy the variables of \( p1 \)
– What does this mean for our example?

\[
p1 = \begin{array}{c}
\text{: TodoPlanner} \\
\text{todos } = \begin{array}{c}
\text{: TodoPlanner} \\
\text{todos } = \begin{array}{c}
\text{: ArrayList}
\end{array}
\end{array}
\end{array}
\]

\[
\text{copy } = \begin{array}{c}
\text{: TodoPlanner} \\
\text{todos } = \begin{array}{c}
\text{: TodoPlanner} \\
\text{todos } = \begin{array}{c}
\text{: ArrayList}
\end{array}
\end{array}
\end{array}
\]

The clone() Object method

• Ideally, we could call clone() to copy the object
  
  ```java
  TodoPlanner p1 = new TodoPlanner();
  TodoPlanner copy = p1.clone();
  ```

• But there are some problems with this
  
  – First, clone() returns an object of type Object
  – So we have to cast (not a big deal)
  – Second, things are not this simple …
  – Note that clone() is a protected method in Object
  – And, Java adds some additional hurdles
The clone() Object method

• Because clone() has protected access
  – The method is technically inherited
  – But not visible to clients of the object
  – This is the definition of protected visibility

• Also, unless a class implements Cloneable
  – Calling clone() will throw a CloneNotSupportedException
  – This is just a “marker” interface (no methods)
    • Note also an “unusual” hurdle – not typical of Java class designs

The clone() Object method

So, if we wanted to use clone() with our planner

  – We must override clone() in TodoPlanner
  – And change its visibility to public
  – We also need TodoPlanner to implement Cloneable

There is no reason to actually do this for our planner class ... this is just used here as an example!
The clone() Object method

How to override clone()

– Use super.clone() to create the object
– Do a “deep” copy on variables (as needed)
  • Unlike a shallow copy, a deep copy recursively calls clone()

```
public TodoPlanner clone()
    throws CloneNotSupportedException
{
    TodoPlanner copy = (TodoPlanner)super.clone();
    copy.todos = todos.clone(); // hope this works
    return copy;
}
```

This doesn’t work! Instead we need this… Why?
```
copy.todos = (ArrayList)((ArrayList)todos).clone();
```

The clone() Object method

• Now if we call clone() to copy the object

```
TodoPlanner p1 = new TodoPlanner();
TodoPlanner copy = p1.clone();
```

• We end up with what we would expect …

_Can you find another potential issue here though?_
The clone() Object method

• In general, you rarely need to override clone()
  – According to the textbook, less than 5% of the classes in the standard Java library implement clone()

• It is difficult to do, for a reason
  – Cloning is tricky to get right, especially with inheritance

• Understanding how it works is useful though
  – Sometimes you may need to implement it
  – It emphasizes the use of references in Java

Exercise 8
Next time …

• We’ll talk about `equals()` and `hashCode()`

• These can be even trickier to get right than `clone()`
  – Although fewer hurdles for implementing them