Lecture 23

We consider inheritance further. We look at inheritance of instance methods. We see how inheritance of instance methods can be overridden, and we contrast overriding with overloading. And then we look at how inheritance affects the writing of constructors.

Inheritance of instance methods

- We saw that a subclass inherits instance variables from its superclass, and it can extend its superclass by defining additional instance variables.
- Similarly, a subclass can inherit instance *methods* and it can extend the behaviour of its superclass by defining additional instance methods
- Often methods in the superclass will be `public` so the subclass will have no visibility problems; but in those cases where they are `private`, you may have to consider making them `protected`


```
public class Publication
{  public void setTitle(String theTitle)
   {  title = theTitle;
   }
   public String getTitle()
   {  return title;
   }
   protected String title;
   protected PersonName author;
   protected int yearOfPub;
}

public class ConferencePaper
             extends Publication
{  public int getLength()
   {  return endPage - startPage + 1;
   }
   private String confName;
   private int startPage;
   private int endPage;
}
```

— objects of class `ConferencePaper` have 6 instance variables and 3 instance methods

I don't want to inherit all the instance methods!

- You can *override* an inherited method, i.e. supply a definition of your own *in place of* the one that would be inherited. E.g. `Publication`'s `getTitle` can be overridden by a definition in `ConferencePaper`
- Your new definition will have the same name, parameter list and return type but a different body
- (If visible), `Publications`'s `getTitle` can still be invoked from `ConferencePaper`'s instance methods by invoking `super.getTitle()`
- The subclass can't offer fewer services. Its definitions must be equally or more visible than the superclass' (e.g. if the superclass definition is `public`, the subclass definition must be `public`; if the superclass definition is `protected`, the subclass definition must be `protected` or `public`)

Overloading versus overriding

- *Overloading:* Two methods in the same class definition with the same name but different parameter lists
- *Overriding:* Two methods, one in a subclass, the other in a superclass, with the same signatures (name, parameter list and return type)

```
public class One
{  public void meth(int x)      // A
   { ...
   }

   public void meth(double x)   // B
   { ...
   }
}
```



```
public class Two extends One
{  public void meth(boolean x) // C
   { ...
   }
   public void meth(String x)  // D
   { ...
   }
   public void meth(int x)     // E
   { ...
   }
}
```

- In `One`, `meth` is overloaded with 2 definitions
- In `Two`, E overrides A; `meth` is overloaded with 4 definitions (B, C, D, E); and I can still invoke A from within `Two`'s instance methods (using `super.meth(int)`)
- Both overloading and overriding should be used sparingly

Inheritance facilitates re-use

- Suppose your project requires a class definition that is similar but not identical to an existing one (e.g. one in a library)
- Define a subclass of the existing class
  - inherit what you like
  - override what you don't like
  - extend with additional methods and variables if you want
- You can do this irrespective of whether you have the source code of the existing class definition or just the byte code.

Creating objects

- Object creation in client code, e.g.:

```
p = new Publication("Rabbit plague",
   new PersonName("Dr", "Mick",
   "Somatosis"), 1972);

cp = new ConferencePaper(
   "Chicago weather patterns",
   new PersonName("Dr", "Wendy", "City"),
   1984, "Climate Conference", 23, 45);
```

- `Publication` would have a constructor:

```
public Publication(String theTitle,
   PersonName theAuthor, int theYearOfPub
{  title = theTitle;
   author = theAuthor;
   yearOfPub = theYearOfPub;
}
```


Constructors are not inherited!

- So, would this be the `ConferencePaper` constructor?

```
public ConferencePaper(String theTitle,
   PersonName theAuthor,
   int theYearOfPub, String theConfName,
   int theStartPage, int theEndPage)
{  title = theTitle;
   author = theAuthor;
   yearOfPub = theYearOfPub;
   confName = theConfName;
   startPage = theStartPage;
   endPage = theEndPage;
}
```

- Ideally, to avoid duplication of code, the constructor in `ConferencePaper` should make use of the one in `Publication`. In fact, not only is this desirable, it is mandatory

Invoking the superclass constructor

- The first statement in `ConferencePaper`'s constructor must invoke a constructor in `Publication`
- This is done using `super`

```
public ConferencePaper(String theTitle,
   PersonName theAuthor,
   int theYearOfPub, String theConfName,
   int theStartPage, int theEndPage)
{  super(theTitle, theAuthor, theYearOfPu
   confName = theConfName;
   startPage = theStartPage;
   endPage = theEndPage;
}
```


More details on `super`

- Don't confuse this use of `super`, which applies only to the very first statement in a constructor body, with the other use of `super`
- There must be a constructor in the superclass with a matching signature
- If you leave out an invocation of `super`, Java implicitly puts one in for you anyway. Specifically, it invokes `super()`, which means you must then have a zero-parameter constructor in the superclass, otherwise you'll get a compiler-error