Lecture 10

We write another class definition, this one is for simple library books. Again we see constructors, instance methods and instance variables.

We also explain the concept of *overloading*.

What do we want the `LibraryBook` class to offer?

Some client code

```
PersonName bookAuthor =
   new PersonName("Sir", "Hugh", "Jeegoh");
LibraryBook b1 = new LibraryBook(bookAuthor,
   "Java through Dance");
LibraryBook b2 = new LibraryBook(
   new PersonName("Ms.", "Helen", "Back"),
   "Java: Who Gives a #*$%?");
b2.acquire();
b1.takeOut();
b2.takeOut();
b2.takeOut();
b2.bringBack();
b1.display();
b2.display();
```


The `LibraryBook` interface

Constructor

```
public LibraryBook(PersonName theAuthor,
   String theTitle)
```

Getters

```
public int getNumOfCopiesPossessed()
public int getNumOfCopiesOnLoan()
public int getNumOfCopiesOnShelves()
public void display()
```

Setters

```
public void takeOut()
public void bringBack()
public void acquire()
```

(A useful class might offer many more.)

The `LibraryBook` class definition

(Comments essential but omitted to save space.)

```
public class LibraryBook
{
   public LibraryBook(PersonName theAuthor,
      String theTitle)
   {


   }

   public int getNumOfCopiesPossessed()
   {
   }

   public int getNumOfCopiesOnLoan()
   {
   }

   public int getNumOfCopiesOnShelves()
   {
   }
```



```
   public void display()
   {


   }

   public void takeOut()
   {
   }

   public void bringBack()
   {
   }

   public void acquire()
   {
   }



}
```


Improving the `LibraryBook` interface

- Suppose we want *another* constructor, one that allows us to specify how many copies we possess
- Suppose we want another method for when we acquire more than one copy of a book at a time. Most naturally, it will have the same name as the existing method

```
public LibraryBook(PersonName theAuthor,
   String theTitle, int theNumOfCopies)
public void acquire(int theNumOfNewCopies
```

- Some client code:

```
LibraryBook b3 = new LibraryBook(
   bookAuthor, "Java for Vegetarians", 4)
b3.display();
b1.acquire(2);
b1.display();
```

```
public LibraryBook(PersonName theAuthor,
   String theTitle, int theNumOfCopies)
{




}

public void acquire(int theNumOfNewCopies
{
}
```


Overloading

- *Overloading of constructors:* Having two or more constructors within a class definition
- *Overloading of methods:* Having two or more methods with the same name within a class definition
- The parameter lists must be different (different number of parameters or different types in different positions). (The return type can be different or the same. But you cannot overload a method by changing only the return type.)
- Java will know which constructor/method to invoke. E.g. which definitions do these refer to:

```
LibraryBook b = new LibraryBook(
   bookAuthor, "Unjust Java");
b.acquire(3);
```


Exercise

Suppose we have defined the following

```
public int someMethod(int x, int y)
```

Which of these can be included in the same class definition as the above?

```
public int someMethod(int x, int y)

public void someMethod(int x)

public boolean someMethod(int x, int y)

public int someMethod(int x, double y)

public int someMethod(int x, int y,
   double z)

public int someMethod(int y, int x)

public int someMethod(int a, int b)
```


When to use overloading

- *Overloading of constructors:*
  - very common, see, e.g., `String`, `Color`,...
  - a good way of having optional parameters
- *Overloading of methods:*
  - less common, e.g. `indexOf` and `substring` in `String`
  - use only when the methods have very nearly identical behaviour: saves inventing different names & signals to readers the methods' similarity
  - if you use overloading for unrelated methods, your program is less understandable

Overloading of operators

- Java has overloaded operators too: one operator may have several definitions
- `/` applied to two `int`s is integer division; `/` applied to at least one `double` is floating point division
- `+` applied to two `int`s is integer addition; `+` applied to at least one `double` is floating point addition; `+` applied to at least one `String` is concatenation
- However, unlike C++, Java doesn't allow us to provide operators with further definitions of our own