Chapter 7

Defining Your Own Classes
Part 2

CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University
Objectives

This week we will study:
- methods returning objects
- the reserved word this
- overloading
- class methods and variables
- pass-by-value parameter passing
- javadoc comments
- packages
Returning an Object from a Method

- As we can return a primitive data value from a method, we can return an object from a method also.

- When we return an object from a method, we are actually returning a reference (or an address) of an object.
  - This means we are not returning a copy of an object, but only the reference of this object.
Example class: Fraction

- Consider a class that stores fractions.
- It has two data members
  - `int numerator, denominator`.
- The methods include:
  - `getNumerator()`, `setDenominator()`, ...
  - `add(Fraction)`, `subtract(Fraction)`, `simplify()`
  - `toString()`
- A `toString` method usually produces the value of the object as a string.
Here's a sample method that returns an object:

```java
public Fraction simplify() {
    Fraction simp;
    int num   = getNumberator();
    int denom = getDenominator();
    int gcd   = gcd(num, denom);
    simp = new Fraction(num/gcd, denom/gcd);
    return simp;
}
```

Return type indicates the class of an object we're returning from the method.

Return an instance of the Fraction class
A Sample Call to `simplify`

```java
public Fraction simplify() {
    int num = getNumerator();
    int denom = getDenominator();
    int gcd = gcd(num, denom);

    Fraction simp = new Fraction(num/gcd, denom/gcd);
    return simp;
}
```

```java
f1 = new Fraction(24, 26);
f2 = f1.simplify();
```
A Sample Call to **simplify**

```java
f1 = new Fraction(24, 26);
f2 = f1.simplify();
```

```java
public Fraction simplify() {
    int num   = getNumerator();
    int denom = getDenominator();
    int gcd   = gcd(num, denom);

    Fraction simp = new Fraction(num/gcd, denom/gcd);

    return simp;
}
```

The value of `simp`, which is a reference, is returned and assigned to `f2`. 
The reserved word *this* is called a *self-referencing pointer* because it refers to an object from the object's method.

- The reserved word *this* can be used in three different ways. We will see all three uses this week.
public Fraction add(Fraction frac) {
    int a, b, c, d;
    Fraction sum;
    a = this.getNumerator();   //get the receiving
    b = this.getDenominator(); //object's num and denom
    c = frac.getNumerator();   //get frac's num
    d = frac.getDenominator(); //and denom
    sum = new Fraction(a*d + b*c, b*d);
    return sum;
}
Because \( f1 \) is the receiving object (we're calling \( f1 \)'s method), so the reserved word \textit{this} is referring to \( f1 \).
f3 = f2.add(f1)

This time, we're calling f2's method, so the reserved word this is referring to f2.
In the previous example, we showed the use of 
**this** to call a method of a receiving object.

It can be used to refer to a data member as well.

```java
class Person {
    int age;
    public void setAge(int val) {
        this.age = val;
    }
    ...
}
```

Note that **this** is usually implied and can be omitted. However, if we have a local variable or parameter with the same identifier as a data member, then **this** can be used to refer to the data member.
What does this mean?

class Person {
    int age;

    public void setAge(int age) {
        age = age;
    }
    ...
}

class Person {
    int age;

    public void setAge(int age) {
        this.age = age;
    }
    ...
}
Overloaded Methods

Methods can share the same name as long as
- they have a different number of parameters (Rule 1) or
- their parameters are of different data types when the number of parameters is the same (Rule 2)

```java
public void myMethod(int x, int y) { ... }
public void myMethod(int x) { ... }
```

Rule 1

```java
public void myMethod(double x) { ... }
public void myMethod(int x) { ... }
```

Rule 2
Method signature

- The method name, and type of parameters (in order) is called a method signature.
  E.g.
  
  ```java
  myMethod(int, String)
  myMethod(int)
  getSalary(JOptionPane, float)
  getSalary(float, JOptionPane)
  ```

- A method can be overloaded as long as the signature is distinct from others.

- The signature is used to determine which method to call.

- IMPORTANT: return type is not part of the signature.

- Methods cannot differ only in return type!
Examples

Rule 1

```
public void myMethod(int x, int y) { ... }
public void myMethod(int x) { ... }
```

Rule 2

```
public void myMethod(double x, int y) { ... }
public void myMethod(int x, double y) { ... }
```

```
public void myMethod(double x, int y) { ... }
public int myMethod(double x, int y) { ... }
```
Overloaded Constructor

- The same rules apply for overloaded constructors
  - this is how we can define more than one constructor for a class

```java
public Person() { ... }
public Person(int age) { ... }
```

```java
public Pet(int age) { ... }
public Pet(String name) { ... }
```

Rule 1

Rule 2
Constructors and **this**

- To call a constructor from another constructor of the same class, we use the reserved word **this**.

```java
public Fraction( ) {
   //creates 0/1
   this(0, 1);
}
public Fraction(int number) {
   //creates number/1
   this(number, 1);
}
public Fraction(Fraction frac) {
   //copy constructor
   this(frac.getNumerator(), frac.getDenominator());
}
public Fraction(int num, int denom) {
   setNumerator(num);
   setDenominator(denom);
}
```
Copy constructor

- A copy constructor can be very handy.
- It takes an object as input and creates another object (of the same class) and copies the values.
- Useful also for preventing surreptitious access to private objects.
- If a method returns a pointer to a private object, then the client can modify the private object!
- Avoid this by returning a copy object
Use of a copy constructor

```java
class Jedi {
    private Person father;

    public void Jedi(Person f) {
        father = f;
    }

    Person getFather() {
        return father;
    }
}
```

class Corruptor {
    Jedi luke;
    Person p;
    public static void main(String[] args) {
        luke = new Jedi(new Person("ObiWan"));
        p = luke.getFather();
        p.setName("Darth Vader");
        p = luke.getFather();
        System.out.println(p.getName());
    }
}
```
The reserved word this can be used to

1. Access methods of the same class from another method of the class.
   ```java
   this. getNumerator(); numerator = 5;
   ```

2. Access data members of the class from a method of the class
   ```java
   this. numerator = 5;
   ```

3. Call the constructor of the class
   ```java
   this(0,1);
   ```
Class (static) Methods

- Some methods do not access object data members. No need to call them on objects.
- These can be declared as class methods.
- They are called by using the class name.
  - E.g. the Math class methods.
- We use the reserved word `static` to define a class method.

```java
public static int gcd(int m, int n) {
    //the code implementing the Euclidean algorithm
}

public static Fraction min(Fraction f1, Fraction f2) {
    //convert to decimals and then compare
}
```
Static methods

- IMPORTANT: a static method cannot access any instance data members or instance methods
  - I.e. it can only access other static members and methods

- Note that main is a static method!
  - No object is necessary to run main.
  - But, it can’t call non-static methods.
  - Recall Chapter 4 LoanCalculator
Calling non-static methods

```java
class LoanCalculator {
    public static void main (String[] arg) {
        LoanCalculator calculator = new LoanCalculator();
        calculator.start();
    }

    public void start() {
        ...
    }
}
```

- `main` cannot call `start` because it is an instance method.
- Thus, we have to create a `LoanCalculator` object and call `start()` on it.
Class (static) variables

- Similarly, we can define class data members:  
  ```java
  static int account;
  ```
- These are not stored for each object of the class -- only once for the entire class.
- Accessed using class name.
- Class constants are examples too.  
  ```java
  static final int YES_OPTION = 1;
  ```
- They can be very useful.
  - E.g. Suppose each bank account should have a number of the type US_100, US_101, …
Using class variables

class Account {
    private static int nextAccount = 100;
    private static final String PREFIX = "US_";
    private String accountNumber;
    public Account() {
        accountNumber = PREFIX + Account.nextAccount;
        Account.nextAccount++;
    }
    public String getAccountNumber() {
        return accountNumber;
    }
}

class Test {
    Account acc1, acc2;
    acc1 = new Account();
    acc2 = new Account();
    System.out.println("Acc1: " + acc1.getAccountNumber());
    System.out.println("Acc2: " + acc2.getAccountNumber());
}
Account
- PREFIX
- nextAccount

+ Account(): void
+ getAccountNumber(): String

acc1 = new Account();

acc2 = new Account();
Static Initializer

- Earlier, we initialized static variables upon declaration. This initialization takes place when the class is loaded.
  - Imported or used for the first time in a program.
- What if we want to do more?
  - E.g. set the initial value based upon user input?
- We can define a static initializer segment that gets executed when a class is loaded.
As with static methods, we cannot reference any non-static method or data member from the static initializer block.
null is a special value. Its type is that of a reference to an object (of any class).

We can set an object identifier to this value to show that it does not point to any object.

- Bicycle bike1=null;

A method that returns objects (of any class) can return a null value.

Note that you will get a run-time error if you access a data member of call a method of a null object -- null pointer exception.
Testing for null values.

class Account {
    private Person owner;
    public Account()
    {
        owner = null;
    }
    public void setOwner(Person p)
    {
        owner = p;
    }
    public Person getOwner()
    {
        return (owner);
    }
}

class Bank {
    public static void main(String[] arg)
    {
        Account acc = new Account();
        Person p;
        ...
        p = acc.getOwner();
        if (p == null)
        {
            System.out.println("No owner");
        }
        ...
    }
}

We can use == or != to check if an object reference is null or not.
Examples of class methods

- The Math class has numerous class methods and constants
  - Math.abs, Math.pow,
  - Math.PI

- We have also seen Wrapper classes for the primitive data types:
  - Integer: Integer.parseInt, Integer.MAX_VALUE
  - Double: Double.parseDouble, ...
  - Similarly for long, short, byte, and boolean.
Wrapper Classes

- There is a wrapper class for each primitive data type.
- Wrapper classes create an object out of a primitive data type. What is the use?
- We can provide extra functionality
  - Parsing, converting to and from strings
  - Type casting (doubleValue), toHexString
  - Constants such as PI, E, MAX_VALUE
  - See java API for details
- Sometimes we must have an object.
Boxing and Unboxing

- Converting a primitive to its corresponding wrapper is called **boxing**.
- Converting a wrapper to its corresponding primitive type is called **unboxing**.
- In earlier versions of Java converting was a little painful:
  - Integer iObj = new Integer(5);
  - int j = iObj.intValue();
- Java 5 does this automatically:
  - Integer iObj = 5;
  - int j = iObj;
Autoboxing, Auto-unboxing

- Automatic boxing and unboxing is convenient.
- This also works when converting actual to formal parameters.
- E.g. if a method expects an Integer object, you can (in Java 5) simply pass an int value or int literal

```java
addToCollection(Integer iObj){ ... }
addToCollection(5);
addToCollection(i);
```
Overloading and Type casting

- Automatic (un)boxing and type casting is used with overloaded methods.
- BUT, only if no matching method is found before automatic type conversion.

```java
int i;
doMyThing(i);
```

This call can match:

1. `doMyThing(int)` OR
2. `doMyThing(long)` `doMyThing(Integer)` `doMyThing(double)`

**BE CAREFUL!**
Call-by-Value Parameter Passing

- When a method is called,
  - the value of the argument is passed to the matching parameter, and
  - separate memory space is allocated to store this value.

- This way of passing the value of arguments is called a *pass-by-value* or *call-by-value scheme*.

- Since separate memory space is allocated for each parameter during the execution of the method,
  - the parameter is local to the method, and therefore
  - changes made to the parameter will not affect the value of the corresponding argument.
Call-by-Value Example

class Tester {
    public void myMethod(int one, double two) {
        one = 25;
        two = 35.4;
    }
}

Tester tester;
int x, y;
tester = new Tester();
x = 10;
y = 20;
tester.myMethod(x, y);
System.out.println(x + " " + y);

produces
10 20
Memory Allocation for Parameters

```java
Tester tester;
int x, y;
tester = new Tester();
x = 10;
y = 20;
tester.myMethod(x, y);
System.out.println(x + " " + y);

class Tester {
    public void myMethod(int one, double two ){
        one = 25;
        two = 35.4;
    }
}
```

1. x = 10, y = 20
2. one = 25, two = 35.4
Memory Allocation for Parameters

```java
class Tester {
    public void myMethod(int one, double two) {
        one = 25;
        two = 35.4;
    }
}

Tester tester;
int x, y;
tester = new Tester();
x = 10;
y = 20;
tester.myMethod(x, y);
System.out.println(x + " " + y);
```

```java
int x, y;
tester = new Tester();
x = 10;
y = 20;
tester.myMethod(x, y);
System.out.println(x + " " + y);
```
Parameter Passing: Key Points

1. Arguments are passed to a method by using the pass-by-value scheme.
2. Arguments are matched to the parameters from left to right. The data type of an argument must be assignment-compatible with the data type of the matching parameter.
3. The number of arguments in the method call must match the number of parameters in the method definition.
4. Parameters and arguments do not have to have the same name.
5. Local copies, which are distinct from arguments, are created even if the parameters and arguments share the same name.
6. Parameters are input to a method, and they are local to the method. Changes made to the parameters will not affect the value of corresponding arguments.
7. When we pass objects, the reference to the object is passed, therefore, objects get modified.
Organizing Classes into a Package

- For a class A to use class B, their bytecode files must be located in the same directory.
  - This is not practical if we want to reuse programmer-defined classes in many different programs.
- The correct way to reuse programmer-defined classes from many different programs is to place reusable classes in a package.
- A package is a Java class library.
Creating a Package

The following steps illustrate the process of creating a package named `myutil` that includes the `Fraction` class.

1. Include the statement

   ```java
   package myutil;
   ```

   as the first statement of the source file for the Fraction class.

2. The class declaration must include the visibility modifier public as

   ```java
   public class Fraction {
   ...
   }
   ```

3. Create a folder named `myutil`, the same name as the package name. In Java, the package must have a one-to-one correspondence with the folder.

4. Place the modified Fraction class into the `myutil` folder and compile it.

5. Modify the CLASSPATH environment variable to include the folder that contains the `myutil` folder.
Multiple classes per file

- So far, we only defined a single class per file.
- It is possible to define multiple classes in a single file.
- However, only one file can be public. This class must have the same name as the file (.java).
- Only public classes can be imported.
Program Structure and Source Files

class Test1 {
    ...
}
class Test2 {
    ...
}

Two class files are created. Both are available to classes in the same directory.

javac Test.java

Test1.class
Test2.class
Program Structure and Source Files

```java
class Test1 {
    public static void main (String[] arg) {
        ...
    }
}
class Test2 {
    public static void main (String[] arg) {
        ...
    }
}
```

Can execute:
- `%java Test1`
- `%java Test2`
Program Structure and Source Files

```java
package myPack;
public class Test1 {
    public static void main (String[] arg) {
        ...
    }
}
class Test2 {
}
```

```java
import myPack;
class Tester {
    public static void main (String[] arg) {
        Test1 t;
        ...
    }
}
```

javac Test.java

Test1.class
Test2.class
myPack

CLASSPATH must include myPack

Cannot see Test2 class in Tester.java!
Using Javadoc Comments

- Many of the programmer-defined classes we design are intended to be used by other programmers.
  - It is, therefore, very important to provide meaningful documentation to the client programmers so they can understand how to use our classes correctly.

- By adding javadoc comments to the classes we design, we can provide a consistent style of documenting the classes.

- Once the javadoc comments are added to a class, we can generate HTML files for documentation by using the javadoc command.
This is a portion of the HTML documentation for the Fraction class shown in a browser.

This HTML file is produced by processing the javadoc comments in the source file of the Fraction class.
javadoc Tags

- The javadoc comments begins with /** and ends with */
- Special information such as the authors, parameters, return values, and others are indicated by the @ marker
  - @param
  - @author
  - @return
  - etc
Example: javadoc Source

```java
/**
 * Returns the sum of this Fraction and the parameter frac. The sum returned is NOT simplified.
 *
 * @param frac the Fraction to add to this Fraction
 *
 * @return the sum of this and frac
 */
public Fraction add(Fraction frac) {
   ...
}
```
Example: javadoc Output

public Fraction add(Fraction frac)

Returns the sum of this Fraction and the parameter frac. The sum returned is NOT simplified.

Parameters:
frac - the Fraction to add to this Fraction

Returns:
the sum of this and frac
javadoc Resources

- General information on javadoc is located at
  
  http://java.sun.com/j2se/javadoc

- Detailed reference on how to use javadoc on Windows is located at
  
  http://java.sun.com/j2se/1.5/docs/tooldocs/windows/javadoc.html
Problem Statement

Write an application that computes the total charges for overdue library books. For each library book, the user enters the due date and (optionally) the overdue charge per day, the maximum charge, and the title. If the optional values are not entered, then the preset default values are used. A complete list of book information is displayed when the user finishes entering the input data. The user can enter different return dates to compare the overdue charges.
Overall Plan

Tasks:

1. Get the information for all books
2. Display the entered book information
3. Ask for the return date and display the total charge. Repeat this step until the user quits.
Required Classes

- OverdueChecker
- Scanner
- BookTracker
- LibraryBook

helper class
Development Steps

- We will develop this program in five steps:
  1. Define the basic **LibraryBook** class.
  2. Explore the given **BookTracker** class and integrate it with the **LibraryBook** class.
  3. Define the top-level **OverdueChecker** class. Implement the complete input routines.
  4. Complete the **LibraryBook** class by fully implementing the overdue charge computation.
  5. Finalize the program by tying up loose ends.
Step 1 Design

- Develop the basic LibraryBook class.
- The key design task is to identify the data members for storing relevant information.
- We will include multiple constructors for ease of creating LibraryBook objects.
  - Make sure that an instance will be initiated correctly no matter which constructor is used.
Step 1 Code

Directory:  Chapter7/Step1

Source Files:  LibraryBook.java
                 Step1Main.java  (test program)
Step 1 Test

- In the testing phase, we run the test main program Step1Main and confirm that we get the expected output:

<table>
<thead>
<tr>
<th>Title</th>
<th>Price 1</th>
<th>Price 2</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title unknown</td>
<td>$ 0.50</td>
<td>$ 50.00</td>
<td>03/14/04</td>
</tr>
<tr>
<td>Introduction to OOP with Java</td>
<td>$ 0.75</td>
<td>$ 50.00</td>
<td>02/13/04</td>
</tr>
<tr>
<td>Java for Smarties</td>
<td>$ 1.00</td>
<td>$ 100.00</td>
<td>01/12/04</td>
</tr>
<tr>
<td>Me and My Java</td>
<td>$ 1.50</td>
<td>$ 230.00</td>
<td>01/01/04</td>
</tr>
</tbody>
</table>
Step 2 Design

- Explore the helper BookTracker class and incorporate it into the program.
- Adjust the LibraryBook class to make it compatible with the BookTracker class.
Step 2 Code

Directory: Chapter7/Step2

Source Files: LibraryBook.java
              Step2Main.java (test program)
Step 2 Test

- In the testing phase, we run the test main program Step2Main and confirm that we get the expected output.
- We run the program multiple times trying different variations each time.
Step 3 Design

- We implement the top-level control class OverdueChecker.
- The top-level controller manages a single BookTracker object and multiple LibraryBook objects.
- The top-level controller manages the input and output routines
  - If the input and output routines are complex, then we would consider designing separate classes to delegate the I/O tasks.
Step 3 Pseudocode

GregorianCalendar returnDate;
String reply, table;
**double** totalCharge;

inputBooks(); //read in all book information

table = bookTracker.getList();
System.out.println(table);

//try different return dates
do {
    returnDate = *read return date*;
    totalCharge = bookTracker.getCharge(returnDate);
    displayTotalCharge(totalCharge);
    reply = *prompt the user to continue or not*;
} while (reply is yes);
Step 3 Code

**Directory:** Chapter7/Step3

**Source Files:** OverdueChecker.java
LibraryBook.java
Step 3 Test

- Now we run the program multiple times, trying different input types and values.

- We confirm that all control loops are implemented and working correctly.
  - At this point, the code to compute the overdue charge is still a stub, so we will always get the same overdue charge for the same number of books.

- After we verify that everything is working as expected, we proceed to the next step.
Step 4: Compute the Charge

- To compute the overdue charge, we need two dates: the due date and the date the books are or to be returned.
- The getTimeInMillis method returns the time elapsed since the epoch to the date in milliseconds.
- By subtracting this since-the-epoch milliseconds value of the due date from the same of the return date, we can find the difference between the two.
  - If the difference is negative, then it’s not past due, so there’s no charge.
  - If the difference is positive, then we convert the milliseconds to the equivalent number of days and multiply it by the per-day charge to compute the total charge.
Step 4 Code

Directory: Chapter7/Step3

Source Files: OverdueChecker.java
LibraryBook.java
Step 4 Test

- We run the program multiple times again, possibly using the same set of input data.
- We enter different input variations to try out all possible cases for the computeCharge method.
  - Try cases such as the return date and due date are the same, the return date occurs before the due date, the charge is beyond the maximum, and so forth.
- After we verify the program, we move on to the next step.
Step 5: Finalize / Extend

Program Review

- Are all the possible cases handled?
- Are the input routines easy to use?
- Will it be better if we allow different formats for entering the date information?

Possible Extensions

- Warn the user, say, by popping a warning window or ringing an alarm, when the due date is approaching.
- Provide a special form window to enter data

(Note: To implement these extensions, we need techniques not covered yet.)