Chapter 9 – Designing Classes

Chapter Goals
- Learn to identify side effects
- Know when to use method preconditions and postconditions
- Go in depth a bit more on static (aka class) methods and fields
- Design programs the object-oriented way

9.3 Immutable Classes
- Recall that accessor methods only retrieve information
  - They do not change the state of the object
  - Calling them multiple times in a row will yield same results
- Immutable Classes are classes that only have accessor methods

Example: String Class
- Advantages
  - Do not have to worry about the dangers of handing off references in methods because there are no methods that can modify them.
- Disadvantages
  - Your objects are fixed once they are created.

9.4 Side Effects
- Mutators are designed to change the internal data of the implicit parameter
- Side effect of a method: modifying externally observable data (explicit parameters or reference types)

```java
public void transfer(double amount, BankAccount other) {
    balance = balance - amount;
    other.balance = other.balance + amount;
    // Modifies explicit parameter
}
```
- Best to avoid if possible (otherwise, document the effect thoroughly)

Other Side Effects
- Why don’t we add a printBalance() method to BankAccount?

```java
public void printBalance() {
    System.out.println("The balance is now $" + balance);
}
```
- Assumes the printing should be done to the screen
- Assumes the printing should be in English
- Makes this class dependent on the System class and PrintStream class
- You want to localize the input and output of your program to as few places as possible
How to minimize Side Effects?

- Never modify explicit parameters to a method
  - Treat them as constants

```java
public void deposit(double amount) {
    amount = amount + balance;
}
```

Pass By Value

- Two ways to pass parameters
  - Pass by reference – the memory location is sent, meaning that the data can be changed
  - Pass by value – a copy of the memory location is created

- Java only uses pass by value
  - Even objects are passed by value, since the reference is copied

Won’t work!

```java
public class BankAccount {
    public void transferTo(BankAccount other, double amount) {
        balance = balance - amount;
        double newBalance = other.balance + amount;
        other = new
    }
}
```

Neither will this

```java
class Q:
    public String addResponse(String question) {
        String response = "I accept";
        question = new String(question + response);
    }
class R:
    public void invite() {
        Q friend = new Q();
        String invite = "Want to go to the dance?";
        int inviteLength = invite.length();
        do {
            friend.addResponse(invite);
        } while(invite.length() == inviteLength);
    }
```
9.5 Preconditions

- Precondition: Requirement that the caller of a method must meet
  - Publish preconditions so the caller won’t call methods with bad parameters

```java
/**
 * Deposits money into this account.
 * @param amount the amount of money to deposit
 * (Precondition: amount >= 0)
 */
```

Preconditions

- Typical use:
  - To restrict the parameters of a method
  - To require that a method is only called when the object is in an appropriate state

- If precondition is violated, method is not responsible for computing the correct result. It is free to do anything.

Handling Violations

- Method does not have to behave correctly when preconditions are violated

- But what should be done?
  - Assume data is correct. Continue until the bad data causes another error (i.e divide by zero)
  - Throw an exception (Chapter 15) if a condition is violated. The control is shifted to an exception handler
    - Runtime error messages are all exceptions

Another option

- Method can do an assertion check
  ```java
  assert amount >= 0;
  balance = balance + amount;
  ```
  - To enable assertion checking:
    ```
    java -enableassertions MyProg
    ```
  - You can turn assertions off after you have tested your program, so that it runs at maximum speed

Common error

- Returning silently on error is hard to detect and debug

```java
if (amount < 0) return;
// Not recommended; hard to debug
balance = balance + amount;
```

Postconditions

- Condition that is true after a method has completed. This is the designer’s duty (as opposed to preconditions being the caller’s duty)

- If method call is in accordance with preconditions, it must ensure that postconditions are valid
There are two kinds of postconditions:
- The return value is computed correctly
- The object is in a certain state after the method call is completed (mutators)

```bash
/**
   Deposits money into this account.
   @param amount the amount of money to deposit
   (Precondition: amount >= 0)
   (Postcondition: getBalance() >= 0)
*/
```

Don't document trivial postconditions that repeat the @return clause

Formulate pre- and postconditions only in terms of the interface of the class
```
getBalance() >= 0
// is a better postcondition than
balance >= 0
```

Contract
- Think of pre and postconditions as a contract
- If caller fulfills precondition, method must fulfill postcondition.

Static Fields (Class Fields)
- Stores a field outside of any particular instance (object)
- Only one copy of a field that is shared by all instances of a class
- Minimize use of static for OO programs

Static Methods (Class Methods)
- Can be invoked without creating an instance.
- Can only access explicit parameters, class fields, and class methods.
- `main` method must be static

Field Modifier choices
- For each field:
  1. Choose data type and identifier.
  2. is it a constant?
     - if so, add `final` modifier
  3. does it require a different value for each instance?
     - if not, add `static` modifier
  4. is it safe to be directly accessible to methods in other classes?
     - if so, `public`
     - else, `private`
  5. is the value known?
     - if it is static, initialize now, otherwise initialize it in the constructor
Method Modifier choices

- For each method:
  1. Choose identifier, return type, and parameters
  2. does it need to access any instance (non-static) fields or instance methods?
     - if not, add static modifier
  3. Must it be called from methods in other classes?
     - if so, must be public
     - else make it private
  4. is it safe to be overridden? (chapter 13)
     - if not, add final modifier

Why make fields static?

- so that only one copy of the value is stored (all instances share the same copy)

Why make methods static?

- so that the method is accessible without creating an instance
- Note: Can only do this if the method only requires explicit parameters or static fields.

Call with class name instead of object:
double tax = Financial.percentOf(taxRate, total);

Why write a method that does not operate on an object?

Common reason: encapsulate some computation that involves only numbers. Numbers aren't objects, you can't invoke methods on them. E.g., x.sqrt() can never be legal in Java

Call with class name instead of object:
double tax = Financial.percentOf(taxRate, total);

Why write a method that does not operate on an object?

Common reason: encapsulate some computation that involves only numbers. Numbers aren't objects, you can't invoke methods on them. E.g., x.sqrt() can never be legal in Java

Call with class name instead of object:
double tax = Financial.percentOf(taxRate, total);

main is static–there aren't any objects yet

public class Financial {
    public static double percentOf(double p, double a) {
        return (p / 100) * a;
    }
    // Other methods can be added
}

- calculateMultiplier from assignment 2 probably should have been static. Why?
public class Financial
{
public static double percentOf(double p, double a)
{
return (p / 100) * a;
}
public static void main(String[] args)
{
double percent = percentOf(20.0, 10.0);
System.out.println(percent + " is 20% of 10");
}
}

Static Fields
- Three ways to initialize:
  - Do nothing. Field is with 0 (for numbers), false (for boolean values), or null (for objects)
  - Use an explicit initializer
    ```java
    BankAccount
    {
    private static int lastAssignedNumber = 1000;
    // Executed once, when class is loaded
    }
    - Use a static initialization block in a method

Scope Of Local Variables
- Scope of variable: Region of program in which the variable can be accessed
- Scope of a local variable extends from its declaration to end of the block that encloses it
- Sometimes the same variable name is used in two methods:

> public class RectangleTester
> {
>   public static double area(Rectangle rect)
>   {
>     double r = rect.getWidth() * rect.getHeight();
>     return r;
>   }
>   public static void main(String[] args)
>   {
>     Rectangle r = new Rectangle(5, 10, 20, 30);
>     double a = area(r);
>     System.out.println(r);
>   }
> }

> Scope of a local variable cannot contain the definition of another variable with the same name

```java
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
  double r = Math.sqrt(x);
  // Error-can't declare another variable
  //called r here
  
  ...
```
Fields have class scope: You can access all fields in any method of the class.
Must qualify public fields outside scope.

```
Math.PI
harrysChecking.BANK_FEE
```

A local variable can *shadow* a field with the same name
Local scope wins over class scope

```
public class Coin
{
    private String name;
    private double value; // Field with the same name
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        . . .
        return value;
    }
}
```

Packages

- Package: Set of related classes
- To put classes in a package, you must place a line
  ```
  package packageName;
  ```
  as the first instruction in the source file containing the classes
- Package name consists of one or more identifiers separated by periods

```
public class Financial
{
    . . .
}
```

For example, to put the Financial class introduced into a package named com.horstmann.bigjava, the Financial.java file must start as follows:

```
package com.horstmann.bigjava;
public class Financial
{
    . . .
}
```

So, for example, to put the Financial class into a package named com.horstmann.bigjava, the Financial.java file must start as follows:

```
package com.horstmann.bigjava;
public class Financial
{
    . . .
}
```

However, can have local variables with identical names if scopes do not overlap

```
if (x >= 0)
{
    double r = Math.sqrt(x);
    . . .
} // Scope of r ends here
else
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    // OK—it is legal to declare another r here
    . . .
}
```

Inside a method, no need to qualify fields or methods that belong to the same class
An unqualified instance field or method name refers to the this parameter

```
public class BankAccount
{
    public void transfer(double amount, BankAccount other)
    {
        withdraw(amount);
        // same as: this.withdraw(amount);
        other.deposit(amount);
    }
    . . .
}
```

A local variable can *shadow* a field with the same name
Local scope wins over class scope

```
public class Coin
{
    private String name;
    private double value; // Field with the same name
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        . . .
        return value;
    }
}
```

```
If (x >= 0)
{
    double r = Math.sqrt(x);
    . . .
} // Scope of r ends here
else
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    // OK—it is legal to declare another r here
    . . .
}
```

Fields have class scope: You can access all fields in any method of the class.
Must qualify public fields outside scope.

```
Math.PI
harrysChecking.BANK_FEE
```

A local variable can *shadow* a field with the same name
Local scope wins over class scope

```
public class Coin
{
    private String name;
    private double value; // Field with the same name
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        . . .
        return value;
    }
}
```

```
If (x >= 0)
{
    double r = Math.sqrt(x);
    . . .
} // Scope of r ends here
else
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    // OK—it is legal to declare another r here
    . . .
}
```

Fields have class scope: You can access all fields in any method of the class.
Must qualify public fields outside scope.

```
Math.PI
harrysChecking.BANK_FEE
```

A local variable can *shadow* a field with the same name
Local scope wins over class scope

```
public class Coin
{
    private String name;
    private double value; // Field with the same name
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        . . .
        return value;
    }
}
```

```
If (x >= 0)
{
    double r = Math.sqrt(x);
    . . .
} // Scope of r ends here
else
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    // OK—it is legal to declare another r here
    . . .
}
```

Fields have class scope: You can access all fields in any method of the class.
Must qualify public fields outside scope.

```
Math.PI
harrysChecking.BANK_FEE
```

A local variable can *shadow* a field with the same name
Local scope wins over class scope

```
public class Coin
{
    private String name;
    private double value; // Field with the same name
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        . . .
        return value;
    }
}
```

```
If (x >= 0)
{
    double r = Math.sqrt(x);
    . . .
} // Scope of r ends here
else
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    // OK—it is legal to declare another r here
    . . .
}
```

Fields have class scope: You can access all fields in any method of the class.
Must qualify public fields outside scope.

```
Math.PI
harrysChecking.BANK_FEE
```

A local variable can *shadow* a field with the same name
Local scope wins over class scope

```
public class Coin
{
    private String name;
    private double value; // Field with the same name
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        . . .
        return value;
    }
}
```
9.1 Choosing Classes

- Division of labor:
  - Methods = functions
  - Classes/objects = entities
- A class represents a single concept from the problem domain
- Name for a class should be a noun that describes concept

Good classes

- Concepts from mathematics:
  - Point
  - Rectangle
  - Ellipse
- Concepts from real life
  - BankAccount
  - CashRegister

Good Classes

- Actors (end in -er, -or)—objects do some kinds of work for you
  - Scanner
  - Random // better name: RandomNumberGenerator
- Utility classes—no objects, only static methods and constants
  - Math
- Degenerate case of a class: only main() method (like application and test classes)

Bad Classes

- Don’t turn actions into classes:
  - Paycheck is better name than ComputePaycheck
- The name can tip you off when you are on wrong track
  - Representing a PayCheck is more useful than only designing a class that seems to compute a pay check

Practice Question

- What are the operations needed to use the die?
- What data are needed to represent the concept of a die?
17.2 Discovering Classes

- Recall that part of the design phase is deciding what structures you need to solve a task.
- In OOD this translates into 3 steps:
  - Discover classes
  - Determine the responsibilities of each class
  - Describe relationships between each class

Class = concept

- Recall that a class represents a concept:
  - Some are concrete (i.e. real world)
    - A bank account
    - Rental items
    - Database of items
    - Pile
  - Others are abstract
    - Scanner
    - Streams, Math

Simple rule

- Look for nouns in task description (specs):
  - Obviously not all nouns are classes
  - But can create a list of candidate classes
- Then determine which ones are useful:
  - Cross them off your list

Key points

- Class represents set of objects with the same behavior:
  - Entities with multiple occurrences in problem description are good candidates for objects
  - Find out what they have in common
  - Design classes to capture commonalities
- Not all nouns need a new class:
  - Address needs to represented, do we need a new class or can we use a String?
  - Could have argument for both – but must balance generality with limiting design
Behavior

- After set of classes have been sketched up, define behavior/purpose, of each class
- Verbs = methods

CRC Card

- Describes a class, its responsibilities, and its collaborators
- Use an index card for each class
- Pick the class that should be responsible for each method (verb)
- Write the responsibility onto the class card
- Indicate what other classes are needed to fulfill responsibility (collaborators)

9.2 Cohesion

- A class should represent a single concept
- The public interface of a class is cohesive if all of its features are related to the concept that the class represents
  - methods and public constants/variables should all relate to the central idea of the class

Cohesion

- This class lacks cohesion:
public class CashRegister{
  public void enterPayment(int dollars, int quarters, int dimes, int nickels, int pennies)
  . . .
  public static final double NICKEL_VALUE = 0.05;
  public static final double DIME_VALUE = 0.1;
  public static final double QUARTER_VALUE = 0.25;
  . . .
}

- What is the problem with this lack of cohesion?
  - More confusion
  - Ties the role of a cash register to the value of the coins
  - What if we wanted to sell this cash register to other countries?
Solution
- CashRegister, as described above, involves two concepts: cash register and coin
- Solution: Make two classes:
  ```java
  public class Coin{
    public Coin(double aValue, String aName){ . . . }
    public double getValue(){ . . . }
  }
  ```
  ```java
  public class CashRegister{
    public void enterPayment(int coinCount, Coin coinType) { . . . }
  }
  ```

Coupling
- A class depends on another if it uses objects of that class
  - `CashRegister` depends on `Coin` to determine the value of the payment
  - `Coin` does not depend on `CashRegister`
- High Coupling = many class dependencies

Minimize coupling to minimize the impact of interface changes
- To visualize relationships draw class diagrams
- UML: Unified Modeling Language. Notation for object-oriented analysis and design

Figure 1
Dependency Relationship Between the CashRegister and Coin Classes

17.3 Relationships Between Classes
- Good practice to document relationship between classes
  - Can uncover common behavior
  - Divide uncommon classes among programming teams
- We have learned about inheritance as a relationship
  - 4 total important relationships
  - We’ve come across 2 so far, 2 more in upcoming chapters
Dependency

- Dependency occurs when a class uses another class methods
- *Uses* relationship
- Example: many of our applications depend on the `Scanner` class to read input

Aggregation

- Aggregation is a stronger form of dependency
- *Has-a* relationship
- Objects of one class contain references to objects of another class
- Use an instance variable
- Class A aggregates class B if A contains an instance field of type B
- Arrow is drawn from the contained class to the class that contains it
  - Car aggregates Tire
  - Car ← Tire

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Symbol</th>
<th>Line Style</th>
<th>Arrow Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance (Chapter 13)</td>
<td>Solid</td>
<td>Solid</td>
<td>Triangle</td>
</tr>
<tr>
<td>Interface</td>
<td>Dotted</td>
<td>Dotted</td>
<td>Triangle</td>
</tr>
<tr>
<td>Implementation</td>
<td>Solid</td>
<td>Solid</td>
<td>Diamond</td>
</tr>
<tr>
<td>Aggregation</td>
<td>Dotted</td>
<td>Dotted</td>
<td>Open</td>
</tr>
</tbody>
</table>

17.1 Software Life Cycle

- Software Life Cycle: all activities from initial analysis until obsolescence
  - Formal process for software development
  - Describes phases of the development process
  - Gives guidelines for how to carry out the phases

5 phases

1. Analysis
2. Design
3. Implementation
4. Testing
5. Deployment / Operation

5 phases

1. Analysis – What is the program supposed to do?
   - Output: requirements document
2. Design
3. Implementation
4. Testing
5. Deployment / Operation
5 phases

1. Analysis
2. Design – How is it going to be implemented?
   Output: Design document (UML/CRC cards/Javadocs)
3. Implementation
4. Testing
5. Deployment / Operation

Perfect World

- In a perfect world, everything would flow perfectly in this process
  - Output from one phase signifies it is complete and can start the next phase

- Doesn’t really work
  - You’ve probably noticed this
  - Was anyone’s A5 perfect?
  - Have your tests every worked completely?

Waterfall Model

- Figure 1: The Waterfall Model

5 phases

1. Analysis
2. Design
3. Implementation – Write the code (edit/compile/run)
   Output: Completed program
4. Testing
5. Deployment / Operation

5 phases

1. Analysis
2. Design
3. Implementation
4. Testing – Verify that it works correctly
   Output: Test cases passed (unit/system)
5. Deployment / Operation

5 phases

1. Analysis
2. Design
3. Implementation
4. Testing
5. Deployment / Operation – Maintain the program
   Output: Public product, patches, new features
Problems with Waterfall Model

- Specs usually have flaws
  - Contradictions
  - Non-thorough (what needs to happen on bad input?)
- Design too complicated, implementation flawed
- Testing incomplete

Spiral Model

- Breaks development process down into multiple phases
- Early phases focus on the construction of prototypes
  - Shows some aspects of the final product → quick implementation
  - Lessons learned from development of one prototype can be applied to the next iteration
- Problem: can lead to many iterations, and process can take too long to complete → high cost and low throughput

Spiral

Extreme Programming

- Approach suggested by Kent Back in 1999
- Goal: Simplicity
  - Cut out formal structure
  - Focus on set of practices to make programming more efficient and satisfactory

Practices

- **Realistic planning**: Customers make business decisions (what should it look like?), programmers make technical ones (how do we that?)
- **Small Releases** – start small, update later
- **Metaphor** – common story among programmers
- **Simplicity** – simple solution is best
- **Testing** – by everyone!
- **Refactoring** – restructure as you go

Cont.

- **Pair Programming**
- **Collective Ownership**
- **Continuous Organization**
- **40-hour week**
- **On-site customer**
- **Coding standards**