CH. 2
OBJECT-ORIENTED PROGRAMMING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
OBJECT-ORIENTED PROGRAMMING GOALS

• Robustness
  • We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.

• Adaptability
  • Software needs to be able to evolve over time in response to changing conditions in its environment.

• Reusability
  • The same code should be usable as a component of different systems in various applications.
OBJECT-ORIENTED PROGRAMMING
DESIGN PRINCIPLES

- **Object Oriented Programming** – paradigm for programming involving modularizing code into self contained **objects** that are a concise and consistent view of a “thing” without exposing unnecessary detail like the inner workings of the object
  
  - Abstraction – What makes up an object? The model
  - Composition – Objects can own other objects, "has-a" relationships
  - Encapsulation – Hiding implementation details, only exposing the "public interface"
  - Inheritance – Types and subtypes, "is-a" relationships
  - Polymorphism – Provision of a single interface to entities of different types
OBJECT-ORIENTED PROGRAMMING GUIDELINES

• Responsibilities
  • Divide the work into different actors, each with a different responsibility.

• Independence
  • Define the work for each class to be as independent from other classes as possible.

• Behaviors
  • Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.
OBJECT-ORIENTED PROGRAMMING
ABSTRACTION AND COMPOSITION

• Model entities as objects. Specific objects are called instances. Types of objects are often referred to as a class. Objects are composed of two parts
  • State describes the data composing the entity. Also called member data, instance data, fields, etc.
  • Behavior describes operations on or with the data. Also called methods.
  • Example: A type of entity President contains a name, term limits, etc. and can veto congressional bills as an operation, among other things

• Abstraction is deciding what data and operations compose an object type and what various instances an application has

• Composition simple states that the data can be another object, i.e., the "has-a" relationship
OBJECT-ORIENTED PROGRAMMING
ENCAPSULATION

• **Encapsulation** means (1) put the data with the methods that modify it, i.e., compose classes and (2) implementation hiding

• Essentially, you carefully design what anyone will see of an object, i.e., the operations to interact on the data, and what people won't see, e.g., the data or bookkeeping
  • Simple approach: keep the data private and methods public

• Frees the programmer to change the internal implementation as needed without affecting the user of an object
  • Supports adaptability
Inheritance describes "is-a" relationships, i.e., type (parent) and subtype (child) relationships

- The subtype inherits all properties of the parent type, but has the power to override, really specialize, the behaviors offered by the parent type
- Note: this is about types, not instances!
- Example: in 2D graphics, a sprite is an entity in a 2D world. Some sprites have an animation sequence, while others are static.

Polymorphism is the concept for utilizing inheritance. Algorithms are written in terms of the parent class and the actual behavior is specified based on the subtype

- Example: in 2D graphics, all sprites need to be drawn. The animated sprites (subtype) will not only draw an image, but then update their place in the animation.
WRITING OBJECTS

IN JAVA
TERMINOLOGY

• Object type, i.e., class – specifies instance variables, also known as data members, that the object contains, as well as the methods, also known as member functions, that the object can execute

• Object instance, i.e., object – variable of that object type
USING A CLASS

• Initialize a variable of an object with the keyword `new` followed by a call to the **constructor** of the object:

```java
String s = new String("Hello"); // Python omits the type
```

• Use a method of the class to do something useful:

```java
int l = s.length(); // Same as python
```
CLASS DEFINITIONS

• A class serves as the primary means for expressing an object type, i.e., the state and behavior of an object
  • State – Data members – defines the data composing an object and its size/layout in memory
    • Can be primitive types or other objects, i.e., composition – "has-a" relationship
  • Behavior – Member methods – set of behaviors that act upon the state of an object instance

<table>
<thead>
<tr>
<th>Class name</th>
<th>Data fields</th>
<th>Constructors and methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>radius: double</td>
<td>Circle()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Circle(newRadius: double)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getArea(): double</td>
</tr>
</tbody>
</table>
public class Circle {
    private double radius = 0;

    public Circle() {
    }

    public Circle(double r) {
        this.radius = r; // "this" can be omitted
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
1. public class ClassName {
2.   /* All instance variables declared private*/
3.   private int i = 0;
4.   /* Any public static final variables - these model constants */
5.   /* All constructors - constructors initialize all member data,
   and must be named the same as the class */
6.   public ClassName() {}
7.   /* All accessor (getters) and simple modifiers (setters) needed
   for the object */
8.   public int getI() {return i;}
9.   public int setI(int i) {this.i = i;
10.  /* All other public methods */
11.  /* Any and all private methods */
12.  /* Any and all static methods */
13.}
EXAMPLE

• Lets program (in pairs) a class for a bank account, Account.java
  • Have getters and setters for private member data (name and balance)
  • Have a deposit and withdrawal method to operate on an account
• Program a simple test to exercise all of the methods of Account
JAVA REFERENCES (POINTERS)

• Instances are stored as references, i.e., pointers to a memory location storing the state data

• When a reference variable does not reference any object, it holds a special literal value, **null**, which is typically memory location 0x0 or integer 0

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>myCircle</td>
</tr>
<tr>
<td>0xA (reference)</td>
</tr>
<tr>
<td>0xA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>radius</td>
<td>5</td>
</tr>
</tbody>
</table>
**ASSIGNMENT**

**PRIMITIVE DATA TYPES VS REFERENCE VARIABLE TYPES**

- Note how assignment of pointers does not copy the underlying memory!
- Same goes for passing objects into methods, its pass-by-value, i.e., pass-by-object-reference in this case
- If an object no longer has any variable referring to it, the *Java Garbage Collector* will reclaim the memory for the Java Virtual Machine

<table>
<thead>
<tr>
<th>Primitive type assignment</th>
<th>i = j</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before:</td>
<td>After:</td>
</tr>
<tr>
<td>i 1</td>
<td>i 2</td>
</tr>
<tr>
<td>j 2</td>
<td>j 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object type assignment c1 = c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before:</td>
</tr>
<tr>
<td>c1</td>
</tr>
<tr>
<td>c2: Circle radius = 5</td>
</tr>
<tr>
<td>c2</td>
</tr>
<tr>
<td>c1: Circle radius = 9</td>
</tr>
<tr>
<td>After:</td>
</tr>
<tr>
<td>c1</td>
</tr>
<tr>
<td>c2</td>
</tr>
<tr>
<td>c1: Circle radius = 5</td>
</tr>
<tr>
<td>C2: Circle radius = 9</td>
</tr>
</tbody>
</table>
ARRAYS OF OBJECTS

- Note: An array of objects is an array of pointers to objects
CONSTRUCTORS

• **Constructors** are a special kind of methods that are invoked to construct objects.

• This is where you describe how memory for an object is **initialized**

```java
Circle() {
}
Circle(double r) {
    radius = r;
}
```
CONSTRUCTORS

• A constructor with no parameters is referred to as a no-arg constructor.

• Constructors must have the same name as the class itself.

• Constructors do not have a return type, not even void.

• Constructors are invoked using the new operator when an object is created.

 Constructors play the role of initializing objects.

  • new ClassName();
  • Example: new Circle(2.3);
DEFAULT CONSTRUCTOR

• A class may be defined without constructors
• In this case, a no-arg constructor with an empty body is implicitly defined in the class.
• This constructor, called a **default constructor**, is provided automatically only if no constructors are explicitly defined in the class.
• Recall that you use the following to invoke a method in the Math class
  • Math.methodName(arguments) (e.g., Math.pow(3, 2.5))

• Can you invoke getArea() using Circle.getArea()?
  • The answer is no, you need to do this objectRefVar.methodName(arguments) (e.g., myCircle.getArea()).

• So what is the difference?
  • The difference is the modifiers of each member, more specifically the usage of static
INSTANCE VS STATIC

- **Instance** — a, or relating to a, specific object’s value. Does not use the `static` keyword.
  - Instance variables belong to a specific instance.
  - Instance methods are invoked by an instance of the class.

- **Static** — not a, or relating to a, specific object’s value (related to the type). Uses the `static` keyword
  - Static variables are shared by all the instances of the class
  - Static methods are not tied to a specific object
VISIBILITY MODIFIERS AND ACCESSOR/MUTATOR METHODS

• A **visibility modifier** defines the scope of a variable/method and enforces **encapsulation** (data hiding) in objects

• **public** – the class, data, or method is visible to any class in any package.

• **private** – the data or methods can be accessed only by the declaring class.

• By default (no modifier), the class, variable, or method can be accessed by any class in the same package (in between public and private)

• Typically, get and set methods are provided to read and modify private properties.
SCOPE OF VARIABLES

• Recall — **scope** is the lifetime of a variable. It dictates where you as the programmer may refer to the identifier (name) in code
  • Rule — The scope of instance and static variables is the entire class (including inside of any method). They can be declared anywhere inside a class.
  • Rule — The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be initialized explicitly before it can be used.
THE THIS KEYWORD

• The **this** keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class’s hidden data fields.

• Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.
IMMUTABLE OBJECTS AND CLASSES

• If the contents of an object cannot be changed once the object is created, the object is **immutable** and its class is called an **immutable class**.

• For a class to be immutable, it must mark all data fields private and provide no mutator methods and no accessor methods that would return a reference to a mutable data field object.
IMMUTABLE WRAPPER CLASSES

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double
- String

- All are immutable without a no-arg constructor
- All provide limits of their data types (e.g., `Integer.MAX_VALUE` and `Double.POSITIVE_INFINITY`)
- All provide functions to convert between each other (e.g., `Integer.parseInt()` and `String.valueOf()`)
- Since Java 5, primitive types can be automatically be converted to their immutable class counterpart (called **boxing**)

- `All` refer to the following data types: Boolean, Character, Short, Byte, Integer, Long, Float, Double, String.
NESTED CLASSES

• Java allows a class definition to be nested inside the definition of another class.

• The main use is in defining a class that is strongly affiliated with another class to increase encapsulation.

• Nested classes are a valuable technique when implementing data structures. A instance of the nested class could represent:
  • A small portion of the larger data structure
  • An auxiliary class to help navigation of the data structure.

```java
public class A {
    // Can be public or private
    // Can be static or non-static
    public class B {
    }
    // We will use this form
    // most often
    private static class C {
    }
}
```