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 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
  - Composition/Abstraction – What makes up an object? The model
  - Encapsulation – Hiding implementation details, only exposing the “public interface”
  - Inheritance – Types and subtypes, it’s a modeling decision
  - Polymorphism – Provision of a single interface to entities of different types
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 SOFTWARE DESIGN

- 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.
CRASH COURSE IN USING AND MAKING OBJECTS

REVIEW OF CMSC 150
TERMINOLOGY

• “object” is a little ambiguous
  • Object type or 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., variable of that object type
CLASS DEFINITIONS

• A class serves as the primary means for abstraction in object-oriented programming.

• In Java, every variable is either a primitive type or is a reference (pointer) to an instance of some class.

• A class provides a set of behaviors in the form of member functions (also known as methods), with implementations that belong to all its instances.

• A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of attributes (also known as fields, instance variables, or data members).
USING A CLASS (QUICK AND DIRTY REFRESHER)

• Initialize a variable of an object with the keyword `new` followed by a call to a constructor of the object:
  
  ```java
  String s = new String("Hello");
  ```

• Use a method of the class to execute a computation:
  
  ```java
  int l = s.length();
  ```
CLASS TEMPLATE (QUICK AND DIRTY REFRESHER)

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

- Let's program (in pairs) a class for a circle, Circle.java
  - Have getter and setter for private member data
  - Have an area computation
  - Have a computation to determine if two circles overlap

- Program a simple test to exercise all of the methods of circle
ABSTRACT DATA TYPES

• **Abstraction** is to distill a system to its most fundamental parts.

• Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).

• An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
  - This would essentially be the “public interface” of a class

**An ADT specifies what each operation does, but not how it does it**

• Lets repeat, an ADT is the operations not the implementation!
• We will see that we can implement ADTs in many, many ways
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, as an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary 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 {
        
    }
}
```
INHERITANCE
MOTIVATIONS

• Suppose you will want to model objects for shapes. Many of the objects will have common features, maybe colors, or the ability to compute their areas, or computing overlap between them. BUT, is there a way to reduce the amount of repeated code? Improve the robustness (correctness) of the model? Design this type of model hierarchy?

• How about an example of allied characters in a game? Some help you by healing, some help offensively, some help defensively. However, all of these types of allies have commonality. So the same questions exist!

• The answer is to use inheritance – modeling types and subtypes in a way that reduces duplicated components.
**INHERITANCE**

- **Inheritance** is a type/sub-type relationship (parent/child) denoted with an arrow pointed to the type in a UML diagram
  - A **superclass (base class)** is the inherited object type
  - A **subclass (derived class)** is the inheriting object type
  - All of the state (data fields) and behavior (methods) of the superclass is inherited (“handed-down”) to the subclass
    - The superclass constructors are NOT inherited

---

**GeometricObject**
- `-color: String`
- `-filled: boolean`
- `-dateCreated: java.util.Date`

- `+GeometricObject()` creates a GeometricObject.
- `+GeometricObject(color: String, filled: boolean)` creates a GeometricObject with the specified color and filled values.

- `+getColor(): String` returns the color.
- `+setColor(color: String): void` sets a new color.
- `+isFilled(): boolean` returns the filled property.
- `+setFilled(filled: boolean): void` sets a new filled property.
- `+getDateCreated(): java.util.Date` returns the dateCreated.
- `+toString(): String` returns a string representation of this object.

---

**Circle**
- `-radius: double`

- `+Circle()` creates a Circle.
- `+Circle(radius: double)` creates a Circle with the specified radius.
- `+Circle(radius: double, color: String, filled: boolean)` creates a Circle with the specified radius, color, and filled property.

- `+getRadius(): double` returns the radius.
- `+getArea(): double` returns the area of the Circle.
- `+getPerimeter(): double` returns the perimeter of the Circle.
- `+getDiameter(): double` returns the diameter of the Circle.
- `+printCircle(): void` prints the Circle.

---

**Rectangle**
- `-width: double`
- `-height: double`

- `+Rectangle()` creates a Rectangle.
- `+Rectangle(width: double, height: double)` creates a Rectangle with the specified width and height.
- `+Rectangle(width: double, height: double, color: String, filled: boolean)` creates a Rectangle with the specified width, height, color, and filled property.

- `+getWidth(): double` returns the width.
- `+setWidth(width: double): void` sets a new width.
- `+getHeight(): double` returns the height.
- `+setHeight(height: double): void` sets a new height.
- `+getArea(): double` returns the area of the Rectangle.
- `+getPerimeter(): double` returns the perimeter of the Rectangle.
INHERITANCE IN JAVA

1. public class A {
2.     private int a;
3. }
4. public class B extends A {
5.     private int b;
6. }

• In Java, the keyword **extends** denotes an inheritance relationship
• In this example, by inheritance **B** is an object whose state is defined by two **ints**, the one in **A** and the one in **B**
• In this relationship, the superclass is responsible for constructing (initializing) the superclass’s data fields, while the subtype is responsible for the subclass’s data fields
CONSTRUCTION IN INHERITANCE

• The superclass constructor is not inherited, so how do we construct it’s part of memory?
• They are invoked explicitly (by the programmer) or implicitly (by the Java compiler)
• We use the `super` keyword to invoke explicitly
• The Java compiler will always attempt to invoke the no-arg constructor implicitly
• Caveats:
  • We must use the keyword `super`, otherwise error
  • It must be the very first line of the constructor, otherwise error

Explicitely:
```java
public B() {
    super();  //note this is like any constructor, we are free to pass parameters as well!
}
```

Implicitely:
```java
public B() {
    //java inserts super() - always calling the no-arg constructor
}
```
SUPER

• A reference to the superclass
  • Synonymous to this

• Can be used to
  • Call superclass constructor
  • Call methods/data fields of superclass

1. public class A {
2.   int x;
3.   public A(int a) {x = a;}
4.   public void printA() {System.out.print(x);}
5. }
6. public class B extends A {
7.   int y;
8.   public B(int a, int b) {
9.     super(a); //Example of construction
10.    y = b;
11. }
12. public void printB() {
13.    super.printA(); //Example of method invocation
14.    System.out.print(", "+ y);
15. }
16. }
DEFINING A SUBCLASS

• A subclass inherits from a superclass. You can also:
  • Add new properties
  • Add new methods
  • Override the methods of the superclass

• Conceptually a subclass represents a smaller set of things, so we make our subclass more detailed to model this
OVERRIDING

• A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

• Note this is different than method overloading — two functions named identically with different signatures.
1. public class Shape {
2.     private Color c;
3.     /* other parts omitted for brevity */
4.     public void draw() {
5.         StdDraw.setPenColor(c);
6.     }
7. }

1. public class Circle extends Shape {
2.     private double x, y;
3.     private double radius;
4.     /* other parts omitted for brevity */
5.     public void draw() {
6.         super.draw();
7.         StdDraw.filledCircle(x, y, radius);
8.     }
9. }

Circle overrides the implementation of draw
THE JAVA OBJECT CLASS

• Every class in Java is descended from the `java.lang.Object` class. If no
ingheritance is specified when a class is defined, the superclass of the class is
Object.

• Java Object provides for a few basic functions, like `toString()`.

• We will use others as we go.
POLYMORPHISM
POLYMORPHISM

- **Polymorphism** means that a variable of a superclass (supertype) can refer to a subclass (subtype) object

```
Shape s = new Circle(5);
```

- Under the context of polymorphism, the supertype here is the **declared type** and the subtype is the **actual type**

- Polymorphism implies that an object of a subtype can be used wherever its supertype value is required
WHY WOULD YOU EVER DO THIS?

- Allow types to be defined at runtime, instead of at compile time:

```java
Scanner s = new Scanner(System.in);
Shape shape = null;
String tag = s.next();
if (tag.equals("Circle")) { //user wants a circle
    double r = s.nextDouble();
    shape = new Circle(r, Color.red);
} else if (tag.equals("Rectangle")) { //User wants a rectangle
    double w = s.nextDouble(), h = s.nextDouble();
    shape = new Rectangle(w, h, Color.red);
}
System.out.println("Area: " + shape.area()); //works no matter what!
```
WHY WOULD YOU EVER DO THIS?

• Arrays can only store one type

1. `Circle[]` circles;  //all circles
2. `Rectangle[]` rects;  //all rectangles
3. `Shape[]` shapes;  //depends on subtypes! Can have some circles and some rectangles.
WHY WOULD YOU EVER DO THIS?

• Lets say we have an array of Shape shapes then we can do something like:

1. `double total = 0;`
2. `for (int i = 0; i < shapes.length; ++i)`
3. `total += shapes[i].area();` //Uses specific
4. `return total;` //instance’s function
• Method m takes a parameter of the Object type. You can invoke it with any object.

• When the method m(Object x) is executed, the argument x’s toString method is invoked. Classes Student, Person, and Object have their own implementation of the toString method.

• The correct implementation is dynamically determined by the Java Virtual Machine. This is called dynamic binding.

• Polymorphism allows superclass methods to be used generically for a wide range of object arguments (any possible subclass). This is known as generic programming.
POLYMORPHISM AND TYPE CONVERSION

• So when assigning a value of a subtype to a variable of a supertype, the conversion is implicit:

  Shape s = new Circle(5);  //implicit conversion from Circle to Shape
  This is called **upcasting**.

• When going from a supertype value to a subtype variable, the conversion must be explicit:

  Circle c = (Circle)s;  //explicit conversion from Shape to circle
  This is called **downcasting**. This type of casting might not always succeed, why?
THE INSTANCEOF OPERATOR

- Use the `instanceof` operator to test whether an object is an instance of a class:

```java
1. Object myObject = new Circle();
2. /** Perform downcasting only if myObject is an instance of Circle */
3. if (myObject instanceof Circle) {
4.     System.out.println("The circle diameter is " +
5.         ((Circle)myObject).getDiameter());
6. }```

```
The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```java
public boolean equals(Object obj) {
    return this == obj;
}
```

What is the problem? How do we fix it?

• `==` for objects compares their memory addresses, not their values.

As an example of overriding the method for our Circle:

```java
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    } else {
        return false;
    }
}
```
THE FINAL MODIFIER

• The final modifier, introduced with variables to define constants, e.g., PI, has extended meaning in the context of inheritance:

• A final class cannot be extended:
  ```java
  final class Math {
    ...
  }
  ```

• The final method cannot be overridden by its subclasses:
  ```java
  public final double getArea() {
    return Math.PI*radius*radius;
  }
  ```
INTERFACES AND ABSTRACT CLASSES

• The main structural element in Java that enforces an application programming interface (API) is an interface.

• An interface is a collection of method declarations with no data and no bodies.

• Interfaces do not have constructors and they cannot be directly instantiated.
  • When a class implements an interface, it must implement all of the methods declared in the interface.

• An abstract class also cannot be instantiated, but it can define one or more common methods that all implementations of the abstraction will have.
INTERFACE EXAMPLE

1. public interface Robot {
2.   void sense(World w);
3.   void plan();
4.   void act(World w);
5. }
USE IMPLEMENTS TO ENFORCE THE INTERFACE

1. public class Roomba implements Robot {
2.   /* code specific to Roomba */
3.   public void sense(World w) { /* Roomba’s don’t sense */}
4.   public void plan() { /* code for roombas actions */}
5.   public void act(World w) { /* code to power motors */}
6.   /* code specific to Roomba */
7. }

GENERIC PROGRAMMING

JAVA GENERICS
Generic Programming is a programming paradigm where the programmer programs interfaces and algorithms without a specific object hierarchy in mind. Rather the programmer only worries about operations (methods) needed to support an interface or perform an algorithm
HOW DO WE PROGRAM GENERICALLY?

1. Inheritance/polymorphism – Treat everything as an Object, or use very deep class hierarchies in combination with polymorphism

2. Generics/templates – A programming technique where we program without any specific type. Then when we instantiate a class the type becomes known

• (1) is done with polymorphism and (2) is done with Java Generics
• In all honesty though, we use both in combination.
SYNTAX FOR GENERIC OBJECTS

• Types can be declared using generic names:

    1. public class Array<E> {
    2.     private E e[];
    3.     /* Rest of class */
    4. }

• They are then instantiated using actual types:
  • Array<String> arr = new Array<>();

• There is not much to it actually, but it is a very strange thought process that you do not know what E is as you write it.
GENERIC OBJECTS

• You may have one or more generic types. This class, we will have at most two
  • public class Map<Key, Value> {
    }
  • Map<Integer, String>

• **Generic types must be Java Objects**, i.e., any class that inherits from Java Object, i.e., any class meaning no `int` or `float`.

• Many other quirks and oddities that will be experienced as we go!

• All code examples in the book and programming assignments involve this form of programming actually.
**GENERIC FUNCTIONS**

- Can also be used in functions:
  1. `public static <T, S> String concat(T t, S s) {`
  2.   //make assumptions on the types.
  3.   //Any type that satisfies operation constraints may be used!
  4.   return t.toString() + s.toString();
  5. }

- Used like:
  1. `MyObject1 a;`
  2. `MyObject2 b;`
  3. `String c = concat(a, b);`

Types are implicitly determined by compiler
• We can specify constraints on the generic parameters as well through interfaces

```java
public <T extends Comparable<T>> int compareTwo(T a, T b) {
    return a.compareTo(b); // Comparable<T> enforces
    // having the compareTo function
}
```

• `extends` is always used, it generally refers to `extends` or `implements` in this context
EXERCISE

• With a team, program a generic point class whose coordinates can be integers, complex numbers, floats, doubles, etc.

• At a minimum support two-dimensions, ability to scale points, and ability to add points together
ADVANCED PROGRAMMING TECHNIQUES
**EXCEPTIONS**

- **Exceptions** are unexpected events that occur during the execution of a program.
- An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer.
- In Java, exceptions are objects that can be **thrown** by code that encounters an unexpected situation.
- An exception may also be **caught** by a surrounding block of code that “handles” the problem.
- If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console.
CATCHING EXCEPTIONS

• The general methodology for handling exceptions is a **try-catch** construct in which a guarded fragment of code that might throw an exception is executed.

• If it **throws** an exception, then that exception is caught by having the flow of control jump to a predefined **catch block** that contains the code to apply an appropriate resolution.

• If no exception occurs in the guarded code, all catch blocks are ignored.

```java
1. ...
2. try {
3.   /*Code that may generate exception*/
4. }
5. catch(ExceptionType1 e1) {
6. }
7. catch(ExceptionType2 e2) {
8. }
9. ...
```
THROWING EXCEPTIONS

• Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object.

• This is done by using the `throw` keyword followed by an instance of the exception type to be thrown:

  ```java
  throw new ExceptionType(parameters);
  ```
THE THROWS CLAUSE

• When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method.

• The syntax for declaring possible exceptions in a method signature relies on the keyword `throws` (not to be confused with an actual throw statement).

```java
public static int parseInt(String s) throws NumberFormatException;
```
OTHER USEFUL TIDBITS
The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
## ACCESSIBILITY SUMMARY

<table>
<thead>
<tr>
<th>Modifier on members in a class</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>default</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
VISIBILITY MODIFIERS FULL EXAMPLE

package p1;
public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;
  
  protected void m() {
  }
}

package p2;
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;
  can invoke o.m();
}

package p3;
public class C3 extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;
  
  can invoke m();
}

public class C4 extends C1 {
  C1 o = new C1();
  can access x;
  can access y;
  cannot access z;
  cannot access u;
  
  cannot access o.u;
  can invoke o.m();
}

public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;
  
  cannot invoke o.m();
}
A SUBCLASS CANNOT WEAKEN THE ACCESSIBILITY

• A subclass may override a protected method in its superclass and change its visibility to public.

• However, a subclass cannot "weaken" the accessibility of a method defined in the superclass.
  
  • For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.