JAVA PROGRAMMING: CLASS 5

Amount of work and time required  (30-60 mins/day practice-study)
Timeline
Rewrite ( static imports, quiz, assignment 1 )
New style requirements for assignment 4+.
Classes, Constructors, SuperClasses, Inheritance
Arrays and Garbage Collection
Abstract Classes, Interfaces
Inheritance Pitfalls
Assignment 4 (due in 2 weeks)
TIMELINE: MIDTERM

• New Midterm date: Feb 26th.
• Will cover chapters 1-5 in the book.
• Problems will be similar in difficulty to what’s in the book.
## TIMELINE: ASSIGNMENTS/PROJECT

<table>
<thead>
<tr>
<th>Assignments/Project</th>
<th>Due Date</th>
<th>Covers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment 4</td>
<td>March 5</td>
<td>Chapters 4-5</td>
</tr>
<tr>
<td>Assignment 5</td>
<td>March 12</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>Assignment 6</td>
<td>March 26</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Semester Project</td>
<td>April 23</td>
<td>Everything</td>
</tr>
</tbody>
</table>

Change Summary:
- We’ll cover chapter 7 before chapter 6
- Two additional assignments, but smaller scope
Static imports can be used for importing static methods and static variables.

Statically imported methods and variables don’t require qualification with the name of the class.

Useful especially when there are numerous methods and you want to keep the names short.
REWRITE: QUIZ

• Numerous misinterpretations and misreads of the question/problem. Please read the questions carefully and ask for clarification if you need it.

• Problem 1:
  • Missed “highest to lowest order”.
  • Incorrect syntax in for loop, or for assignments.

• Problem 2:
  • Numerous misreads: lots of 5,4,3,2,1 which is x not y.
  • Missed second ‘120’, because x starts at 5 not 4.

• Problem 3:
  • non-Java syntax! Pointer arithmetic not valid in Java.
  • returning the lowest, not the highest index where array value = 8.
  • repeated printing of “value 8 not in the array” or “value 8 at index...” or BOTH
REWRITE: ASSIGNMENT 1

• Types of errors: various degrees of problem misinterpretation and problem incorrectness.

• Solution Review
ASSIGNMENT 4+ SOLUTION REQUIREMENTS

• Code must be correct and comply with the problem statement.
• If there are any ambiguities, ask for clarification. If that isn’t available in time, document your interpretation.
• Use short (single responsibility) methods with meaningful names.
• Use short (single responsibility) classes with meaningful names.
• Use meaningful variable names where it will increase clarity.
• Use formatting standards (files will be posted to website and can be imported into Eclipse).
CLASSES AND OBJECTS (INSTANCES)

• Classes are similar to factories from which objects (instances) are created.
• Object and instance are interchangeable terms.
• Static member variables and methods of classes are distinct from instance variables and methods.
• Instance variables and methods are allocated per instance.
• Contrast to: Static variables and methods that are allocated per class.
• An instance method can refer to a static variable or method, but a static method cannot refer to an instance variable or method without having a reference to the instance.
• See book’s diagram on pages 190 and 191.
CONSTRUCTORS

• A constructor looks much like a method, but it does not return a value.
• If you do not supply a constructor, the compiler makes a default empty one for you.
• Constructors may throw checked exceptions, but it is better to avoid it.
• The constructor’s job is to initialize an object’s instance variables into a consistent state.
INHERITANCE HIERARCHY

• All classes in Java inherit from a single class called Object.
• Check Object’s API: you’ll notice several methods:
  • toString() // returns a default string representing an object: (class name, ‘@’ and the hex representation of the hash code of the object,
  • wait() and some variations
  • notify() and a variation
  • equals() // Object’s implementation is ‘==’ or reference equality, probably not what you want.
  • You should override this method whenever you plan to pass an object to a method that depends on it, or inserting it into an array. Arrays.binarySearch() depends on this to properly find elements in an array.
  • hashCode() // Default is based on the address of the object in memory
  • hashCode() defines a “general contract” to follow: if two objects compare true for equals() they must generate the same hash code. hashCode() must be idempotent, meaning it returns the same on each call.
  • getClass() returns a Class instance that represents the object’s class. (Yes, this is confusing.)
JAVA INHERITANCE HIERARCHY

- Can be extended through an arbitrary number of levels.
- Supports only single class inheritance.
INHERITANCE

• You may define a subclass of another class using the ‘extends’ keyword.
• Terminology: Parent-Child, Superclass-Subclass, or Base-Derived classes are equivalent.
• All public, protected, and package-private member variables and methods are inherited.
• Inherited methods can legally overload a method in the child class,
• Inherited methods can be overridden – i.e. shadowed by methods in the child class provided. Somewhat complex rules apply on whether a method has been legally overridden. We’ll discuss that more in class 6.
• If you don’t want a subclass to override a superclass method, declare the method ‘final’ in the superclass.
GARBAGE COLLECTION

- Objects that are no longer referenced by any program variable are eligible for automatic garbage collection.
- There is no way to predict nor force garbage collection to occur.
- You can “ask” for it with System.gc(), but it’s still not guaranteed to run.
- If you inadvertently hold a reference someplace to an object that is no longer used, you can create a memory leak.
THIS VARIABLE AND THIS METHOD()

• ‘this’ is a reference to the current instance and can be used to pass that reference outside the class. That’s called “‘this’ reference escape“. It is something we’ll consider further when we discuss multithreaded programs.

• Why use ‘this’ in a class? It clarifies and simplifies setting instance variables from formal parameters in a constructor. Otherwise, you need two different names. (Why?)

• Why use this.someMethod(); vs someMethod()? No compelling reason! Before IDEs existed it clarified that an instance variable was referenced. However, IDEs use color highlighting to indicate this now.
THIS(...)  

- Invokes a different constructor in the same instance, from within a constructor.
- This is useful when you want to avoid repeating code in two different yet similar constructors.
SUPER.VARIABLE AND SUPER.SUPERCLASS METHOD()

• ‘super’ refers to the superclass and provides access to its instance variables and methods.

• Can use it in any instance method or constructor to access variables/methods that might be hidden by subclass variables. (private variables/methods cannot be accessed).
SUPER(...)  

- Provides a means to call a superclass constructor that has been overridden by a subclass method.
- If present, super() must be the first line of a subclass’s constructor.
- If you do NOT supply a constructor for a subclass, then the compiler provides a default empty constructor... except it’s not exactly empty. It calls super() with no parameters.
- If you do NOT supply a constructor for the parent class, then the compiler provides a default empty constructor for it. The child class’ default constructor calls it.
- If you create a superclass constructor that takes any parameters, the constructor will NOT supply a default constructor in the superclass.
- What happens in the default constructor of the subclass? Hint: It’s not very super...
HANDS ON: WRITE AND USE A CLASS

• Create two classes with one class a subclass of the other.
• Add several instance variables to each, with at least one overlapping.
• In the constructors, set the variables to different values in each class.
• Create a main in the subclass and instantiate the subclass.
• Add a superclass method to print the variables of the superclass.
• Call a method that uses super() to invoke a superclass method to print
ABSTRACT CLASSES AND METHODS

- Abstract methods are only declared. There is no implementation. If a class contains any abstract method(s), it is considered an abstract class and must be declared abstract. Example:

```java
public abstract class AbstractClassExample {
    public abstract int abstractMethod(); // no implementation
    public void test() {
        System.out.println("this is a test");
    }
}
```
ABSTRACT CLASSES

- May contain method implementations to be inherited by subclasses, whereas abstract methods are intended to be implemented only by subclasses.

- An abstract method is just a method declaration. Subclasses are required to implement an abstract method or else be abstract themselves (notice the parallel with how methods must catch an exception or declare its type in the throws clause).

- Subclasses that extend an abstract class inherit method interfaces and possibly implementations. Interface inheritance from abstract classes is considered better practice, as there are subtle problems that can arise from inheriting implementations.
INTERFACES

• Can contain public abstract methods
• Or... constants (e.g. public int A_CONSTANT = 1; (but, don’t do this, use enums instead)
• Or... default method implementations (mixins). This is a new Java 8 feature.
• Classes can implement interfaces. One interface can extend another interface.
• Any class SomeClass or interface SomeInterface that implements an interface BaseInterface becomes multityped. SomeClass instances are then considered a instances of type SomeClass, BaseInterface, and SomeInterface.
INTERFACE VS ABSTRACT CLASS USAGE

• Use interfaces when there is no common behavior to factor out that manages object state.

• Use interfaces with default methods when there is common behavior to provide to implementing classes that does not require access to object state.

• Use abstract classes when there is common implementation between classes that uses object state.

• This requires significant examples to grasp well.
INTERFACE VS ABSTRACT CLASS USAGE

• Java supports only single inheritance. Once a class extends an abstract class, it can’t extend another.

• Interfaces enable a class to also become the type of multiple implemented interfaces. This non-hierachical organization may make more conceptual sense than shoehorning a class into an inheritance hierarchy.

• Abstract classes are a better choice than interfaces when you expect to add future common functionality. With an interface, you’d have to add the method(s) into every implementing class.

• Summary: Prefer interfaces unless you have a good reason to use an abstract class.
RUNTIME POLYMORPHISM

• Polymorphism means “many forms”.
• In Java, polymorphism is seen in both method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).
• Runtime polymorphism is a consequence of method overriding in subclasses and the use of an object reference to refer to the superclass method while referring to a subclass object.
• Examples: See Extender 1 and 2. References declared as a superclass type can refer to any subclass instance. When this happens only the interface methods from the superclass can be invoke at compile time. But at runtime, the method implementation in the subclass is invoked.
• This is a simple idea with significant implications for software extensibility and good design.
INHERITANCE PITFALLS (FRAGILE CLASSES)

• It is dangerous to inherit from classes across packages because you may not have control over the source code you are extending.
  • If inherited methods from a superclass are used, your subclass becomes implicitly dependent upon inherited implementation details: i.e. the variables and state of the superclass.
  • If the superclass implementation changes, your code can break: it might fail to compile or it might produce incorrect output.
  • How? Suppose you override two superclass methods where one calls the other in the superclass: super.method1() and method2(). If your overriding method invokes the superclass method directly (using super.method1()) and super.method1() calls super.method2(), then super.method1() will invoke subclass.method2() instead. Unless you know this, method2() may corrupt your object’s state. We’ll review examples of this in detail in the next class.
  • General advice: Avoid inheriting independent concrete methods from a superclass unless you understand its implementation clearly. This is a good reason to prefer interfaces where applicable.
ASSIGNMENT 4 (DUE: MARCH 5, 2015)

• Reorganize your solution to Assignment 3 to organize it into classes as follows:
  • An observable ‘Model’ class that contains the data array and takes input from the Controller, then notifies all observers if there are any changes.
  • An observer ‘View’ class registered with the Model that receives change notifications with details of just the squares of the grid that have changed. It must immediately update the grid when notified.
  • A ‘Controller’ class that takes input from the user and updates the Model.
    • Valid inputs are: Changes per assignment 3.
    • The controller should handle the random refresh timing and update the Model in that case, too.
  • Create a ‘ChangeEvent’ class to send any Model change(s) to the View.
  • Create an Observable interface that the model implements, with one method: register(Observer someObserver). The View class should implement the Observer interface which should contain one method: eventNotify(ChangeEvent c).
  • Create and register a second observer (grid) that immediately displays the inverse colors in each square relative to the other observer. An inverse color is computed from an existing RGB vector with this formula: <redValue, greenValue, blueValue> goes to < 255-redValue, 255-greenValue, 255-blueValue >.