Object Oriented Programming

1. Review
2. Exam format

What is a class?

- A class is *primarily* a description of objects, or instances, of that class
  - A class contains one or more constructors to create objects
  - A class is a *type*
    - A type defines a set of possible values, and operations on those values
    - The type of an object is the class that created it

What is a class?

- But a class can also contain information about itself
  - Anything declared static belongs to the class itself
  - Static variables contain information about the class, not about instances of the class
  - Static methods are executed by the class, not by instances of the class
  - Anything *not* declared static is *not* part of the class, and cannot be used directly by the class
    - However, a static method *can* create (or be given) objects, and can send messages to them

Classes

- **class** MyClass extends ThatClass implements SomeInterface, SomeOtherInterface {...}
  - A top-level class can be public or package (default)
  - A class can be final, meaning it cannot be subclassed
  - A class subclasses exactly one other class (default: Object)
  - A class can implement any number of interfaces

Classes

- **abstract class** MyClass extends ThatClass implements SomeInterface, SomeOtherInterface {...}
  - Same rules as before, except: An abstract class *cannot* be final
  - A class *must* be declared abstract if:
    - It contains abstract methods
    - It implements an interface but does not define all the methods of that interface
  - Any class *may* be declared to be abstract
  - An abstract class can (and does) have constructors
  - You *cannot instantiate* an abstract class

Why inheritance?

- Java provides a huge library of pre-written classes
  - Sometimes these classes are exactly what you need
  - Sometimes these classes are *almost* what you need
  - It's easy to subclass a class and override the methods that you want to behave differently
- Inheritance is a way of providing similar behavior to different kinds of objects, without duplicating code

Why inheritance?

- You should extend a class (and inherit from it) *only* if:
  - Your new class *really is* a more specific kind of the superclass, **and**
  - You want your new class to have *most or all* of the functionality of the class you are extending, **and**
  - You need to add to or modify the capabilities of the superclass
- You *should not* extend a class merely to use *some* of its features
  - Composition is a better solution in this case

What are abstract classes for?

- Abstract classes are suitable when you can reasonably implement some, but not all, of the behavior of the subclasses
- Example: You have a board game in which various kinds of animals move around
  - All animals can move(), eat(), drink(), hide(), etc.
  - Since these are identical or similar, it makes sense to have a default move() method, a default drink() method, etc.

What are abstract classes for?

- Example (cont'd)
  - If you have a default draw() method, what would it draw?
  - Since you probably never want an Animal object, but just specific animals (Dog, Cat, Mouse, etc.), you don't need to be able to instantiate the Animal class
  - Make Animal abstract, with an abstract void draw() method

Interfaces

- interface MyInterface extends SomeOtherInterface {...}
  - An interface can be public or package
  - An interface cannot be final
  - A class can implement any number of interfaces
  - An interface can *declare* (not *define*) methods
    - All declared methods are implicitly public and abstract
  - An interface can define fields, classes, and interfaces
    - Fields are implicitly static, final, and public
    - Classes are implicitly static and public
    - An interface *cannot* declare constructors
  - It's OK (but unnecessary) to explicitly specify implicit attributes

Declarations and assignments

- Suppose class Cat extends Animal implements Pet {...}and class Persian extends Cat {...} and Cat puff = new Cat();
- Then the following are true:
  - puff instanceof Cat, puff instanceof Animal, puff instanceof Pet
- The following is *not* true: puff instanceof Persian
  - To form the negative test, say !(puff instanceof Persian)
- The following declarations and assignments are legal:
  - Animal thatAnimal = puff;
  - Animal thatAnimal = (Animal)puff; // same as above, but explicit upcast
  - Pet myPet = puff; // a variable can be of an interface type
  - Persian myFancyCat = (Persian)puff; // does a runtime check
- The following is also legal:
  - void feed(Pet p, Food f) {...} // interface type as a parameter

What are interfaces for?

- Inheritance lets you guarantee that subclass objects have the same methods as their superclass objects
- Interfaces let you guarantee that unrelated objects have the same methods
  - Problem: Your GUI has an area in which it needs to *draw* some object, but you don't know yet what kind of object it will be

What are interfaces for?

- Solution:
  - Define a Drawable interface, with a method draw()
  - Make your tables, graphs, line drawings, etc., implement Drawable
  - In your GUI, call the object's draw() method (legal for any Drawable object)
- If you didn't have interfaces, here's what you would have to do:
  - `if (obj instanceof Table) ((Table)obj).draw();`
    `else if (obj instanceof Graph) ((Graph)obj).draw();`
    `else if (obj instanceof LineDrawing) ((LineDrawing)obj).draw();`
    `// etc.`
  - Worse, to add a new type of object, you have to change a lot of code

Inner Classes

- Inner classes are classes declared within another class
- A member class is defined immediately within another class
  - A member class may be static
  - A member class may be abstract or final (but not both)
  - A member class may be public, protected, package, or private

Inner Classes

- A local class is declared in a constructor, method, or initializer block
  - A local class may be abstract or final (but not both)
  - A local class may access only final variables in its enclosing code
  - An anonymous class is a special kind of local class

Inner Classes

- An anonymous inner class is a kind of local class
  - An anonymous inner class has one of the following forms:
    - new ***NameOfSuperclass*(*parameters*) { *class body* }**
    - new ***NameOfInterface*() { *class body* }**
  - Anonymous inner classes cannot have explicit constructors
- A static member class is written inside another class, but is not actually an inner class
  - A static member class has no special access to names in its containing class
  - To refer to the static inner class from a class outside the containing class, use the syntax ***OuterClassName*.*InnerClassName***
  - A static member class may contain static fields and methods

What are inner classes for?

- Sometimes a class is needed by only one other class
  - Example: A class to handle an event, such as a button click, is probably needed only in the GUI class
  - Having such a class available at the top level, where it isn't needed, just adds clutter
  - It's best to "hide" such classes from other classes that don't care about it

What are inner classes for?

- Sometimes a class needs access to many variables and methods of another class
  - Again, an event handler is a good example
  - Making it an inner class gives it full access
- Sometimes a class is only needed once, for one object, in one specific place
  - Most event handlers are like this
  - An anonymous inner class is very handy for this purpose

Enumerations

- An enumeration, or "enum," is simply a set of constants to represent various values
- Here's the old way of doing it
  - public final int SPRING = 0;
    public final int SUMMER = 1;
    public final int FALL = 2;
    public final int WINTER = 3;
- This is a nuisance, and is error prone as well
- Here's the new way of doing it:
  - enum Season { WINTER, SPRING, SUMMER, FALL }

enums are classes

- An enum is actually a new type of class
  - You can declare them as inner classes or outer classes
  - You can declare variables of an enum type and get type safety and compile time checking
    - Each declared value is an instance of the enum class
    - Enums are implicitly public, static, and final
    - You can compare enums with either equals or ==

enums are classes

- enums extend java.lang.Enum and implement java.lang.Comparable
  - Hence, enums can be sorted
- Enums override toString() and provide valueOf()
- Example:
  - Season season = Season.WINTER;
  - System.out.println(season); // prints WINTER
  - season = Season.valueOf("SPRING"); // sets season to Season.SPRING

Enums *really are* classes

```
public enum Coin {
    // enums can have instance variables
    private final int value;
    // An enum can have a constructor, but it isn't public
    Coin(int value) { this.value = value; }
    // Each enum value you list really calls a constructor
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    // And, of course, classes can have methods
    public int value() { return value; }
}
```


Other features of enums

- values() returns an array of enum values
  - Season[] seasonValues = Season.values();
- switch statements can now work with enums
  - switch (thisSeason) { case SUMMER: ...; default: ...}
  - You *must* say case SUMMER:, *not* case Season.SUMMER:
  - It's still a very good idea to include a default case
- It is possible to define value-specific class bodies, so that each value has its own methods
  - The syntax for this is weird so we will not discuss it

Generic classes

```
public class Box<T> {
    private List<T> contents;
    public Box() {
        contents = new ArrayList<T>();
    }
    public void add(T thing) { contents.add(thing); }
    public T grab() {
        if (contents.size() > 0) return contents.remove(0);
        else return null;
    }
```

- Sun's recommendation is to use single capital letters (such as T) for types
- Many people, don't think much of this recommendation

Access

- There are four types of access:
  - public means accessible from everywhere
    - Making a field public means that it can be changed arbitrarily from anywhere, with no protection
    - Methods should be public only if it's desirable to be able to call them from outside this class
  - protected means accessible from all classes in this same directory *and* accessible from all subclasses anywhere

Access

- Package (default; no keyword) means accessible from all classes in this same directory
- private means accessible only within this class
  - Note: Making a field private does not hide it from other objects in this same class!
- In general, it's best to make all variables as private as possible, and to make methods public enough to be used where they are needed

Proper use of fields

- An object can have fields and methods
  - When an object is created,
    - It is created with all the non-static fields defined in its class
    - It can execute all the instance methods defined in its class
    - Inside an instance method, this refers to the object executing the method
  - The fields of the object should describe the *state* of the object
    - All fields should say something significant about the object
    - Variables that don't describe the object should be local variables, and can be passed from one method to another as parameters

Proper use of fields

- The fields of an object should be impervious to corruption from outside
  - This localizes errors in an object to bugs in its class
  - Hence, fields should be as private as possible
  - All public fields should be documented with Javadoc
  - Getters and setters can be used to check the validity of any changes
  - If a class is designed to be subclassed, fields that the subclass needs to access are typically marked protected

Composition and inheritance

- Composition is when an object of one class *uses* an object of another class
  - class MyClass {
    String s; ... }
  - MyClass has complete control over its methods
- Inheritance is when a class *extends* another class
  - class MyClass extends Superclass { ... }
  - MyClass gets all the static variables, instance variables, static methods, and instance methods of Superclass, whether it wants them or not
  - Constructors are *not* inherited
  - Inheritance should only be used when you can honestly say that a MyClass object ***is a*** Superclass object
    - Good: class Secretary extends Employee
    - Bad: class Secretary extends AccountingSystem

Constructors

- A constructor is the *only* way to make instances of a class
- Here's what a constructor does:
  - First, it calls the constructor for its superclass:
    - public MyClass() { **super();** ... } // implicit (invisible) call
      - Note that it calls the superclass constructor with *no* arguments
      - But you can explicitly call a different superclass constructor: public MyClass(int size) { **super(size);** ... } // explicit call
      - Or you can explicitly call a different constructor in this class: public MyClass() { **this(0);** ... } // explicit call

Constructors

- Next, it adds the instance fields declared in this class (and possibly initializes them)
  - class MyClass { **int x; double y = 3.5; …** } // in class, not constructor
- Next, it executes the code in the constructor:
  - public MyClass() { super(); **next = 0; doThis(); doThat(); …** }
- Finally, it returns the resultant object
  - You can say return; but you can't explicitly say what to return

Constructor chaining

- *Every* class *always* has a constructor
  - If you don't write a constructor, Java supplies a default constructor with no arguments
  - If you *do* write a constructor, Java does *not* supply a default constructor
- The first thing any constructor does (except the constructor for Object) is call the constructor for its superclass
  - This creates a *chain* of constructor calls all the way up to Object
  - The default constructor calls the default constructor for its superclass
  - Note: generally, the term Factory Method is often used to refer to any method whose main purpose is to create objects

Constructor chaining

- Therefore, if you write a class with an explicit constructor with arguments, and you write subclasses of that class,
  - Every subclass constructor will, by default, call the superclass constructor with no arguments (which may not still exist)
- Solutions: Either
  - Provide a no-argument constructor in your superclass, or
  - Explicitly call a particular superclass constructor with super(***args***)

Proper use of constructors

- A constructor should *always* create its objects in a *valid* state
  - A constructor should not do anything *but* create objects
  - If a constructor cannot guarantee that the constructed object is valid, it should be private and accessed via a factory method

Proper use of constructors

- A factory method is a static method that calls a constructor
  - The constructor is usually private
  - The factory method can determine whether or not to call the constructor
  - The factory method can throw an Exception, or do something else suitable, if it is given illegal arguments or otherwise cannot create a valid object
  - public Person create(int age) { // example factory method
    if (age < 0) throw new IllegalArgumentException("Too young!");
    else return new Person(n);
    }

References

- When you declare a primitive, you also allocate space to hold a primitive of that type
  - int x; double y; boolean b;
  - If declared as a field, it is initially zero (false)
  - If declared as a local variable, it may have a garbage value
  - When you assign this value to another variable, you *copy* the value

References

- When you declare an object, you also allocate space to hold *a reference to* an object
  - String s; int[ ] counts; Person p;
  - If declared as a field, it is initially null
  - If declared as a local variable, it may have a garbage value
  - When you assign this value to another variable, you *copy* the value
    - ...but in this case, the value is just a *reference* to an object
  - You *define* the variable by assigning an actual object (created by new) to it

Methods

- A method may:
  - be public, protected, package, or private
  - be static or instance
    - static methods may not refer to the object executing them (this), because they are executed by the class itself, not by an object
  - be final or nonfinal
  - return a value or be void
  - throw exceptions
- The signature of a method consists of its name and the number and types (in order) of its formal parameters

Methods

- You overload a method by writing another method with the same name but a different signature
- You override an *inherited* method by writing another method with the same signature
  - When you override a method:
    - You cannot make it less public (public > protected > package > private)
    - You cannot throw additional exceptions (you can throw fewer)
    - The return types must be compatible

Methods

- A method declares formal parameters and is "called" with actual parameters
  - void feed(int amount) { hunger -= amount; } // amount is formal
  - myPet.feed(5); // 5 is actual
- But you don't "call" a method, you send a message to an object
  - You may not know what kind of object myPet is
  - A dog may eat differently than a parakeet

Methods

- When you send a message, the values of the actual parameters replace the formal parameters
  - If the parameters are object types, their "values" are references
  - The method can access the actual object, and possibly modify it
- When the method returns, formal parameters are *not* copied back
  - However, changes made to referenced objects will persist

Methods

- Parameters are passed by assignment, hence:
  - If a formal parameter is double, you can call it with an int
    - ...unless it is overloaded by a method with an int parameter
  - If a formal parameter is a class type, you can call it with an object of a subclass type
- Within an *instance* method, the keyword this acts as an extra parameter (set to the object executing the method)

Methods

- Local variables are not necessarily initialized to zero (or false or null)
  - The compiler *tries* to keep you from using an uninitialized variable
- Local variables, including parameters, are discarded when the method returns
- Any method, regardless of its return type, may be used as a statement

Generic methods

- Method that takes a List of Strings:
- ```
  private void printListOfStrings(List<String> list) {
      for (Iterator<String> i = list.iterator(); i.hasNext(); ) {
          System.out.println(i.next());
      }
  }
  ```
- Same thing, but with wildcard:
  - ```
    private void printListOfStrings(List<?> list) {
        for (Iterator<?> i = list.iterator(); i.hasNext(); ) {
            System.out.println(i.next());
        }
    }
    ```


Proper use of methods

- Methods that are designed for use by other kinds of objects should be public
  - All public methods should be documented with Javadoc
  - public methods that can fail, or harm the object if called incorrectly, should throw an appropriate Exception
- Methods that are for internal use only should be private
  - private methods can use assert statements rather than throw Exceptions
- Methods that are only for internal use by this class, or by its subclasses, should be protected
- Methods that don't use any instance variables or instance methods should be static
  - Why require an object if you don't need it?

Proper use of methods

- Ideally, a method should do only one thing
  - You should describe what it does in one simple sentence
  - The method name should clearly convey the basic intent
    - It should usually be a verb
  - The sentence should mention every source of input (parameters, fields, etc.) and every result
  - There is no such thing as a method that's "too small"
- Methods should usually do *no* input/output
  - Unless, of course, that's the main purpose of the method
  - Exception: Temporary print statements used for debugging
- Methods should do "sanity checks" on their inputs
  - Publicly available methods should throw Exceptions for bad inputs

Proper use of polymorphism

- Methods with the same name should do the same thing
  - Method *overloading* should be used only when the overloaded methods are doing the same thing (with different parameters)
  - Classes that implement an interface should implement corresponding methods to do the same thing
  - Method *overriding* should be done to change the details of what the method does, without changing the basic idea

Proper use of polymorphism

- Methods shouldn't duplicate code in other methods
  - An overloaded method can call its namesake with other parameters
  - A method in a subclass can call an overridden method ***m*(*args*)** in the superclass with the syntax super.***m*(*args*)**
    - Typically, this call would be made by the overriding method to do the usual work of the method, then the overriding method would do the rest

Program design

- Good program design pays for itself many times over when it comes to actually writing the code
- Good program design is an art, not a science
- Generally, you want:
  - The simplest design that could possibly work
  - Classes that stand by themselves, and make sense in isolation
  - Aptly named methods that do one thing only, and do it well
  - Classes and methods that can be tested (with JUnit)

What happens when an exception is thrown

- An exception object is created *(on the heap)*
- The current "context" is halted/aborted
- Execution starts in some error handling code
  - Can be in current method
  - Can be external to current method
- The error handling code has access to the exception object which can be used to
  - Access a String message contained in the exception
  - Determine what type of exception was thrown
  - Print a stack trace
  - Other cool stuff (like rethrow the exception, increment a counter, etc.)

Vectors

- Vector is a *class* that provides a dynamic collection, similar to a Linked List, Queue, etc.
- Must be instantiated via "*new*" to get an *instance* of Vector.
- Vector elements are accessed via various utility methods

Commonly used methods

size( ) **returns current number of elements.**

elementAt(int index) **returns reference to element at specified index.**

insertElementAt(Object obj, int index) **ala insertion into linked list (but slower); cannot do at end.**

addElement (Object obj) **adds to end.**

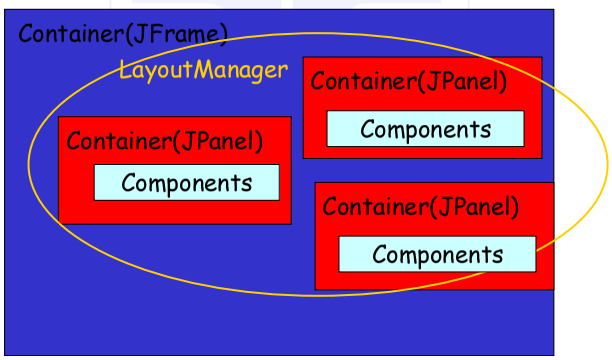
Graphical User Interface

- Components, Containers, Layouts
- Components
  - an object having a graphical representation that can be displayed on the screen and that can interact with the user.
  - e.g. Canvas, JButton, JLabel, JRadioButton, JTextField, JSlider,
- Container
  - public class **Container** extends Component
  - A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.
  - Components added to a container are tracked in a list.
  - e.g. JFrame, JPanel

LayoutManager

- public interface **LayoutManager**
  - Defines the interface for classes that know how to lay out Containers.
  - e.g. BorderLayout, FlowLayout, GridLayout

Big Picture



GUI Events

- What is event driven programming?
- Events and event listeners
- How do I write an event handler?
- How do I register an event handler?

Applets and threads

- Applets vs. standalone applications
- Methods in an applet
- Applet limitations
- Threads
  - create, start, stop/pause a thread
  - Applet animation using threads

File IO

- Based on Streams
  - Character (aka text)
    - Readers (Input) [i.e. FileReader]
    - Writers (Output)[i.e. FileWriter]
  - Byte (aka binary)
    - InputStream (Input) [i.e. FileInputStream]
    - OutputStream (Output) [i.e. FileOutputStream]
  - Processing Stream
    - Wraps Character or Byte streams to provide more functionality or filter stream
    - Most common: Buffered streams to allow line at a time processing [i.e. BufferedInputStream, BufferedReader]

Basic Exam Format

- No computers or cell phones
- Bring pen and paper with you
- Two parts:
  - Closed book part (cca. 1 hour)
    - Questions on OO and Java concepts
    - Be able to contrast and exemplify concepts
  - Open book, open notes (cca. 1 h 50 min)
    - One or two small problems to solve on paper
    - For this part do not forget to bring some documentation (notes, lab notes, book) as an aid

Success!