Java review

part 1: classes & objects
Anatomy of a Java program

• A (non-empty) set of classes containing:
  – At least one member method
  – 0 or more member variables and/or constants

• Problem: The CS1 syndrome!
  – Most real programs do not consist of one method (main) using one or more simple algorithms to solve a problem
  – A pure OOP approach would describe a Java program as a set of interacting objects
  – How do objects interact?
Terminology

- Program
- Object
- Class
- Method
- Variable
- Argument
- Parameter
Objects

• Data structures – literally, storage containers for data – constitute object knowledge

• Operations an object can perform on data are its actions

• Object knowledge and actions are specified by its class

• In a class definition:
  – knowledge: member variables
  – actions: member methods
Information hiding

• Separation of program specification from its implementation
  – several programmers can work on different facets of same problem, knowing specs others will meet (and not having to know how specs are met)
  – an **ADT** (abstract data type) is a class presented to client programmers with information hiding

• What do you, as the client programmer, need to know?
Class

• Basis for a Java program, but more to the point: *specification for a data type*

• An object is an *instance* of a class

• Class specification includes these members:
  – data (variables)
  – constructor(s)
  – methods
Example: a throttle

• Models physical object that controls flow of fuel to an engine

• Attributes include:
  – shutoff point: 0% fuel flow
  – progressively higher flow rates represented by lever position; maximum is fully open, 100% fuel flow
  – each position between 0 and max is proportionately higher than the previous

source: www.flypfc.com
Member variables

• We need to represent the attributes described on the previous slide:
  – number of throttle positions (max)
  – current position (position)

• Fill in the blanks of the class definition (next slide)
public ____________ Throttle
{
    // member variables:
    private _______ max;
    _______ _______ position;
}
Constructor

- Method for initializing instance variables of a new object
- For the Throttle class, we’ll define two constructors:
  - a default constructor that presets a chosen maximum value
  - a constructor that allows the client programmer to choose the maximum value
// class definition continued
// default constructor:

public Throttle () {
    max = 100;
    position = 0;
}

// second constructor:
public _____________ (int cap) {


}

// second constructor:
Questions

• What happens if you don’t define a constructor for your class?

• Is there an advantage to defining your own constructor?
Instance methods - Accessors

• Accessors provide information about an object’s state
• Accessors do not change the state of the object
• An accessor method always returns a value
Why not use public instance variables instead?

- Accessors support the principle of information hiding; client programmers don’t need to know anything about implementation (e.g., names of the variables), just how to get the information they need (return value from accessor).

- Use of accessors (instead of public variables) allows change to the implementation of a class without any required change to client code.

- Private data can’t be modified in ways the author of the class didn’t intend.
Accessors for the Throttle class

- `getFlow`: returns value indicating proportion of max is currently flowing
- `isOn`: returns value indicating whether or not throttle is turned on
// accessor methods

public _____________ getFlow () {

}

public _____________ isOn () {

}
Instance methods - modifiers

- Also known as mutators: methods that make specific changes to object’s state
- Usually void – why?
Modifiers for Throttle class

• shutOff: turns off Throttle
• shift: moves Throttle’s position up or down by a specific amount
  – how will this amount be specified?
  – how do we handle a value that goes out of bounds (<0 or >max)?
// modifier methods:
public void shutOff () {

}

public void shift (_________ setting) {

}
Using a class

• Once a class has been established, you can use instances of the class (objects) in other programs

• Remember, the class is a data type; in order to perform the actions specified by its instance methods, we need to instantiate an object
Instantiating objects

• Object creation is a two step process:
  – declaration:
    Throttle choke;
  – initialization:
    choke = new Throttle(50);

• This is often done in a single step:
  Throttle choke = new Throttle(50);
Calling methods

• Once we have an object, we can call instance methods defined in the class

• The **calling object** and method are specified, along with any necessary arguments; syntax:

  ```
  objectName.methodName(arg(s))
  ```

• Write a line of code that adjusts choke (the Throttle object created on the previous slide) up 3 steps:
Key points to remember

• Objects are instances of classes
• Object names are references to objects
• Objects are created using the new operator and one of the class’s constructors

• What happens in the following situation? How do we avoid the problem?
  Throttle uhoh;
  uhoh.shutOff();
Simple data types & wrapper classes

• Simple data types are the built-in types provided as part of the Java language: int, double, char, etc.

• These differ from the vast majority of data types we usually work with in Java, in that variables of the simple types actually contain values, rather than referring to objects

• For each primitive data type, the Java API includes a wrapper class, which allows programmers to exchange information between objects and simple-type variables
Wrappers & their uses

• The wrapper classes are commonly used when we have data in the form of an object (e.g. a String) and want to convert that data to a simple type, or vice versa; the code fragment below illustrates such a situation:

```java
String s = JOptionPane.showInputDialog(null, “Please enter your age:”);
int n = Integer.parseInt(s);
```
Primitive to wrapper conversion

• We can create an instance of a wrapper object in the same way as we create any object, as in the example below:
  
  ```java
  Double dobj = new Double(17.35);
  ```

• We can obtain the original primitive value used to construct the object using a wrapper value method:
  
  ```java
  double d = dobj.doubleValue();
  ```
Primitive to wrapper conversion

• The act of converting a primitive value to a wrapper object is called **boxing**

• Starting with Java 5.0, we can shorten this code by using automatic boxing, as in the example below:

  ```java
  Character cobj = '%';
  ```

• Conversion in the other direction (aka **unboxing**) can also be done automatically:

  ```java
  char c = cobj;  // automatically calls charValue()
  ```

• Automatic boxing and unboxing applies to parameter passing as well as direct assignment
An object by any other name ...

• Unlike primitive variables, variables of any class data type are not direct containers of data
• Each object variable can refer to the address of actual data
• In fact, object variables would be called, in any other language, pointers
• An important distinction: an object’s name is an indirect reference (indirectly) to (a set of, usually) value(s), while a primitive variable’s name is a direct reference to a value
Null references

• Instead of creating an object immediately, you can declare a variable and set it to null:
  Throttle t = null;

• You can also set an object reference to null after it’s been in use:
  Throttle t = new Throttle();
  t.shift(30);
  .... // after a while, finished with this object
t = null;
The null value

- The value null is a special constant that literally means no value
- Any object variable (that is, any pointer) can be assigned null
- The advantage of using this value is that we can test any object reference against it (using the == operator) and avoid program errors that might otherwise result
- The null value can stand in for any object type’s value – so, for example, we can pass null to a method instead of passing an object
Much more ado about nothing

- You may have already been bitten once or twice by the “Null Pointer Exception” runtime error
- From this you might deduce (correctly), that object variables are automatically assigned null unless they are assigned to point to something using the new operator and an appropriate constructor
- So why make an explicit null assignment?
  - good programmer habit: serves to remind you it’s something to check for
  - many programming languages aren’t as gentle as Java; an uninitialized pointer in C or C++ is like a loaded gun – not a good idea to wave that thing around and not know where you’re pointing
Why use null references?

• Helps conserve system resources; even though Java has automatic garbage collection, not a bad idea to help out

• Means every object reference can be counted on to be in one of two conditions:
  – refers to an object (not null)
  – refers to NO object (null)

• We can test an object reference to see if it is null; this means we can always avoid a NullPointerException; see code on next slide
// One way:

if (t != null)
    System.out.println("Current setting is " + t.getFlow());
else
    System.out.println("Can’t get setting – no active throttle");

// Another way:
try {
    System.out.println("Current setting is " + t.getFlow());
} catch (NullPointerException e) {
    System.out.println("Can’t get setting – no active throttle");
}
Consider the code snippet below:

```java
Throttle t1=null, t2=null;
t1 = new Throttle(100);
t2 = t1;
t2.shift(50);
System.out.println("Current value of t1 is " + t1.getFlow());
```

• What is output? Why?
Since both variables refer to the same object, a change invoked by one shows up in both.
A better way

• Using the assignment operator with objects does not copy the object, it only copies its address, leading to the situation we’ve just seen

• If we need two identical objects, we could simply instantiate two objects with the same parameters:
  Throttle t1 = new Throttle(100);
  Throttle t2 = new Throttle(100);

• What would this look like? Is (t1 == t2) a true statement?
The new operator & anonymous objects

• An object can be created and assigned to an object variable using the new operator and its constructor
• We can also create objects to used on the fly, without any permanent reference
• Such an object is called an anonymous object
Copy constructors and cloning

• Constructors are typically used to initialize instance variables, and most classes provide multiple constructors, including a default constructor

• Some classes also provide a special copy constructor, which is used to create an identical copy of an existing object

• The code on the next slide illustrates a class with a copy constructor
public class Some Object implements Cloneable{
    int example;
    // default constructor:
    public SomeObject() {
        example = 0;
    }

    // copy constructor:
    public SomeObject(SomeObject original) { 
        this.example = original.example;
    }

    // etc.
}
The clone() method

- The clone() method (like toString() and equals(), this is inherited from Object) provides essentially the same functionality as a copy constructor
- The clone() method is the preferred form, according to the Java API
- The next slide shows how each of these methods could be used to create a copy of an object
Example

SomeObject first = new SomeObject();
// creates original object using default constructor

SomeObject second = new SomeObject(first);
// creates identical object using copy constructor

SomeObject third = (SomeObject)first.clone();
// creates another identical object using clone() method
// and explicit type cast