More Java

CS2704: Object-Oriented Software Design
and Construction

Constantinos Phanouriou
Department of Computer Science
Virginia Tech

Outline

• Classes
  – class declaration
  – static members
  – access controls
  – packages
  – member functions (including constructors and “destructor” equivalents)
  – exceptions
  – creating own classes

• Objects

• Object Reference Variables
A touch of class

• Why class?
  – encapsulation
    • packaging together of data structures and code that
      manipulates those structures
  – information hiding
    • implementation details are hidden; “client” code
      cannot become dependent on current structures
      (facilitating future changes and extensions)

• Classes in Java?
  – Generally similar to C++
  – Where there are differences, Java has gone back to other
    older models (like original Simula67, Apple’s Object Pascal (1980s),
    Eiffel, and to lesser extent Smalltalk)

  – A pervasive change
    • as in Smalltalk and Eiffel, all Java classes regarded as being
      extensions of a “base class” provided by the language
      (inevitably called class Object)
A universe of classes

- One new contribution
  - A systematic way of uniquely naming every class invented by any Java programmer on the network!
  - OK, not routinely used; but does provide solution to problem of putting together code using class libraries from different sources --- each library could for example define a class List but there would be no confusions because when using a List would have to specify its unique name
    - Example: edu.vt.cs.cs2704.List

  *C++ ‘namespace’ feature provides similar control*

class declaration

- A class declaration will
  - specify any use of inheritance (next topic), if nothing specified then “extends Object” is implicit
  - define constants (if any)
  - specify instance data members (type, access, initial value)
  - specify class data members (shared static)
  - define instance and class member functions (Java prefers the term “methods”)
class declaration

- A class declaration is a single syntactic unit — it must contain the definitions of all member functions.

- No restrictions on order of declaration of different members of a class.

- Each member declaration should include its “access specification” (public, private, protected)
  (a default value applies if nothing specified, you don’t continue with most recent specification as you do in C++)

```java
class MyCanvas extends Canvas {
    Structure s;
    public MyCanvas() { }
    public void setStructure(Structure st){ }
    public void paint(Graphics g){ }
}
```

```java
class Structure extends Object {  
    final int kPTSMAX = 100;
    double[] fX = new double[kPTSMAX];
    ...
    int fNumPts;
    private int GetCount(BufferedReader input)  
        {... }
    ...
}
```
static qualifier

• “static” has same meaning in class declaration as it did in C++.
  – a “static” data member is one that is shared by all instances of the class
  – a “static” member function (method) is one that does not use any data members or only uses static data members

static qualifier

• As in C++, static member functions are invoked via the class (syntactic difference, use `<class name>..<static function name>` rather than `<class name>::<static function name>`)  
  • Example:
    - `int Integer.parseInt(String)`
Access controls

• Java has controls similar to, but not identical to those of C++.
• Access controls depend in part on new Java concept of a “package”
  – package --- a group of classes that in some way belong together, and whose instances often interact

Access controls

• Where C++ distinguished
  – class,
  – subclasses,
  – instance,
  – friends
• Java has
  – class,
  – other classes in same package,
  – subclasses in same package,
  – other subclasses,
  – instance.
• “package” access has some similarities to C++’s friend relations
<table>
<thead>
<tr>
<th></th>
<th>Public</th>
<th>Default</th>
<th>Protected</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Package</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Subclass in package</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Other subclass</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Packages, files, “modules”, classes, ...

- **File**: something that your operating system understands, a unit for editing, copying etc
- **Module**: a unit for compilation (if C++ then typically two files - header and implementation; for Java it is one file); so, for Java “a file is a module”
Packages, files, “modules”, classes, ...

- If you want your classes to be reusable components, then tempting to think in terms like “a file is a module is a class”
- Each file then contains just one class
  - class Queue
  - class Structure
  - class MyCanvas
  - ...

“a file is a module is a class”
In practice, that isn’t really appropriate.
- Your class List needs an auxiliary Link class
  - Although some might want your List class, no one else needs to know about ‘Links’.
  - Having separate file for Links just complicates life.
- A class like “MyCanvas” really has no prospects of reuse
Java “module” (file)

- Generally, a Java module will contain some number of auxiliary classes along with a single principal class.
- A source file can only contain one ‘public’ class
  - if only one class defined in file, it is public
  - otherwise, one and only one class must be declared as public

Example

in List.java:

```java
public class List {
    Link link;
    ...
}

class Link { ... }
```
package

• Java’s packages provide an organizational unit greater than a single file
• Packages provide a defined and therefore clearer model of what we’ve been referring to as “class libraries”
  – a package is a set of modules, each module defines one (publically known) class (and possibly some associated auxiliary classes)
  – the classes defined in the different modules in a package “belong together”

package

• Typical packages
  – java.math
    • classes for multi-digit arithmetic
  – java.awt
    • Graphics User Interface classes
package access

- The awt classes provide examples of where the “package” access mechanisms might be useful
- GUI classes typically work closely together
  - windows interact with subwindows and menus etc
  - so, there may be places where want a Window to have more access to a Menu than would really wish to allow to an ordinary client

package and directory

- Just as a file must have the same name as the (principal) class that it contains,
- so a package name must match a directory name.

- The java math classes are all in the “math” directory which is a subdirectory of the “java” directory.
package and classnames

- It is this naming convention that provides the basis for the scheme that assigns a unique name for every class.
- Unique class name: computer domain name (actually reversed), pathname down to package directory, class name.

packages and imports

- Your classes can specify data members that are instances of other classes - specifying qualified class name

```java
class Communications {
    java.net.Socket fMySocket;
    java.net.URL fURL;
    ...
    void Setup(String host, String file) {
        ...
        fURL = new java.net.URL("http:\", host, 80, file);
        ...
```
packages and imports

• An import statement saves you from having to specify such qualified class names
  import java.net.URL;
  ...
  class Communications {
    URL fURL;
    ...
  }

• You can import individual classes, or all classes in a package (import java.net.*;)

• If you do manage to import two class URLs from different packages, you will again have to use qualified class names.

Example

in edu/vt/cs/cs2704/List.java:
package edu.vt.cs.cs2704;
public class List {
  Link link;
  ...
}
class Link { ... }

import edu.vt.cs.cs2704.List;

List l;
edu.vt.cs.cs2704.List ll;
Member functions

- From C++, you should remember:
  - accessor functions
    - allow code read access to information belonging to an object
  - mutator functions
    - make changes to an object
  - constructors
    - initialize an object
  - destructors
    - release resources held by an object

Java

- Accessors and mutators
  - not much change from C++ (except without the const tag that C++ can use to flag accessors)

- Constructors
  - Java changes working of overloaded constructors (one can call another, which you can’t do in C++ --- this())

- Destructor?
Java & destructors

• Destructor
  – *Which C++ classes had destructors?* The resource manager classes.
  – *Why did they have them?* So that instances could free any resources that they had acquired.
  – *What kinds of resources?* Most commonly, the resource was memory (heap space used for auxiliary data); other resources were things like files, sockets, ...

Java & destructors

• Java has automated garbage collection so it is not necessary to explicitly free memory.

• So, “Java doesn’t need destructors.”

• *But what about other types of resource?*
finalize()

- finalize() apparently serves roughly the same role as a destructor.
- When Java’s garbage collector decides to reclaim space for a discarded object, it first calls that object’s finalize() method (provided by class Object as a do nothing, and possibly redefined in subclass).
- The finalize() method can free other resources.

finalize() ?

- But, no guarantee of when garbage collector will get to an object!
- If the objects in your class do use system resources like files, you shouldn’t really rely on finalize().
- Put your resource release code in a method that you define (tidy_up(), dispose(). or follow class Applet and have a destroy() method)
Exception specifications

- In C++, a function declaration can optionally include an exception specification.
- In Java, specification is *not* optional. If a member function can throw an exception (or pass on an exception thrown by some other function that it calls) then this *must* be stated in the declaration.

```java
class Integer ... {
    ...
    public static int parseInt(String s) throws NumberFormatException
    ...
}
```
import java.io.*;

public class Example {
    public static void main(String[] args) {
        if(args.length<1)
            System.out.println("Need name of data file");
        else Process(args[0]);
    }

    static void Process(String filename) {
        ...
    }
}

import java.io.*;

public class Example {
    public Example() {
    }

    public static void main(String[] args) {
        if(args.length<1)
            System.out.println("Need name of data file");
        else {
            (new Example).Process(args[0]);
        }
    }

    void Process(String filename) {
        ...
    }
}
static void Process(String filename)
{
    // Open file and package with adapter
    // classes until have BufferedReader from which
    // can read lines

    // loop reading data values (have to handle
    // exceptions)
    // read line
    // attempt to extract double
    // accumulate sums etc

    // compute mean
    // compute standard deviation
}

static void Process(String filename) {
    File f = new File(filename);
    if(!f.canRead()) {
        System.out.println("Can't read from that file");
        Runtime.getRuntime().exit(1);
    }

    FileInputStream fStream = null;
    try {
        fStream = new FileInputStream(f);
    } catch (FileNotFoundException e) {
        System.out.println("No such file?");
        Runtime.getRuntime().exit(1);
    }

    BufferedReader datareader = new BufferedReader(new InputStreamReader(fStream));
    ...
... int count = 0; double sum = 0.0; double sumsq = 0.0;

for(;;) { .. next page .. }

if(count == 0) {
    System.out.println("No data"); return;
}

double mean = sum / count;
System.out.println("Mean " + mean);
if(count<2) { System.out.println("Too few items to calculate standard deviation");return; }

double stdev =
    Math.sqrt((sumsq - sum*sum/count) /(count-1));
System.out.println("Standard deviation " + stdev);
```java
for(;;) {
    String s = null;
    try {
        s = datareader.readLine(); s.trim();
    } catch (IOException e) {
        System.out.println("Error reading file");
        Runtime.getRuntime().exit(1);
    }
    double d = 0.0;
    try {
        d = Double.valueOf(s).doubleValue();
    } catch (NumberFormatException n) {
        System.out.println("Bad data in file");
        Runtime.getRuntime().exit(1);
    }
    if(d == 0.0) break;
    count++; sum += d; sumsq += d*d;
}
```

### Arrays

- An array is essentially a kind of object
  - gets allocated on heap
  - supports [ ] access operator
  - provides “length” accessor
- A variable that is to become an array is declared as an array of specified type
- Space for array allocated by **new** operator (when specify actual array size)
Arrays

- Array of built-in (primitive) types
  
  ```java
  int[] data; // or int data[];
  ...
  data = new int[25];
  ```

- “data” is an array of integers
  
  ```java
  ...
  data[3] = 999;
  ```

Arrays

- Arrays of built in types can also be created with an initializer:
  
  ```java
  int gradepts[] = { 0, 45, 50, 65, 75, 85 }; 
  ```
Arrays

- Array of a class type:
  ```
  class Student {
    ...
  }

  Student[] enrollment;
  ...
  enrollment = new Student[25];
  ```

- Now have an array of “object reference variables” (pointers), *NOT* an array of Student objects

- `enrollment = new Student[25];`
- array of “object reference variables”
- Create the actual objects separately:
  ```
  for(int i=0;i<25;i++)
    enrollment[i] = new Student();
  ```

  note requirement for parentheses (different in C++)
Arrays

- Array variable itself is simply a “pointer”

```java
int a[];
int b[];
...
a = new int[77];
...
b = new int[77];
...
// This b = a; does not copy contents of array!
b = a; // Now have two pointers to the same // structure, the array that b used // to point at will be garbage collected
```