Intermediate Java

Thanks to Nick Parlante for much of this handout

This doc gathers miscellaneous intermediate Java topics needed in CS108. Many of these are topics are not given full coverage in an introductory Java course such as CS106A. **Assignment 0**: read through this handout once to have familiarity with these topics, as they will all appear somewhere in the CS108 code. We will review a few of the topics here plus Java Generics in the section Thu eve.

The first part deals with many little topics: arrays, shallowness, packages, command line java, static variables and methods, file reading, exceptions. The second part concentrates on the Java Collection classes: List, Map, and Set. The 2nd part has different typography, as I'm preparing it in HTML to be published on the web.

Pervasive Shallowness in Java / GC

- In C++, when you use an "=" for assignment, you always wonder how "deep" the copy is. In Java, = is always "shallow" -- copying the pointer but not the object. Suppose we have some Foo class:

```
Foo x = new Foo(1);
Foo y = new Foo(2);

x = y;   // shallow -- just make x point to 2nd Foo, Garbage Collector gets the 1st Foo
bar(x);  // same, just pass a pointer to the Foo object into bar()
```

- The garbage collector (GC) does the de-allocation bookkeeping for us, de-allocating a heap object when the number of pointers to it goes to zero. "new" allocates, the GC cleans up.
- It is common in Java to have a few objects and proliferate pointers to those few objects all over the place. Many parts of the program share ownership of the objects. We can afford to have pointers spread all over, since the GC manages the de-allocation bookkeeping.
- This pervasive sharing can be a problem if we want to change an object, since many parts of the program may be pointing to it.
- Sharing is not a problem if the object is "immutable" -- having no mutating methods so the object never changes once it is constructed.
- Another solution is to make a copy of the object before changing it. Some classes provides a "copy constructor" (C++ terminology) -- a constructor that takes a pointer to an existing object as an argument, and initializes a new object to be a copy. In general, making copies of things is something more often done in C++ than in Java. I think this is because the GC gives us the freedom to not make copies in many situations.

```
Foo x = new Foo(1);
Foo copy = new Foo(x);  // make a copy of x (depends on existence of a copy constructor)
```

- There is also an old java object copying facility called "Cloneable", but it turned out to be a messy design and has fallen out of favor. I recommend that you never use it.

Array Review

- As a warm-up, remember how arrays work in Java:

```
int[] a = new  int[100];  // Allocate array in the heap, a points to the  array
a[0] = 13;
a[1] = 26;
a.length       // read-only access .length -- 100 (not .length(), that's for strings)
a[100] = 13;   // throws ArrayOutOfBounds exception at runtime
```

"Arrays" Utility Class

- **Arrays** Class -- The Arrays class contains many static convenience methods that work on arrays -- filling, binary search, array equals, sorting. Type "Arrays." in Eclipse, and let it autocomplete to show you the available static methods.
- There is also a method in the System class, `System.arraycopy()`, that will copy a section of elements from one array to another (it also works correctly, copying elements within a single array). If the number of elements to copy is large, `System.arraycopy()` will probably be faster than your hand-written for-loop
- `System.arraycopy(source-array, source-index, dest-array, dest-index, length-to-copy);`

Arrays.equals(), deepEquals()

- The default `a.equals(b)` does not do a deep comparison for arrays, it just compares the two pointers. This violates the design principle of least surprise, but we're stuck with it for backwards compatibility for now.
- Use the static `equals()` in the Arrays class -- `Arrays.equals(a, b)` -- this checks that 1-d arrays contain the same elements, calling `equals()` on each pair of elements. For multi-dimensional arrays, use `Arrays.deepEquals()` which recurs to check each dimension.

Multidimensional Arrays

- An array with two or more dimensions is allocated like this...
  - `int[][] grid = new int[100][100]; // allocate a 100x100 array`
- Specify two indexes to refer to each element -- the operation of the 2-d array is simple when both indexes are specified.
  - `grid[0][1] = 10;     // refer to (0,1) element`
- Unlike C and C++, a 2-d java array is **not** allocated as a single block of memory. Instead, it is implemented as a 1-d array of pointers to 1-d arrays. So a 10x20 grid has an "outer" 1-d array length 10, containing 10 pointers to 1-d arrays length 20. This detail is only evident if we omit the second index -- mostly we don't need to do that.

```
int temp;
int[][] grid = new int[10][20];  // 10x20 2-d array
grid[0][0] = 1;
grid[9][19] = 2;
temp = grid.length; // 10
temp = grid[0].length; // 20

grid[0][9] = 13;
int[] array = grid[0];  // really it's just a 1-d array
temp = array.length; // 20
temp = array[9]; // 13
```

- Note that `System.arraycopy()` does not copy all of a 2-d array -- it just copies the pointers in the outer 1-d array.

Packages / Import

Java Packages

- Java classes are organized into "packages". e.g. `java.lang` holds built-in Java classes, `com.oracle` contains Oracle corporation's classes.
- Every class has a "long" or "fully qualified" name that includes its package. e.g. the full name of the `String` class is `java.lang.String`.

- e.g. for `ArrayList` the full name is `java.util.ArrayList`. The java.util package contains utilities for Java
- In this way, a class `Account` in your e-commerce package will not conflict with a class also named `Account` in the sales-tax computation package that you are using, since they can be distinguished by their fully qualified names -- `com.foo.Account` vs. `com.taxcorp.Account`.
- If you need to find the package of a class, you can look at its javadoc page -- the fully qualified name is at the top.
- In the compiled .class bytecode form of a class, the fully qualified name, e.g. `java.lang.String`, is used for **everything**. The idea of a "short" human readable name is a form that is only used in the .java source files. All the later bytecode stages in the JVM use the full name.

Package Declaration

- A "package" declaration at the top of the .java source file indicates what package that class goes in
- `package stanford.cslib; // statement at start of file`
- If a package is not declared, the class goes into the one, catch-all "default" package. For simplicity, we will put our own classes in the default package, but you still need to know what a package is.

Compile Time Import

- `import java.util.*;` makes all the classes in that package available by their short names.
- `import java.util.ArrayList;` makes just that one class available by its short name
- When the compiler sees something like `Foo f = new Foo()`, how does it know which `Foo` you are talking about?
- Can write it as new `java.util.Foo()` -- can always write the full name to disambiguate which class you mean
- Can add an `import java.util.*;` at the top of the file to make the short name work
- At compile time, the compiler checks that the referenced classes and methods exist and match up logically, but it does not link in their code. Each reference in the code, such as to `java.lang.String` is just left as a reference when the code is compiled. Later, when the code runs, each class (e.g. `java.lang.String`) is loaded when it is first used. Unlike in C, where you can statically link in a library, so its object code is copied into your program code.
- The `.*` version of import does not search sub-directories -- it only imports classes at that immediate level.
- Having lots of import statements will not make your code any bigger or slower -- it only allows you to use shorter names in your .java code.

"List" Example

- In the Java libraries, there are two classes with the name `List` -- `java.util.List` is a list data structure and `java.awt.List` is a graphical list that shows a series of elements on screen.
- Could import `java.util.*` at the top of a file, in which case `List` in that file refers to `java.util.List`. Or could import `java.awt.*`, in which case `List` is `java.awt.List`. If both imports are used, then the word `List` is ambiguous, and we must spell it out in the full `java.util.List` form.
- In any case, the generated .class files always use the long `java.util.List` form in the bytecode.
- Compiling and referring to `java.util.List` does not link the `java.util.List` code into our bytecode. The `java.util.List` bytecode is brought in by the JVM at runtime. This is why your compiled Java code which many standard classes, is still tiny -- your code just has references to the classes, not copies of them.

.jar Files

- A .jar file is a standard way of packaging a bunch of .class files all together.

- e.g. the file junit.jar contains the classes that implement junit testing.
- On leland, there is a copy of the junit.jar file for your use at /usr/class/cs108/jar/junit.jar

Compile Time Classpath

- To compile java code that uses a `Foo` class, the compiler needs access to `Foo` to check that the method prototypes etc. are used correctly.
- If you cannot compile because "cannot find Symbol" for a class your code uses -- the likely cause is that the classpath does not include the class you use.
- Compiling **does not** link in the `Foo` code. The `Foo` bytecode is pulled in just used for checking that the interfaces all match up.
- Normally, the compiler has its own access to the standard Java Development Kit (JDK) classes (e.g. `String`, `ArrayList`), so you do not need to bring those in.

Java Command Line

Almost all compiling and running can be done from within Eclipse. However, it's useful to know how to work from the command line as well.

Command Line Compiling-- javac

- The command line java compiler is called `javac`
- Suppose we have a directory full of .java files. The easiest way to compile them all is with the command `javac *.java`
- This will write a bunch of .class files containing the bytecode result of the compile. `Foo.class` contains the bytecode for the `Foo` class, and `Foo$Bar.class` contains the bytecode for a `Bar` class defined inside of `Foo`.
- Alternately, you can let Eclipse generate the .class files in your directory, but use the command line to run the code.
- Suppose we are compiling a `Foo` class in the package java.util. Upon compilation, directories are created to represent the package path. So there's a directory `java` and inside that a directory `util`, and inside that `Foo.class`. For the default package, .class files are simply created in the current directory.
- Having a strict scheme for how .class files are named and stored means that at runtime, the JVM can start with a class name referenced by the running program, e.g. `foo.bar.Account`, and quickly locate the `Account.class` file in the filesystem and load it.

Command Line Running -- java

- The `java` command runs a program, and the argument is the class that contains `main()`.
- e.g. `java JTetris`
- By default, java uses the current directory to look for referenced classes.
- This can be a handy to run a program repeatedly, changing the args by using the up-arrow in the shell to edit the arguments and re-run the program.

Command Line Classpath

- The `-classpath` argument to the command line compiler `javac` specifies the "classpath" which is a series of .jar files and directories where it should search for class definitions during the compile.
- The parts of the classpath are separated by colons (:). e.g. `javac -classpath foo.jar:/some/dir:. Bar.java`
  - `foo.jar`, `/some/dir`, and "." are the components of the classpath. With Java 5 `-classpath` can be abbreviated `-cp`.
- If no classpath is specified, the current directory "." is the default. If you specify a custom classpath, then you need to add the "." manually.

Run Time Classpath

- You also specify a `-classpath` for the `java` command to run a program (can be abbreviated "-cp"). If, at run time, the system cannot find the class you need, the likely cause is that the classpath does not include that class.
- At run time, the classpath is used to load the classes to actually run. As with javac, the current directory "." is the default classpath.
- Summary: the compile time classpath used to check classes during compilation, does not link them in. The run time classpath is used to find and load classes as they are referenced during the run.

Command Line Arguments

- The prototype for the special `main()` to start a program is `static void main(String[] args)` -- the `args` array refers to the command line arguments when the program is run. So the following runs the `Foo` class with the Strings `aaa` and `bbb` as the 2 arguments:

```
> java Foo aaa bbb
```

Sudoku Compiling Example

- This example shows compiling the Sudoku class in the file Sudoku.java, and then a `SudokuTest` class that depends on the separate junit.jar file.

```
elaine3:~/Sudoku> ls *.class          // no .class files to start
ls: No match.
elaine3:~/Sudoku> javac Sudoku.java   // compile creates .class files
elaine3:~/Sudoku> ls *.class
Sudoku$ColSpace.class  Sudoku$SmartComp.class  Sudoku$Spot.class          Sudoku.class
Sudoku$RowSpace.class  Sudoku$Space.class      Sudoku$SquareSpace.class
elaine3:~/Sudoku> java Sudoku        // Run the Sudoku class from current directory
time:32
moves:45
 1 6 4 7 9 5 3 8 2
 2 8 7 4 6 3 9 1 5
 9 3 5 2 8 1 4 6 7
 3 9 1 8 7 6 5 2 4
 5 4 6 1 3 2 7 9 8
 7 2 8 9 5 4 1 3 6
 8 1 9 6 4 7 2 5 3
 6 7 3 5 2 9 8 4 1
 4 5 2 3 1 8 6 7 9
elaine3:~/Sudoku> javac SudokuTest.java
SudokuTest.java:1: package junit.framework does not exist
import junit.framework.TestCase;
                       ^
SudokuTest.java:6: cannot find symbol
symbol: class TestCase
public class SudokuTest extends TestCase {
                                ^
// Errors because the SudokuTest code refers to JUnit classes that are not
// in this directory (default classpath is just ".")
// fix by adding junit.jar to the classpath
elaine3:~/Sudoku> javac -classpath /usr/class/cs108/jar/junit.jar SudokuTest.java
SudokuTest.java:7: cannot find symbol
symbol  : class Sudoku
location: class SudokuTest
        Sudoku basic;
        ^
...
// Now the problem is that it can't see the Sudoku class itself which
// is in the same directory -- need to add "." to the classpath
elaine3:~/Sudoku> javac -classpath /usr/class/cs108/jar/junit.jar:. SudokuTest.java
<that works>
```

Static

- Instance variables (ivars) or methods in a class may be declared `static`.
- Regular ivars and methods are associated with objects of the class.
- Static variables and methods are not associated with an **object** of the class. Instead, they are associated with the **class itself**.

Static variable

- A static variable is like a global variable, except it exists inside of a class.
- There is a single copy of the static variable inside the class. In contrast, each instance variable exists many times -- one copy inside each object of the class.
- Static variables are rare compared to ordinary instance variables.
- The full name of a static variable includes the name of its class.
  - So a static variable named `count` in the `Student` class would be referred to as `Student.count`. Within the class, the static variable can be referred to by its short name, such as "count", but I prefer to write it the long way, `Student.count`, to emphasize to the reader that the variable is static.
- e.g. `System.out` is a static variable `out` in the System class that represents standard output.
- Monster Example -- Suppose you are implementing the game Doom. You have a `Monster` class that represents the monsters that run around in the game. Each monster object needs access to a `roar` variable that holds the sound "roar.mp3" so the monster can play that sound at the right moment. With a regular instance variable, each monster would have their own `roar` variable. Instead, the `Monster` class contains a static `Monster.roar` variable, and all the monster objects share that one variable.

Static method

- A `static` method is like a regular C function that is defined inside a class.
- **A static method does not execute against a receiver object**. Instead, it is like a plain C function -- it can have parameters, but there is no receiver object.
- Just like static variables, the full name of a static method includes the name of its class, so a static `foo()` method in the Student class is called `Student.foo()`.
- The Math class contains the common math functions, such as `max()`, `abs()`, `sin()`, `cos()`, etc.. These are defined as static methods in the `Math` class. Their full names are `Math.max()`, `Math.sin()`, and so on. `Math.max()` takes two ints and returns the larger, called like this: `Math.max(i, j)`
- A `static int getCount()` {… method in the `Student` class is invoked as `Student.getCount();`
- In contrast, a regular method in the `Student` class would be invoked with a message send (aka a method call) on a Student object receiver like `s.getStress();` where `s` points to a `Student` object.
- The method `static void main(String[] args)` is special. To run a java program, you specify the name of a class. The Java virtual machine (JVM) then starts the program by running the static `main()` function in that class, and the `String[]` array represents the command-line arguments.
- It is better to call a static method like this: `Student.foo()`, NOT `s.foo()`; where `s` points to a `Student` object, although both syntaxes work.
  - `s.foo()` compiles fine, but it discards `s` as a receiver, using its compile time type to determine which class to use and translating the call to the `Student.foo()` form. The `s.foo()` syntax is misleading, since it makes it look like a regular method call.

static method/var example

- Suppose we have a `Student` class. We add a static variable and a static method for the purpose of counting how many `Student` objects have been created.

- Add a `static int numStudents = 0;` variable that counts the number of `Student` objects constructed -- increment it in the `Student` constructor. Both static and regular methods can see the `static numStudents` variable. There is one copy of the `numStudents` variable in the `Student` class, shared by all the `Student` objects.
- Initialize the `numStudents` variable with `= 0;` right where it is declared. This initialization will happen when the `Student` class is loaded, which happens before any `Student` objects are created.
- Add a static method `getNumStudents()` that returns the current value of `numStudents`.

```
public class Student {
    private int units;  // units ivar for each Student

    // Define a static int counter
    // to count the number of students.
    // Assign it an initial value right here.
    private static int numStudents = 0;

    public Student(int init_units) {
        units = init_units;

        // Increment the counter
        Student.numStudents++;
        // (could write equivalently as numStudents++)

    }

    public static int getNumStudents() {
        // Clients invoke this method as Student.getNumStudents();
        // Does not execute against a receiver, so
        // there is no "units" to refer to here
        return Student.numStudents;
    }

    // rest of the Student class
    ...
}
```

Typical static method error

- Suppose in the static `getNumStudents()` method, we tried to refer to the `units` instance variable...

```
public static int getNumStudents() {
    units = units + 1; // error
    return Student.numStudents;
}
```

- This gives an error message -- it cannot compile the `units` expression because there is no receiver object to provide instance variables. The error message is something like cannot make static reference to the non-static `units`. The `static` and the `units` are contradictory -- something is wrong with the design of this method.
- Static vars, such as `numStudents`, are available in both static and regular methods. However, ivars like `units` only work in regular (non-static) methods that have a receiver object.

Files

File Reading

- Java uses input and output "stream" classes for file reading and writing -- the stream objects respond to `read()` and `write()`, and communicate back to the file system. `InputStream` and `OuputStream` are the fundamental superclasses.
- The stream objects can be layered together to get overall effect -- e.g. wrapping a `FileInputStream` inside a `BufferedInputStream` to read from a file with buffering. This scheme is flexible but a bit cumbersome. As a pattern, this is known as the "decorator" pattern.

- The classes with "reader" or "writer" in the name deal with **text files**
  - `FileReader` -- knows how to read text chars from a file
  - `BufferedReader` -- buffers the text and makes it available line-by-line
- For non-text data files (such as jpeg, png, mp3) use `FileInputStream`, `FileOutputStream`, `BufferedInputStream`, `BufferedOutputStream` -- these treat the file as a plain sequence of bytes.
- You can specify a unicode encoding to be used by the text readers and writers -- defines the translation between the bytes of the file and the 2-byte unicode encoding of Java chars.

Common Text Reading Code

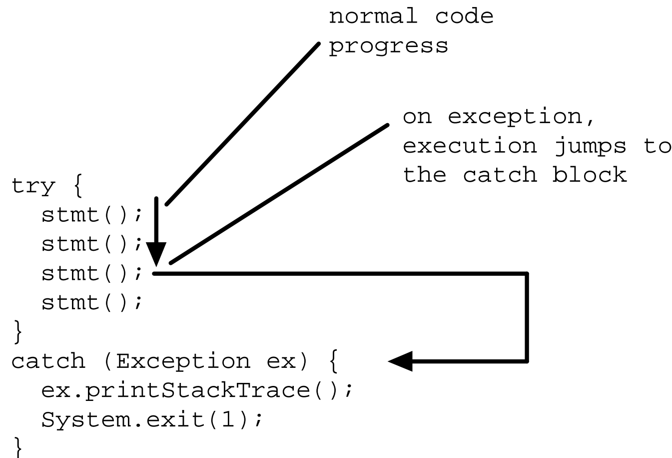
```
    // Classic text file reading code --  the standard while/readLine loop
    // in a try/catch.
    public void echo(String filename) {
        try {
            // Create reader for the given filename
            BufferedReader in = new BufferedReader(new FileReader(filename));

            // While/break to call readLine() until it returns null
            while (true) {
                String line = in.readLine();

                if (line == null) {
                    break;
                }

                // do something with line
                System.out.println(line);
            }

            in.close();
        }
        catch (IOException except) {
            // The code above jumps to here on an IOException,
            // otherwise this code does not run.
            // Good simple strategy: print stack trace, maybe exit
            except.printStackTrace();
            // System.exit(1);  // could do this too
        }

    }


// the while/readLine logic can be written more compactly as
// "while ((line=in.readLine()) != null) {"
```

Exceptions

An exception occurs at runtime when a line of code tries to do something impossible such as accessing an array using an index number that is out of bounds of the array or dereferencing a pointer that is `null`.

An exception halts the normal progress of the code and searches for error handling code that matches the exception. Most often, the error handling code will print some sort of warning message and then possibly exit the program, although it could take some more sophisticated corrective action.

Java uses a "try/catch" structure to position error-handling code to be used in the event of an exception. The main code to run goes in a "try" section, and it runs normally. If any line in the try section hits an exception at runtime, the program looks for a "catch" section for that type of exception. The normal flow of execution jumps from the point of the exception to the code in the catch-block. The lines immediately following the point of the exception are never executed.



For the file-reading code, some of the file operations such as creating the `FileReader`, or calling the `readLine()` method can fail at runtime with an `IOException`. For example, creating the `FileReader` could fail if there is no file named "file.txt" in the program directory. The `readLine()` could fail if, say, the file is on a CD ROM, our code is halfway through reading the file, and at that moment the Cheat comes in and hits the eject button and runs off with the CD. The `readLine()` will soon throw an `IOException` since the file has disappeared midway through reading the file.

The above file-reading code uses a simple try/catch pattern for exception handling. All the file-reading code goes inside the "try" section. It is followed by a single catch-block for the possible `IOException`. The catch prints an error message using the built-in method `printStackTrace()`. The "stack trace" will list the exception at the top, followed by the method-file-line where it occurred, followed by the stack of earlier methods that called the method that failed.

It is possible for an exception to propagate out of the original method to be caught in a try/catch in one of its caller methods, however we will always position the try/catch in the same method where the exception first appears.

Diagnosing Exceptions

When your program crashes with an exception, if you are lucky you will see the exception stack trace output. The stack trace is a little cryptic, but it has very useful information in it for debugging. In the example stack trace below, the method `hide()` in the `Foo` class has failed with a `NullPointerException`. The offending line was line 83 in the file Foo.java. The `hide()` method was called by `main()` in `FooClient` on line 23.

```
java.lang.NullPointerException
    at Foo.hide(Foo.java:83)
    at FooClient.main(FooClient.java:23)
```

In production code, the catch will often exit the whole program, using a non-zero int exit code to indicate a program fault (e.g. call `System.exit(1)`). Alternately, the program could try to take corrective action in the catch-block to address the situation. Avoid leaving the catch empty -- that can make debugging difficult since when the error happens at runtime, an empty catch consumes the exception but does not give any indication that an exception happened. As a simple default strategy, put a `printStackTrace()` in the catch so you get an indication of what happened. If no exception occurs during the run, the catch-block is ignored.

Java exceptions are actually organized into an inheritance hierarchy with the class `Exception` as the general superclass of exceptions. For example `IOException` is a subclass of `Exception`, and `FileNotFoundException` is a subclass of `IOException`. When an exception is thrown at runtime, it looks for the first matching catch (...) clause -- so `catch (Exception e)` would catch any type of exception, but `catch (IOException e)` would catch only `IOExceptions`.

In Java code, if there is a method call like `in.readLine()` that can throw an exception, then the compiler will insist that the calling code deal with the exception, typically with a try/catch block. This can be annoying, since the compiler forces you to put in a try/catch when you don't want to think about that case. However, this strict structure is one of the things that makes Java code reliable in production.

Some exceptions such as `NullPointerException` or `ArrayOutOfBounds` or `ClassCastException` are so common that almost any line of code can trigger them. These common exceptions are grouped under the `UncheckedException` class, and code is not required to put in a try/catch for them. All the other exceptions, such as `IOException` and `InterruptedException`, are called "checked" exceptions and the code is required to handle the exception or it will not compile. There is debate that perhaps it would have been better to make all exceptions unchecked -- making the code easier to write but a little less organized.