Today …

• Final exam “review”
• Java packages and jar files
• Reminder
  – Group projects on Thursday!
• Reading Assignment
  – Core: Ch. 10 pp. 493-500 (Jar files)
  – Core: Ch. 4 pp. 153-161 (Packages and classpaths)
Final overview

- Similar to the midterm (length, questions, etc.)
- Open book and notes
- Again, closed computer(s)
- Be sure you understand:
  - Answers to quizzes and exercises
  - Answers to questions on slides
  - Answers to midterm questions
- Be sure you have finished the reading assignments
- Worth 20% of your grade

Final overview

Comprehensive, but with a focus on material since midterm …

- From before midterm
  - Basic classes
    - static/instance, field/method, public/private/protected
  - Basic language features
    - primitive types, arrays, variable declaration and initialization
    - constants, casting, strings, loops
  - Inheritance
    - extending, overloading, overriding, this and super, constructors, dynamic binding
  - Inner classes
Final overview

Comprehensive, but with a focus on material since midterm …

• Since midterm
  – The `Object` class
    • inheritance, `toString()`, `clone()`, `equals()`, `hashCode()`
  – The `assert` keyword
  – Generics
  – Collection Framework
  – Exceptions
  – Packages (from today …)

Java packages

• A java “package” is a way to organize classes into groups

• We’ve already been using packages!
  – `java.util`
  – `java.io`
  – `javax.swing`
  – `java.awt`

• To use classes in these packages, we have to import
  – `import java.util.List` // import a utility class (List)
  – `import javax.swing.*` // import all the swing classes
Java packages

• “Static imports” are a new feature in Java
  – Allow you to import (public) static fields and methods defined in classes

• For example

```java
import static java.lang.System.out;
import static java.lang.System.exit;
...
out.println("Hello World!");
exit(0);
```

– We can also say: `java.lang.System.*` to import all statics

Creating your own packages

Some advantages of using packages

– Makes your program more modular

– E.g., each package can represent a common theme

– Finer control of visibility (“package” visibility)
  • You can think of this as “class-level” encapsulation (more later)

– Avoids name collisions … package1.Todo ≠ package2.Todo
  • Java class names are “fully qualified”
  • That is, the package name is part of the class name
  • In this case, we’d use package1.Todo to refer to the first one
  • … and package2.Todo to refer to the second one
Putting your files into packages

*Can be tricky to get right at first …*

- To put your class in a package:

  1. **Choose a package name**
     - Usually the *reverse* of your domain (you can also make it up)
     - Domain: `edu.gonzaga` (or `edu.gonzaga.cs`)
     - The relative package name: `mypackage`
     - The full package: `edu.gonzaga.cs.mypackage`
     - *No spaces allowed!*

  2. **Put a package statement in your class**
     ```java
     package edu.gonzaga.mypackage;
     import static System.out.*;
     public class HelloWorld {
         public static void main(String[] args) {
             out.println("Hello World!");
         }
     }
     ```
     - Only one package statement per class file
     - Must be the *first statement* in your file
Putting your files into packages

3. Create a matching directory structure

Here myproject is the directory we run our program from

```
myproject
  edu
    gonzaga
      mypackage
        HelloWorld.java
```

4. Compiling and running

Assuming you are in myproject

```
$ javac edu/gonzaga/mypackage/HelloWorld.java
$ java edu.gonzaga.mypackage.HelloWorld
```

```
myproject
  edu
    gonzaga
      mypackage
        HelloWorld.java
```
Putting your files into packages

4. Or use a classes directory …

Can be more convenient to put the .class files into a classes dir

```sh
$ mkdir classes
$ javac -d classes edu/gonzaga/mypackage/*.java
$ cd classes
$ java edu.gonzaga.mypackage.HelloWorld
```

Using your custom packages

Consider a new class, in a different package:

```java
package edu.gonzaga.mypackage2;
public class MySayings {
    public static String getSaying1() {
        return "Hello World!";
    }
}
```

And our (modified) previous class:

```java
package edu.gonzaga.mypackage;
import edu.gonzaga.mypackage2.MySayings;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(MySayings.getSaying1());
    }
}
```
Putting your files into packages

Compiling now ...

$ mkdir classes
$ javac -d classes edu/gonzaga/mypackage/*.java
edu/gonzaga/mypackage2/*.java
$ cd classes
$ java edu.gonzaga.mypackage.HelloWorld

Package visibility

• By default if you have

```java
package org.example.widgets;
class Widget {
    ...
}
```

  – This class is only “visible” from within the package!

  – Widget cannot be used from outside of the package
    • It can be used by other classes in the package

  – “Package visibility” (i.e., no modifier) is the Java default
Jar files

- Your .class (and even .java) files can be placed into a “JAR” file (Java ARchive)

- A .jar file typically represents either
  - An application (and can be executable)
  - Or a library (for building applications)

- A .jar file is really just a fancy .zip file

Creating a Jar file

- Assume a HelloWorld.java file and no packages

- To create a jar we first compile
  
  javac HelloWorld.java

- And then we run the command:
  
  $ jar -cvf hello.jar HelloWorld.class

- To see the contents, run (table file):
  
  $ jar -tf hello.jar
  META-INF/
  META-INF/MANIFEST.MF ← This is a "manifest file"
  (A set of name-value pairs)
  HelloWorld.class
Creating a Jar file

• We can call the class file using the command:

        $ java -cp hello.jar HelloWorld

• You can also use -cp (classpath) to reuse the library in your own applications

Creating Executable Jar files

• To make your .jar file executable ...
  – Create a special manifest.txt file (to create MANIFEST.MF)
  – Your file should only contain the single line:
    Main-Class: HelloWorld
  – This tells java which class to call main from

• Then create the jar (passing in manifest.txt)

        $ jar -cvmf manifest.txt hello.jar HelloWorld.class

• And now we can run it
  – $ java -jar hello.jar
  – Hello World!
Creating Executable Jar files

• You can also “double-click” your .jar file to start your app
  – This is more exciting if your application has a GUI

• It is also possible to create .jar files if you use packages
  – E.g., from within your classes directory …
    $ jar –cvmf manifest.txt package.jar edu
  – where edu is the start of the package structure

Finish up remaining exercises …

• If you are done, try to create an executable .jar file
  – Either from a HelloWorld class
  – Or for a simple gui application (e.g., frame + label)

• Then, if you have time … place your .java file into a Java package and get it to compile and run