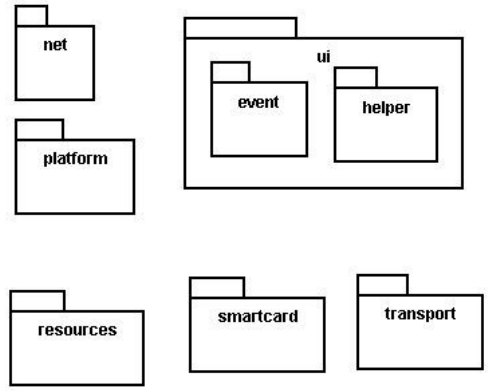
# CSE 331

Java Packages; JAR Archives

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

# Java packages

- **package**: A collection of related classes.
  - Can also "contain" sub-packages.
  - *Sub-packages* can have similar names, but are not actually contained inside.
    - `java.awt` does not contain `java.awt.event`



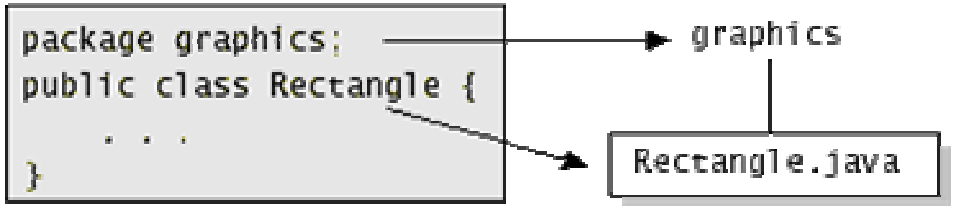

- Uses of Java packages:
  - group related classes together
  - as a *namespace* to avoid name collisions
  - provide a layer of access / protection
  - keep pieces of a project down to a manageable size

# Packages and directories

- package ←→ directory (folder)
- class ←→ file

- A class named `D` in package `a.b.c` should reside in this file:

`a/b/c/D.class`




  - (relative to the root of your project)

- The "root" directory of the package hierarchy is determined by your *class path* or the directory from which `java` was run.

# Classpath

- **class path**: The location(s) in which Java looks for class files.

- Can include:
  - the current "working directory" from which you ran javac / java
  - other folders
  - JAR archives
  - URLs
  - ...

- Can set class path manually when running java at command line:
  - `java -cp /home/stepp/libs:/foo/bar/jbl MyClass`

# A package declaration

```
package name;

public class name { ...
```

Example:

```
package pacman.model;

public class Ghost extends Sprite {
    ...
}
```

- File `Sprite.java` should go in folder `pacman/model`.

# Importing a package

```
import packageName.*;          // all classes
```

Example:

```
package pacman.gui;
import pacman.model.*;

public class PacManGui {
    ...
    Ghost blinky = new Ghost();
}
```

- `PacManGui` must import the model package in order to use it.

# Importing a class

```
import packageName.className;   // one class
```

Example:

```
package pacman.gui;
import pacman.model.Sprite;

public class PacManGui {
    Ghost blinky = new Ghost();
}
```

- Importing single classes has high precedence:
  - if you import `.*`, a same-named class in the current dir will override
  - if you import `.className`, it will not

# Static import

```
import static packageName.className.*;
```

Example:

```
import static java.lang.Math.*;

...
double angle = sin(PI / 2) + ln(E * E);
```

- Static import allows you to refer to the members of another class without writing that class's name.
- Should be used rarely and only with classes whose contents are entirely static "utility" code.

# Referring to packages

**packageName . className**

Example:

```
java.util.Scanner console =
    new java.util.Scanner(java.lang.System.in);
```

- You can use a type from any package without importing it if you write its full name.
- Sometimes this is useful to disambiguate similar names.
  - Example: `java.awt.List` and `java.util.List`
  - Or, explicitly import one of the classes.

# The default package

- Compilation units (files) that do not declare a package are put into a default, unnamed, package.

- Classes in the default package:
  - Cannot be imported
  - Cannot be used by classes in other packages

- Many editors discourage the use of the default package.

- Package java.lang is implicitly imported in all programs by default.
  - `import java.lang.*;`

# Package access

- Java provides the following access modifiers:
  - `public` : Visible to all other classes.
  - `private` : Visible only to the current class (and any nested types).
  - `protected` : Visible to the current class, any of its subclasses, and any other types within the same package.
  - default (package): Visible to the current class and any other types within the same package.

- To give a member default scope, do not write a modifier:

```
package pacman.model;
public class Sprite {
    int points;       // visible to pacman.model.*
    String name;      // visible to pacman.model.*
```

# Package exercise

- Add packages to the Rock-Paper-Scissors game.
  - Create a package for core "model" data.
  - Create a package for graphical "view" classes.

  - Any general utility code can go into a default package or into another named utility (util) package.
  - Add appropriate package and import statements so that the types can use each other properly.

# JAR Files (yousa likey!)

- **JAR**: **J**ava **AR**chive. A group of Java classes and supporting files combined into a single file compressed with ZIP format, and given .JAR extension.

- Advantages of JAR files:
  - compressed; quicker download
  - just one file; less mess
  - can be executable

- The closest you can get to having a .exe file for your Java application.

# Creating a JAR archive

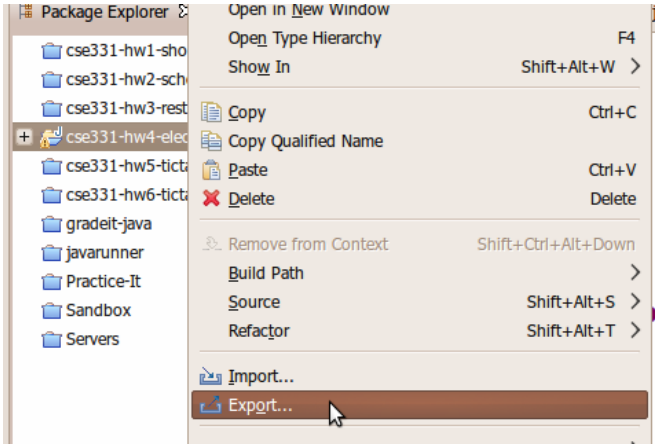
- from the command line:

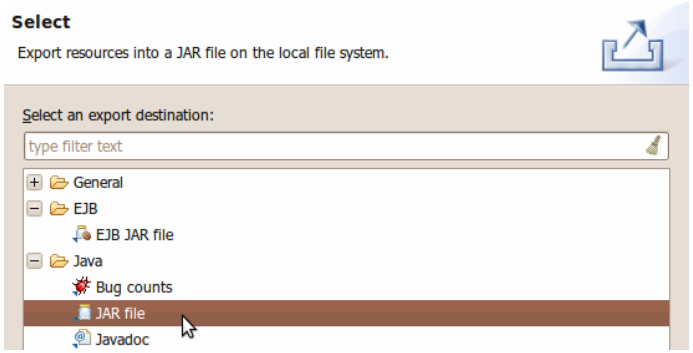
  `jar -cvf` **filename**`.jar` **files**

  - Example:

    `jar -cvf MyProgram.jar *.class *.gif *.jpg`

- some IDEs (e.g. Eclipse) can create JARs automatically
  - File → Export... → JAR file

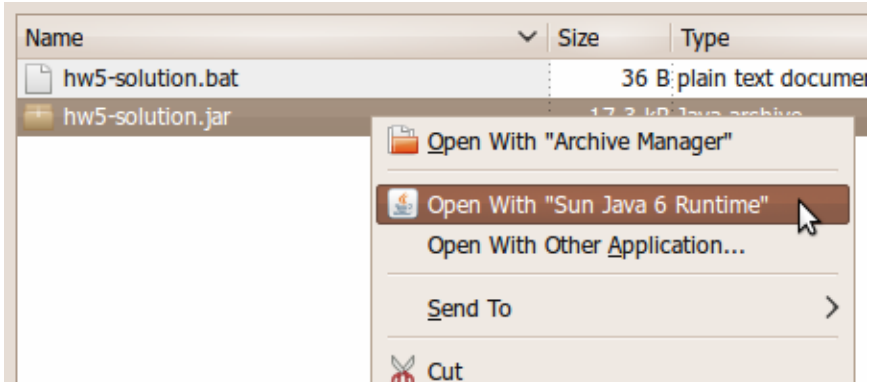


# Running a JAR

- Running a JAR from the command line:
  - `java -jar` **filename**`.jar`

- Most OSes can run JARs directly by double-clicking them:



# Making a runnable JAR

- **manifest file**: Used to create a JAR runnable as a program.

```
jar -cvmf manifestFile MyAppletJar.jar
          mypackage/*.class *.gif
```

*Contents of MANIFEST file:*

```
Main-Class: MainClassName
```

  - Eclipse will automatically generate and insert a proper manifest file into your JAR if you specify the main-class to use.

# Resources inside a JAR

- You can embed external resources inside your JAR:
  - images (GIF, JPG, PNG, etc.)
  - audio files (WAV, MP3)
  - input data files (TXT, DAT, etc.)
  - ...

- But code for opening files will look outside your JAR, not inside it.
  - `Scanner in = new Scanner(new File("data.txt"));  // fail`
  - `ImageIcon icon = new ImageIcon("pony.png");  // fail`
  - `Toolkit.getDefaultToolkit().getImage("cat.jpg");  // fail`

# Accessing JAR resources

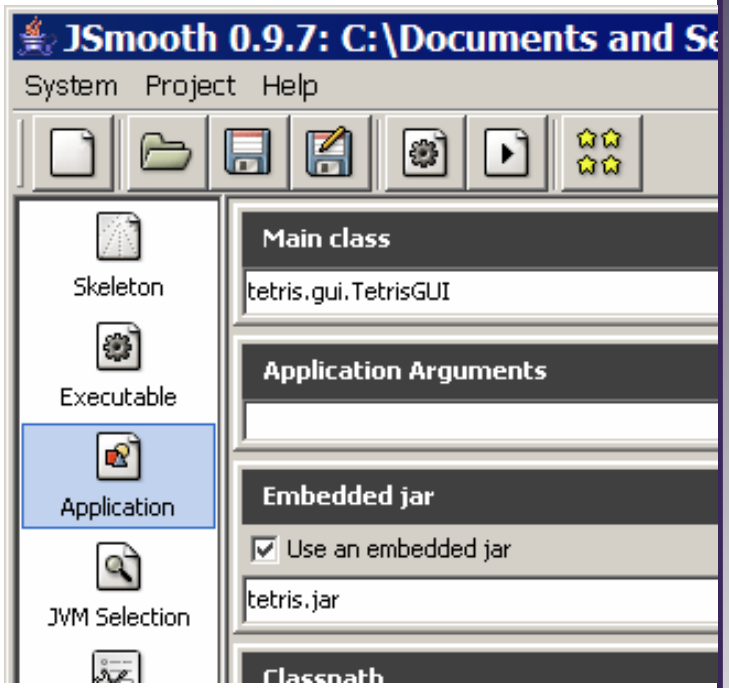
- Every class has an associated .class object with these methods:
  - `public URL getResource(String filename)`
  - `public InputStream getResourceAsStream(String name)`

- If a class named `Example` wants to load resources from within a JAR, its code to do so should be the following:
  - ```
    Scanner in = new Scanner(
        Example.class.getResourceAsStream("/data.txt"));
    ```
  - ```
    ImageIcon icon = new ImageIcon(
        Example.class.getResource("/pony.png"));
    ```
  - ```
    Toolkit.getDefaultToolkit().getImage(
        Example.class.getResource("/images/cat.jpg"));
    ```

  - (Some classes like Scanner read from streams; some like Toolkit read from URLs.)
  - NOTE the very important leading `/` character; without it, you will get a `null` result
    ```

# JAR to EXE (JSmooth)

- *JSmooth* is a free program that converts JARs into Windows EXE files.
  - http://jsmooth.sourceforge.net/
  - If the machine does not have Java installed, your EXE will help the user to download and install Java.
  - A bit of a hack; not generally needed.




- Using JSmooth:
  - choose Skeleton → Windowed Wrapper
  - name your .exe under Executable → Executable Binary
  - browse to your .jar under Application → Embedded JAR
  - select the main class under Application → Main class