



**CMSC 131**

# **Object-Oriented Programming I**

## **Packages**

**Dept of Computer Science  
University of Maryland College Park**

This material is based on material provided by Ben Bederson, Bonnie Dorr,  
Fawzi Emad, David Mount, Jan Plane

# Overview

## ► Packages

# Java Program Organization

## ► Program Organization:

- **Java program**: is composed of **1 or more Java source files**
  - **Source file**: can have **1 or more class and/or interface declarations**. (In our projects we have implemented one class/interface per file.)
  - **Public Class/Interface**: If a class/interface is declared **public** the source file must use the **same name**
  - Only **one public class/interface** is allowed **per source file**.
- Packages**: When a program is very large, its classes can be further organized **hierarchically** into packages

# Packages

- ▶ **Package**: a collection of **related** classes and/or interfaces
  - ▶ **Examples**: The Java API
    - javax.swing**: classes dealing with the development of GUIs
    - java.lang**: essential classes required by the Java language
    - java.text**: facilities for formatting text output
    - java.util**: classes for storing/accessing collections of objects
    - java.net**: for network communication
  - ▶ **Hierarchical**: Packages can be **divided** into subpackages
    - java.awt**: classes for basic GUI elements and graphics
      - java.awt.font**: classes and interface relating to fonts
      - java.awt.geom**: classes for defining 2-dimensional objects
- There is no limit to the nesting depth.

# Access to Package Members

## ► Review of Package Basics:

### Accessing Package Members:

**Fully qualified name:** E.g., **javax.swing.JOptionPane**

**Importing a single class:**

```
import javax.swing.JOptionPane;
```

...

```
JOptionPane.showMessageDialog( ... );
```

**Importing all the classes:**

```
import javax.swing.*;
```

...

```
JOptionPane.showMessageDialog( ... );
```

**Import semantics:** import does not “**insert**” the Java files (as C/C++ do with “include” files). Instead, it tells the compiler **where to look** to find classes that the program refers to

**Multiple import statements:** You can have as many as you like. They go **at the top of your .java** file (before any classes or interfaces)

**java.lang:** is **automatically imported** into every program

# Defining Your Own Package

- ▶ **Why packages?** Packages enable a programmer organize the code into smaller logically related units. A large program may consists of hundreds of classes. (Although we may not need to use them for projects in CMSC 131, but it is important to know how to create packages for when you will need them.)
- ▶ **Every class is part of some package:** Really? Why haven't we seen more of them?
- ▶ **Default package:** If you do not specify a package a class becomes part of the "**default package**"
- ▶ **What special privileges do packages provide?** Classes defined within the same package can access one another more easily (without the need for importing or fully qualified names)

# Defining Your Own Package

- ▶ **Defining a package:** Add a “**package**” statement to specify the package containing the classes of this file

```
package mypackage;
```

```
...
```

```
public class myClass { ... }    // myClass is part of mypackage
```

- ▶ This must be the **first statement** of your file. (Other than comments.)
- ▶ **Subpackages:** Packages organized into subpackages. This is specified using the notation “**main.subpackage**”. Example:

```
package mypackage.mysubpackage;
```

```
...
```

```
public class myClass2 { ... }    // myClass2 is part of mysubpackage  
                                // ... which is within mypackage
```

- ▶ **Packages in Eclipse:** **File**→**New**→**Package**. Enter the full name of the package (e.g. “**mypackage**” or “**mypackage.mysubpackage**”).
- ▶ **Without Eclipse:** Just insert this yourself (“**package mypackage;**”)

# Class Access and Packages

## ▶ **Class access within a package:**

- Classes within a package can refer to each other **without full qualification**
- If a class is **not** declared **public**, it can **only** be accessed by other classes **within the package**

## ▶ **Class access across packages:**

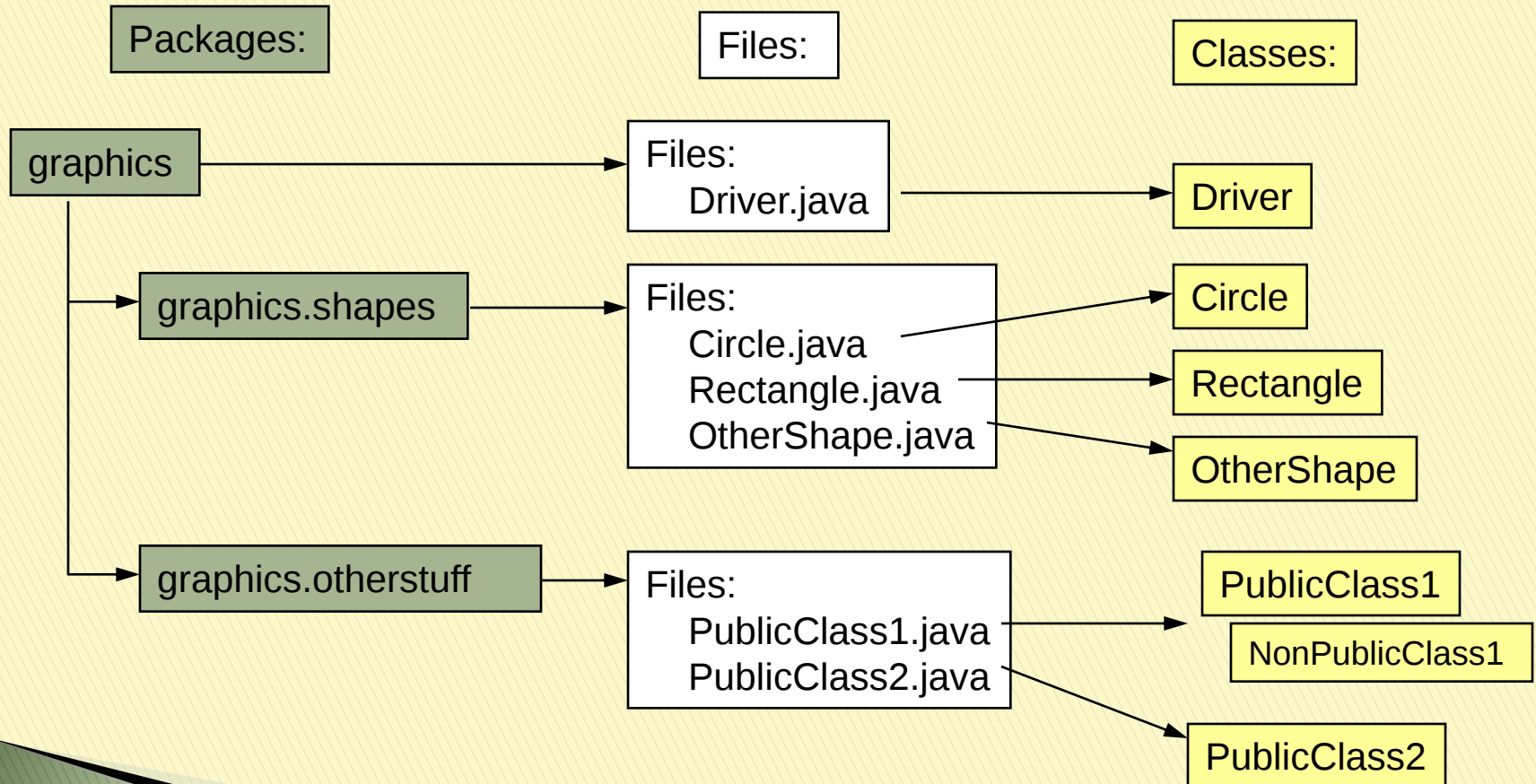
- A **public class** can be accessed from **other packages**
- When this is done, either its name is **fully qualified** or is **imported**
- We can view **public classes of a package** as the “**interface**” of the package with the outside world. (This is analogous to public methods of a class forming its interface.)

## ▶ **Subpackages are not automatically imported:**

- When you import a package (e.g. **import java.awt.\*** ) it **does not** import the subpackages (e.g. **java.awt.font** must be **explicitly** imported)

# Example

- To illustrate these points, let's consider an example of a simple hierarchical package, which we will create.



# Example: graphics.shapes package

```
package graphics.shapes;  
public class Circle {  
    private double radius;  
    public String toString( ) { return "I'm a circle"; }  
}
```

File: Circle.java

```
package graphics.shapes;  
public class Rectangle {  
    private double height, width;  
    public String toString( ) { return "I'm a rectangle"; }  
}
```

File: Rectangle.java

```
package graphics.shapes;  
public class OtherShape {  
    private Circle c;  
    private Rectangle r;  
}
```

File: OtherShape.java

Note: Classes of this package can be accessed without the need for import or using graphics.shapes.Circle

# Example: graphics.otherstuff package

File: PublicClass1.java

```
package graphics.otherstuff;
```

```
public class PublicClass1 {  
    public String toString() {  
        return "This is a PublicClass: " + NonPublicClass1.message();  
    }  
}  
class NonPublicClass1 {  
    static public String message() { return "I'm a nonpublic class"; }  
}
```

Public class:  
Accessible everywhere

Nonpublic class:  
Only accessible in this package

```
package graphics.otherstuff;
```

```
public class PublicClass2 {  
    private Driver d;  
    private Circle c1;  
    private graphics.shapes.Circle c2;  
  
    public String toString() {  
        return "This is a PublicClass2: " + NonPublicClass1.message();  
    }  
}
```

File: PublicClass2.java

Compiler error! We have no direct access to classes outside this package. Import or use qualified name.

Okay, using the fully qualified name.

Okay: Can access a nonpublic classes within this package.

# Example: graphics.package

```
package graphics;  
import graphics.shapes.Circle;
```

Note: importing Circle (but nothing else)

File: Driver.java

```
public class Driver {  
    public static void main( String[ ] args ) {  
        testShapes( );  
        testOtherStuff( );  
    }  
}
```

```
public static void testShapes( ) {  
    Circle c = new Circle( );  
    System.out.println( c.toString( ) );  
    Rectangle r = new Rectangle( );  
}
```

Okay: Circle is accessible.

Compiler error: Cannot access Rectangle without import or qualified name.

```
public static void testOtherStuff( ) {  
    PublicClass1 x = new PublicClass1( );  
    graphics.otherstuff.PublicClass1 y = new graphics.otherstuff.PublicClass1( );  
    System.out.println( y );  
    graphics.otherstuff.NonPublicClass1 z;  
}
```

Compiler error: Cannot access PublicClass1 without import or qualified name.

Compiler error: Non-public classes are never accessible outside the class.

# File Structure

Java organizes the package files using your system's directory structure.

**graphics:**

**Driver.class Driver.java**

**otherstuff/**

**shapes/**

**graphics/otherstuff:**

**NonPublicClass1.class**

**PublicClass1.java PublicClass2.java**

**PublicClass1.class PublicClass2.class**

**graphics/shapes:**

**Circle.class OtherShape.class Rectangle.class**

**Circle.java OtherShape.java Rectangle.java**

When you create new packages in Eclipse, this is done automatically.

Notice that nonpublic classes generate their own .class file, even though there is no .java file.

# Packages and .jar Files

- ▶ Packages are convenient, but large directories are not.
- ▶ **Jar File**: Java allows you to **bundle up an entire directory** of files into a single file, called a **jar file**.
- ▶ **Creating a jar file**:
  - **In Eclipse**: In the Package Explorer window, right click on the project and select “**Export** → **JAR file**”
  - **On Unix**:  
**jar -cvf myJarFile.jar ... (list the file names and/or directories)**  
(c = create; t = list names; x = extract; v = verbose; f = jar file)
- ▶ **Examples**:
  - cmssc131PictureLib.jar**: A jar file we created for a picture library
  - C:\...\Java\jre1.5.0\_14\lib\rt.jar**: This is a large (33+Mbyte) file with all the class files from the Java runtime library

# Packages and Classpath

- ▶ You may have many different packages on your system (e.g. the Java runtime library, cmssc131PictureLib, etc.) in many different locations. **How does Java know where to look** for them?
- ▶ **ClassPath**: is a system **environment variable** that gives a list of directories and jar files where Java should look up its classes.
- ▶ **Windows Example**: Suppose we want to use
  - **graphics package**: stored in directory C:\MyJavaPackages\graphics.
  - **cmssc131PictureLib.jar**: stored in C:\MyJars\cmssc131PictureLib.jar
  - Classes compiled in the **current working directory**: The current directory is denoted by "." (period) on most systems.
- ▶ **C:\>set**  
**CLASSPATH=.;C:\MyJavaPackages;C:\MyJars\cmssc131PictureLib.jar**

Always include "."

The list is separated by semicolons ";"

jar files must be listed explicitly

# Packages and Classpath

- ▶ **In Eclipse:** The ClassPath is already set with important default directories (e.g. the Java runtime library). To modify the ClassPath:
  - In the **Package Explorer**, right click on the project name
  - Select “**Properties** → **Java Build Path** → **Libraries**”
  - **To add Jars:** Select: “**Add External JARs...**” and browse for the file name
  - **To add a directory to the ClassPath:** Select:

**Add Variable... → Configure Variables → New**

and add the name of the new directory