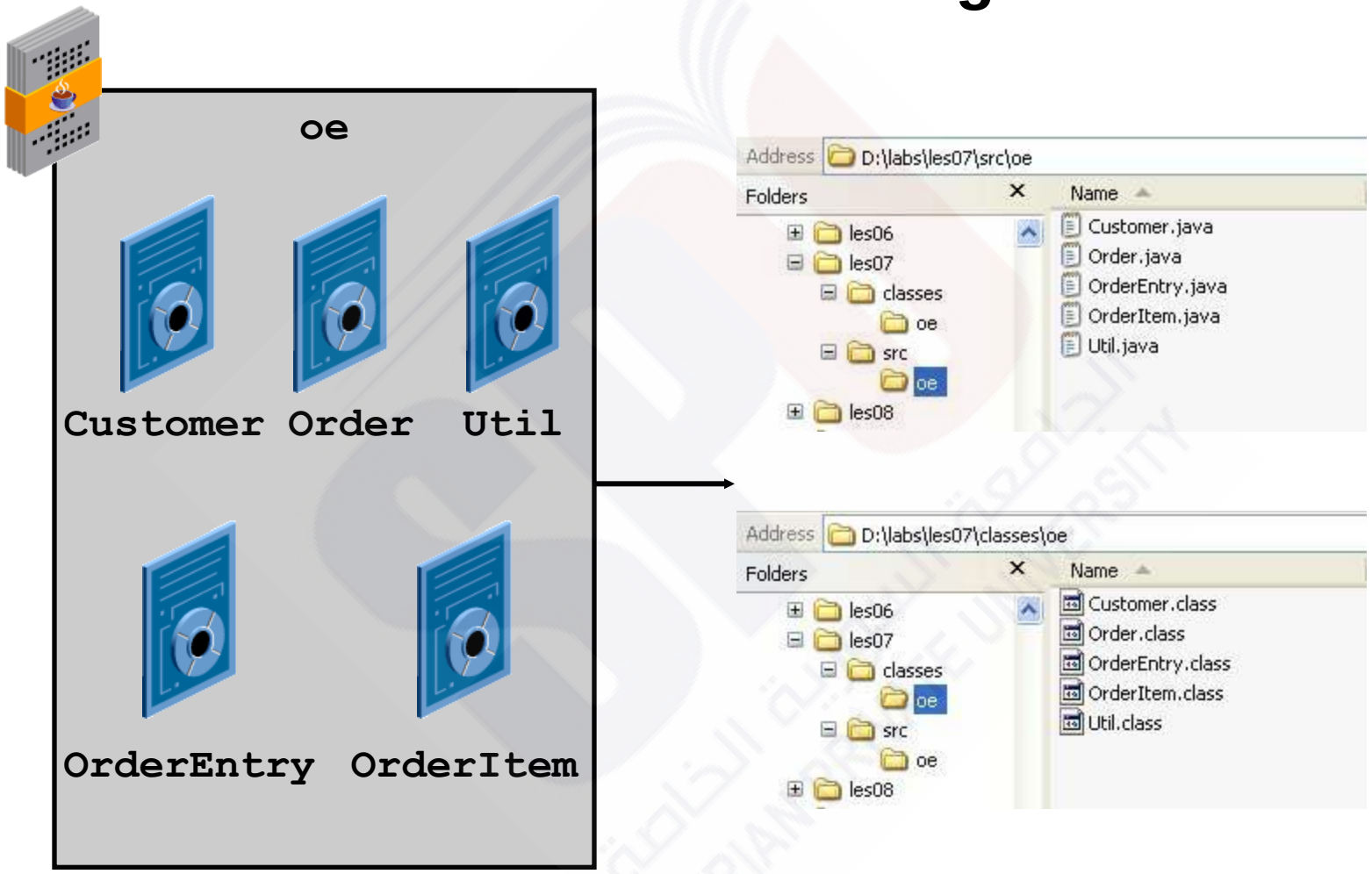
What Are Java Packages?



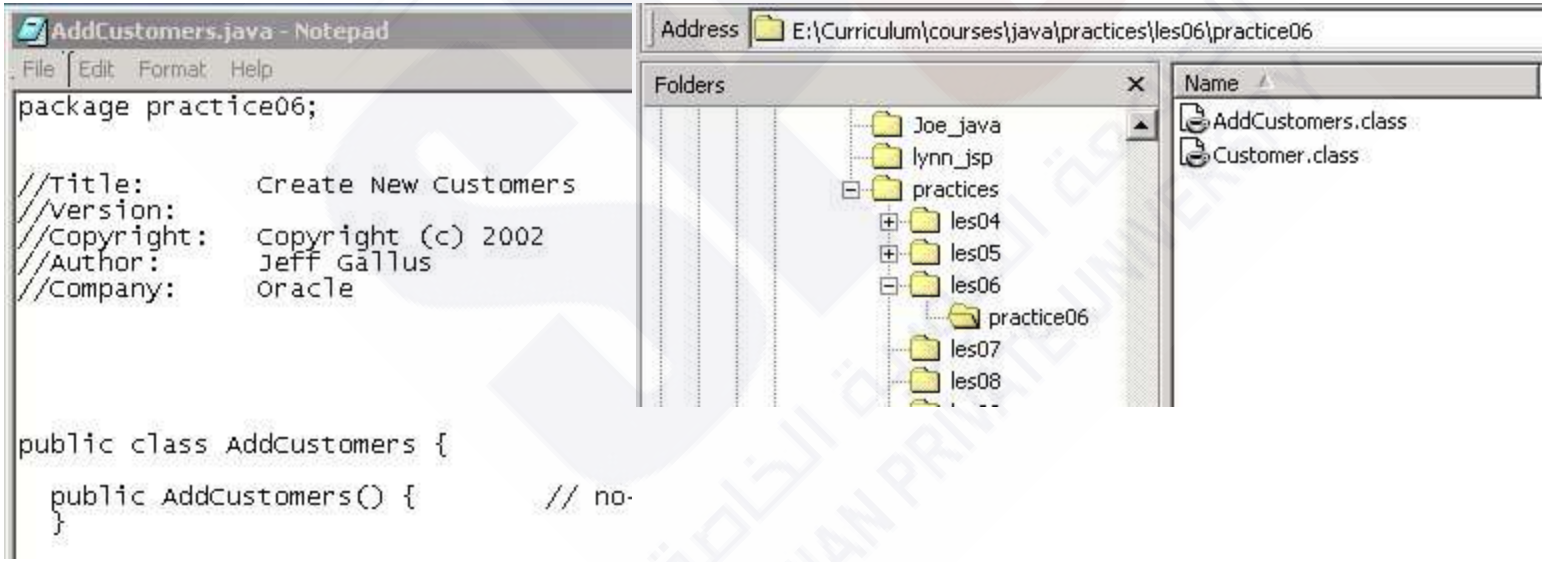
Grouping Classes in a Package

- Include the `package` keyword followed by the package name at the top of the Java source file. Use the dot notation to show the package path.
- If you omit the `package` keyword, then the compiler places the class in a default "unnamed" package.
- Use the `–d` flag with the `javac` compiler to create the package tree structure relative to the specified directory.
- Running a `main()` method in a packaged class requires:
  - That the `CLASSPATH` contains the directory having the root name of the package tree
  - That the class name must be qualified by its package name

Setting the `CLASSPATH` with Packages

The `CLASSPATH` includes the directory containing the top level of the package tree:

Package name **`.class` location**



CLASSPATH

```
C:\>set CLASSPATH=E:\Curriculum\courses\java\les06
```

Hiding the Implementation

package: the library unit

Example:

```
package mypackage;
    public class MyClass {
        // . . .
```

If someone wants to use **MyClass,** they must use the **import** keyword.

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

The alternative is to give the fully qualified name:

```
mypackage.MyClass m = new mypackage.MyClass();
```

Hiding the Implementation

package: the library unit

Example:

```
package com.bruceeckel.simple;
public class List {
  public List() {
    System.out.println("--");
  }
}
```

The first portion of the path taken from CLASSPATH environment variable:

CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT

The files (*.java, *.class) are placed in the subdirectory on system:

C:\DOC\JavaT\com\bruceeckel\simple

When using JAR files, you must put the name of the JAR file in the classpath.

CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar

Hiding the Implementation

package: the library unit

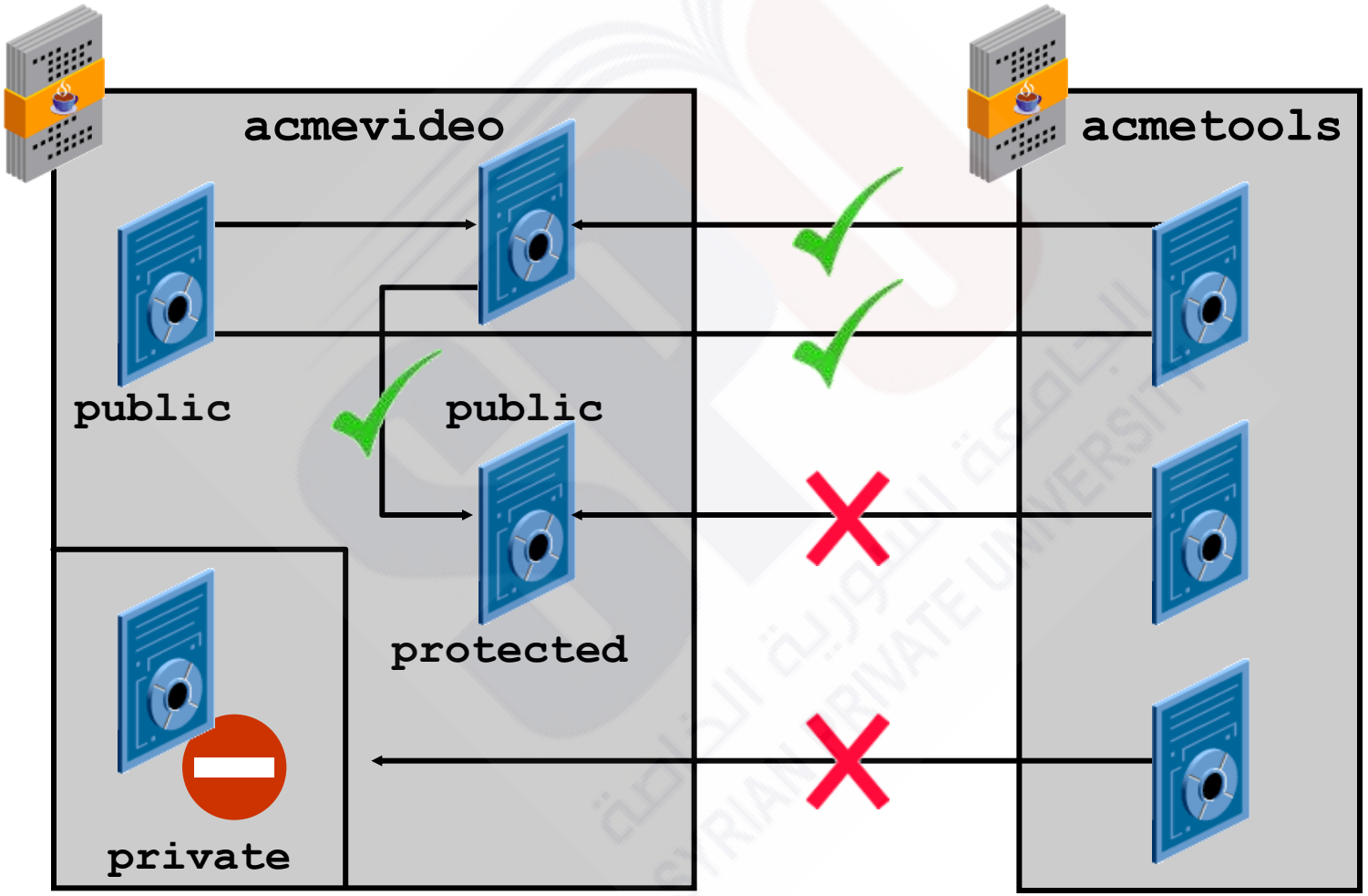
Collisions

What happens if two libraries are imported via '*' and they include the same names? For example, suppose a program does this:

```
import com.bruceeckel.simple.*;
import java.util.*;
Vector v = new Vector();
```

```
java.util.Vector v = new java.util.Vector();
```

Access Modifiers



Hiding the Implementation

Java access specifiers

Package access ("friendly")

Hiding the Implementation

Java access specifiers

public: interface access

```
package c05.dessert;
public class Cookie {
  public Cookie () {
      System.out.println(“---");
  }
  void bite () {
      System.out.println("bite");
  }
}
```

file1

```
import c05.dessert.*;
public class Dinner {
  public Dinner() {
    System.out.println("---");
  }
  public static void main (String[] args) {
    Cookie x = new Cookie();
      //! x.bite(); // Can't access
  }
}
```

file2

Hiding the Implementation

Java access specifiers

The default package

```
class Cake {
  public static void main (String[] args) {
    yyy x = new yyy();
     x.f();
  }
}
```

```
class yyy {
  void f() {
    System.out.println("--");
  }
}
```

2 files in the same directory

(you must have '.' in your CLASSPATH in order for the files to compile.)

Java treats files like this as implicitly part of the "default package" for that directory

Hiding the Implementation

Java access specifiers

private: you can't touch that!

```
class A {
  private A () {}
  static A makeA () {
     return new A();
  }
}
```

```
public class IceCream {
  public static void main(String[] args) {
     //! A x = new A();
    A x = A.makeA ();
  }
}
```

you cannot create a A object via its constructor; instead, you must call the makeA ( ) method to do it for you.

Hiding the Implementation

Java access specifiers

protected: inheritance access

Hiding the Implementation

Interface and implementation

Class access

Using Strings, String Buffer, Wrapper, and Text-Formatting Classes

Objectives

- Create strings in Java
- Use the conversion methods that are provided by the predefined wrapper classes
- Use the `StringBuffer` class for manipulating character data
- Introduce the `DateFormat`, `DecimalFormat`, and `MessageFormat` classes
- Examine standard output and serialization

What Is a `String`?

- **`String` is a class.**
- **A `String` object holds a sequence of characters.**
- **`String` objects are read-only (immutable); their values cannot be changed after creation.**
- **The `String` class represents all strings in Java.**

Modify the contents of string, using StringBuffer class.

Creating a `String`

- **Assign a double-quoted constant to a `String` variable:**

```
String category = "Action";
```

- **Concatenate other strings:**

```
String empName = firstName + " " + lastName;
```

- **Use a constructor:**

```
String empName = new String("Bob Smith");
```

String class is part of java.lang package, which is automatically imported into all Java classes.

String class provides several constructors; the some of the more useful ones:

- **String ( ) create an empty string, with value " ".**
- **String (String str) create a copy of the specified String object,str.**
- **String (char [ ] arr) create a string from the characters in a character array.**

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Find a list of constructor in J2SE documentation

The only overloaded operator in java.

Concatenating Strings

- Use the + operator to concatenate strings.

```
System.out.println("Name = " + empName);
```

- You can concatenate primitives and strings.

```
int age = getAge();
System.out.println("Age = " + age);
```

- The `String` class has a `concat()` instance method that can be used to concatenate strings.

String name = " koko lolo";

String name = " koko ".concat ("lolo");

Performing Operations on Strings

- **Find the length of a string:**

```
int length();
```

```
String str = "Comedy";
int len = str.length();
```

- **Find the character at a specific index:**

```
char charAt(int index);
```

```
String str = "Comedy";
char c = str.charAt(1);
```

- **Return a substring of a string:** c is set to o

```
String substring
       (int beginIndex,
       int endIndex);
```

```
String str = "Comedy";
String sub =
     str.substring(2,4);
```

sub is set to me

Performing More Operations on Strings

- **Convert to uppercase or lowercase:**

```
String toUpperCase();
String toLowerCase();
```

```
String caps =
      str.toUpperCase();
```

- **Trim whitespace:** Space removed from both ends

```
String trim();
```

```
String nospaces =
      str.trim();
```

- **Find the index of a substring:**

```
int indexOf (String str);
int lastIndexOf
       (String str);
```

```
int index =
      str.indexOf("me");
```

Comparing `String` Objects

- Use `equals()` if you want case to count:

```
String passwd = connection.getPassword();
if (passwd.equals("fgHPUw"))… // Case is important
```

- Use `equalsIgnoreCase()` if you want to ignore case:

```
String cat = getCategory();
if (cat.equalsIgnoreCase("Drama"))…
                    // We just want the word to match
```

- Do not use ==.

  = = compare reference.

Producing Strings from Other Objects

- **Use the `Object.toString()` method.**

If your class has a toString method, then you can include your object in concatenated string expressions, and you can print your object as if it were a string.

The **toString ()** method is **invoked** automatically **whenever you use any object reference** in a string **concatenation expression**, or pass it to **System.out.println**.

Producing Strings from Other Objects

- **Use the `Object.toString()` method.**

If a class does not provide its own toString ( ) method, then it inherits one from the Object class.

The string that is produced from Object.toString ( ) consists of:

the name of the class of which the object is an instance and a hexadecimal number representing a hash code.

```
public String toString ( )  {
    return   getClass ( ).getNmae ( ) + '@' +
                Integer.toHexString (hashCode( ));
}
```

```
class WaterSource {
  private   String s;
  WaterSource() {    System.out.println("WaterSource()");   s = new String("Constructed");   }
  public String toString() { return s; }      }

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  private    WaterSource    source;       private int i;       private float f;      private Integer z;
  public String toString() {
    valve1 = "lolo";           z = new Integer (999);          source = new WaterSource ();
    return
      "valve1 = " + valve1 + "\n" +
      "valve2 = " + valve2 + "\n" +
      "valve3 = " + valve3 + "\n" +
      "valve4 = " + valve4 + "\n" +
      "i = " + i + "\n" +
      "f = " + f + "\n" +
      "source = " + source + "\n" +
      "z = " + z;     }
  public static void main(String[] args) {
    SprinklerSystem sprinklers = new SprinklerSystem();
    System.out.println(sprinklers);
  }
}
```

WaterSource()
valve1 = lolo
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = Constructed
z = 999

Producing Strings from Other Objects

- Use the `Object.toString()` method.
- Your class can override `toString()`.

```
public Class Movie {…
      public String toString () {
            return name + " (" + Year + ")";
      }…
```

- `System.out.println()` automatically calls an object's `toString()` method if a reference is passed to it.

```
Movie mov = new Movie(…);
System.out.println(mov);
```

Producing Strings from Primitives

- Use `String.valueOf()`: { Static method }

```
String seven = String.valueOf(7);
String onePoint0 = String.valueOf(1.0f);
```

- There is a version of `System.out.println()` for each primitive type:

```
int count;
…
System.out.println(count);
```

Producing Primitives from Strings

- Use the primitive wrapper classes.
- There is one wrapper class for each primitive type:
  - `Integer` wraps the `int` type.
  - `Float` wraps the `float` type.
  - `Character` wraps the `char` type.
  - `Boolean` wraps the `boolean` type.
  - And so on…
- Wrapper classes provide methods to convert a `String` to a primitive, and primitives to a `String`.

static

Changing the Contents of a String

- Use the `StringBuffer` class for modifiable strings of characters:

```
public String reverseIt(String s) {
  StringBuffer sb = new StringBuffer();
  for (int i = s.length() - 1; i >= 0; i--)
    sb.append(s.charAt(i));
  return sb.toString();
}
```

- Use `StringBuffer` if you need to keep adding characters to a string.

Note: `StringBuffer` has a `reverse()` method.

Formatting Classes

The `java.text` package contains:

- An abstract class called `Format` with the `format()` method shown in the following example:

```
public abstract class Format … {
  public final String format(Object obj){
  //Formats an object and produces a string.
  }
  …
}
```

- Classes that format locale-sensitive information such as dates, numbers, and messages
  - `DateFormat`, `NumberFormat`, and `MessageFormat`

Using the `SimpleDateFormat` Class

The `SimpleDateFormat`:

- Is a concrete class for formatting and parsing dates in a locale-sensitive manner
- Allows you to start by choosing any user-defined patterns for date–time formatting
- Uses time-pattern string to display the date:
  - y year 1996
  - M month in year July or 07
  - m minute in hour 30

Using the `MessageFormat` Class

The `MessageFormat`:

- Is a concrete class for constructing language neutral messages, displayed for end users
- Takes a set of objects, formats them, and then inserts the formatted strings into the pattern at the appropriate places
- Differs from other `Format` classes, in that you create a `MessageFormat` object
- Is typically set dynamically at run time

Using `DecimalFormat`

The `DecimalFormat`:

- Is a concrete subclass of `NumberFormat` for formatting decimal numbers
- Allows for a variety of parameters and for localization to Western, Arabic, or Indic numbers
- Uses standard number notation in format

```
public DecimalFormat(String pattern);
```