# CT326 Programming III

LECTURE 9

PACKAGES, VARARGS,
AND FORMATTED INPUT

DR ADRIAN CLEAR
SCHOOL OF COMPUTER SCIENCE

Ollscoil na Gaillimhe
University of Galway

# Objectives for today

- Understand java packages and how to create them
- Illustrate
  - for-each loops
  - Formatted input
  - varargs

# Creating and Using Packages

- To make types easier to find and to use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.
- Definition: A package is a collection of related types providing access protection and namespace management. Note that types refers to classes, interfaces, enums and annotations.
- The types that are part of the Java platform are members of various packages that bundle classes by function:
  - fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on.
- You can put your types in packages, too.

# Package example

- Suppose that you write a group of classes that represent a collection of graphic objects, such as circles, rectangles, lines, and points. You also write an interface, Draggable, that classes implement if they can be dragged with the mouse by the user:

```
//in the Graphic.java file
public abstract class Graphic {
    . . .
}

//in the Circle.java file
public class Circle extends Graphic implements Draggable {
    . . .
}
```

# Package example

```
//in the Rectangle.java file
public class Rectangle extends Graphic implements Draggable {
    . . .
}

//in the Draggable.java file
public interface Draggable {
    . . .
}
```

# Package example

- You should bundle these classes and the interface in a package for several reasons:
  - You and other programmers can easily determine that these types are related.
  - You and other programmers know where to find types that provide graphics-related functions.
  - The names of your types won't conflict with types names in other packages, because the package creates a new namespace.
  - You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

# Creating a package

- To create a package, you put a type (class, interface, enum or annotation) in it. To do this, you put a package statement at the top of the source file in which the type is defined.
- For example, the following code appears in the source file Circle.java and puts the Circle class in the graphics package:

```
package graphics;

public class Circle extends Graphic implements Draggable {
    . . .
}
```

# Creating a package

- You must include a package statement at the top of every source file that defines a class or an interface that is to be a member of the graphics package.
- So, you would also include the statement in Rectangle.java and so on:

```
package graphics;

public class Rectangle extends Graphic implements Draggable {
    . . .
}
```

# Package scope

- The scope of the package statement is the entire source file, so all classes, interfaces, enums and annotations defined in Circle.java and Rectangle.java are also members of the graphics package.
- If you put multiple classes in a single source file, only one may be public, and it must share the name of the source file's base name.
  - Only public package members are accessible from outside the package.
- If you do not use a package statement, your type ends up in the default package, which is a package that has no name.

# Naming a package

- With programmers all over the world writing classes, interfaces, enums and annotations using the Java programming language, it is likely that two programmers will use the same name for two different classes.
- In fact, the previous example does just that: It defines a `Rectangle` class when there is already a `Rectangle` class in the `java.awt` package. Yet the compiler allows both classes to have the same name. Why?
- Because they are in different packages, and the fully qualified name of each class includes the package name.
  - That is, the fully qualified name of the `Rectangle` class in the `graphics` package is `graphics.Rectangle`, and the fully qualified name of the Rectangle class in the `java.awt` package is `java.awt.Rectangle`.

# Naming a package

- This generally works just fine unless two independent programmers use the same name for their packages.
  - What prevents this problem? Convention.
  - By Convention:  Companies use their reversed Internet domain name in their package names, like this: `com.company.package`.
  - Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name, for example, `com.company.region.package`.

# Static Import

- The static import feature, implemented as "import static", enables you to refer to static constants from a class without needing to inherit from it.
- Instead of `BorderLayout.CENTER` each time we add a component, we can simply refer to `CENTER`.

```
import static java.awt.BorderLayout.*;
 getContentPane().add(new JPanel(), CENTER);
```

# For-Each loop

- The Iterator class is used heavily by the Collections API.
  - It provides the mechanism to navigate sequentially through a Collection.
- The For-Each loop can replace the iterator when simply traversing through a Collection as follows.
  - The compiler generates the looping code necessary and with generic types no additional casting is required.

# For-Each loop

- Before (Iterator)

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Iterator i = list.iterator(); i.hasNext();)
  {
      Integer value=(Integer)i.next();
  }
```

- After (for-each)

```
ArrayList<Integer> list = new ArrayList<Integer>();
  for (Integer i : list)
  { ... }
```

# Exercise

- Modify `ArraySequences` from last week so that it is in a relevant package
- Make the iterator more general by allowing a start point and increment value to be specified on creation
- Add a `printOdd` method that uses the iterator for loop to print all of the odd numbers
- Add a `printAll` method that uses the for-each loop to print all of the numbers

# Formatted Output

- Developers have the option of using printf-type functionality to generate formatted output.
- Most of the common C printf formatters are available, and in addition some Java classes like `Date` and `BigInteger` also have formatting rules.
  - See the `java.util.Formatter` class for more information. Although the standard UNIX newline '\n' character is accepted, for cross-platform support of newlines the Java %n is recommended.

```
System.out.printf("name count%n");
System.out.printf("%s %5d%n", user,total);
```

# Formatted Input

- The scanner API provides basic input functionality for reading data from the system console or any data stream.
- If you need to process more complex input, then there are also pattern-matching algorithms, available from the `java.util.Formatter` class.

```
Scanner s= new Scanner(System.in);
String param= s.next();
int value=s.nextInt();
s.close();
```

# Varargs

- It requires the simple ... notation for the method that accepts the argument list and is used to implement the flexible number of arguments required for example in the printf() function.

```
void argtest(Object ... args) {
 for (int i=0;i <args.length; i++) {

    }
}

argtest("test", "data");
```

# Exercise

- Write a varargs method that takes any number of string parameters and prints using formatted output the total number of strings entered and the total number of letters across all of them.
- Sample output:

  *You entered 5 words with a total number of 17 letters.*



# Next time…

- Demo uses of packages and external jars
- Joda Money API