JAVA PROGRAMMING

Chapter 4

Java Language Fundamentals

Objectives

- Develop code that declares classes, interfaces, and enums, and includes the appropriate use of package and import statements
- Explain the effect of modifiers
- Given an example of a class and a command-line, determine the expected runtime behavior
- Determine the effect upon object references and primitive values when they are passed into methods

Objectives (cont.)

- Recognize the point at which an object becomes eligible for garbage collection, and determine what is and is not guaranteed by the garbage collection system.
- Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class

Contents

Organizing Your Java Application

- A Java application is composed of a set of files generally distributed in a directory structure
  - Source code: `.java`
  - Class file: `.class`
- The compiler searches for a class file when it encounters a reference to a class in a `.java` file
- The interpreter, during runtime, searches the `.class` files
- Both the compiler and the interpreter search for the `.class` files in the list of directories listed in the `classpath` variable

Entering Through the Main Gate

- When executing a Java application, the JVM loads the class, and invokes the main() method of this class
- The method main() must be declared `public`, `static`, and `void`
- A source file may have one or more classes. Only one class (matching the file name) at most may be declared `public`

Listing 4-1. *TestArgs.java*

```
1. public class TestArgs {
2.  public static void main (String [] args) {
3.   System.out.println("Length of arguments array: " + args.length);
4.   System.out.println("The first argument: " + args[0]);
5.   System.out.println("The second argument: " + args[1]);
6.  }
7. }
```

```
javac TestArgs.java
```

```
java  TestArgs  Ruth Srilatha
```

```
Length of arguments array: 2
The first argument: Ruth
The second argument: Srilatha
```

Important points

- A `.java` file name should match the name of a class in the file. If one of those classes is declared `public`, then the name of the `.java` file must match the name of the public class
- There can be only one `public` class at maximum in a source file
- The compiler generates a file with extension `.class` corresponding to each class in the source file that is compiled
- The name of the `.class` file matches the name of the corresponding class

What Is in a Name?

- You can bundle related classes and interfaces into a group called a package in a directory whose name matches the name of the package

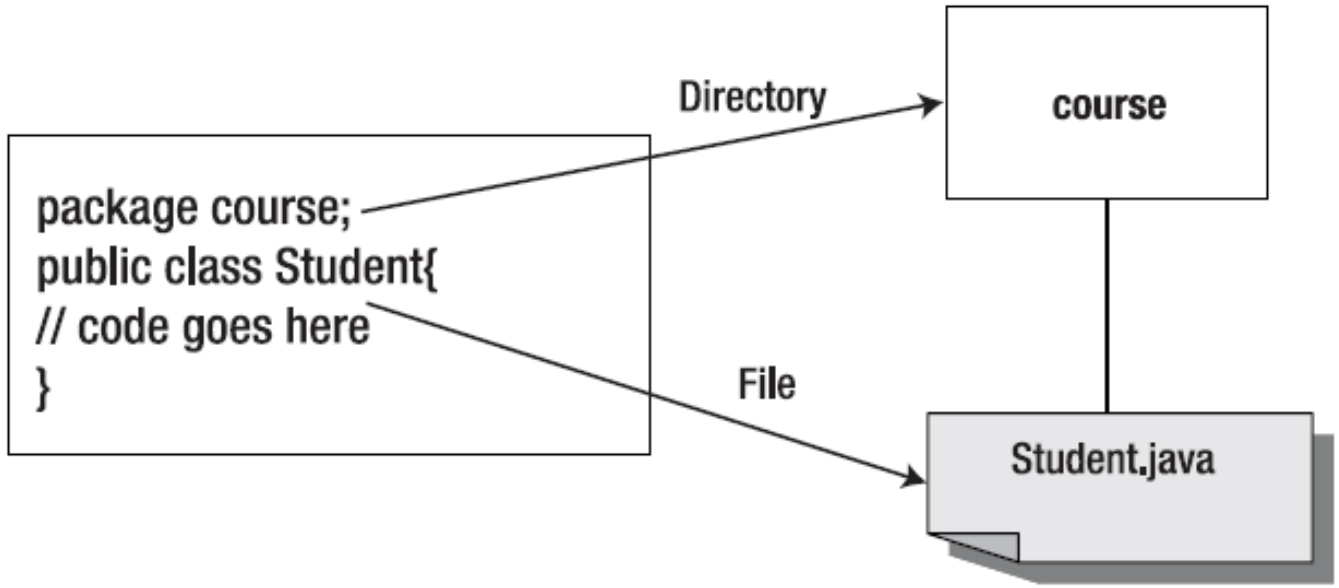


Figure 4-1. *The Student class is specified to be in the package course.*

What Is in a Name? (cont.)

```
course.Student student1 = new course.Student();
```

- The qualified name for the class is `course.Student`, and the path name to it is `course/Student.java`
- to import the package:

```
import course.Student;
```

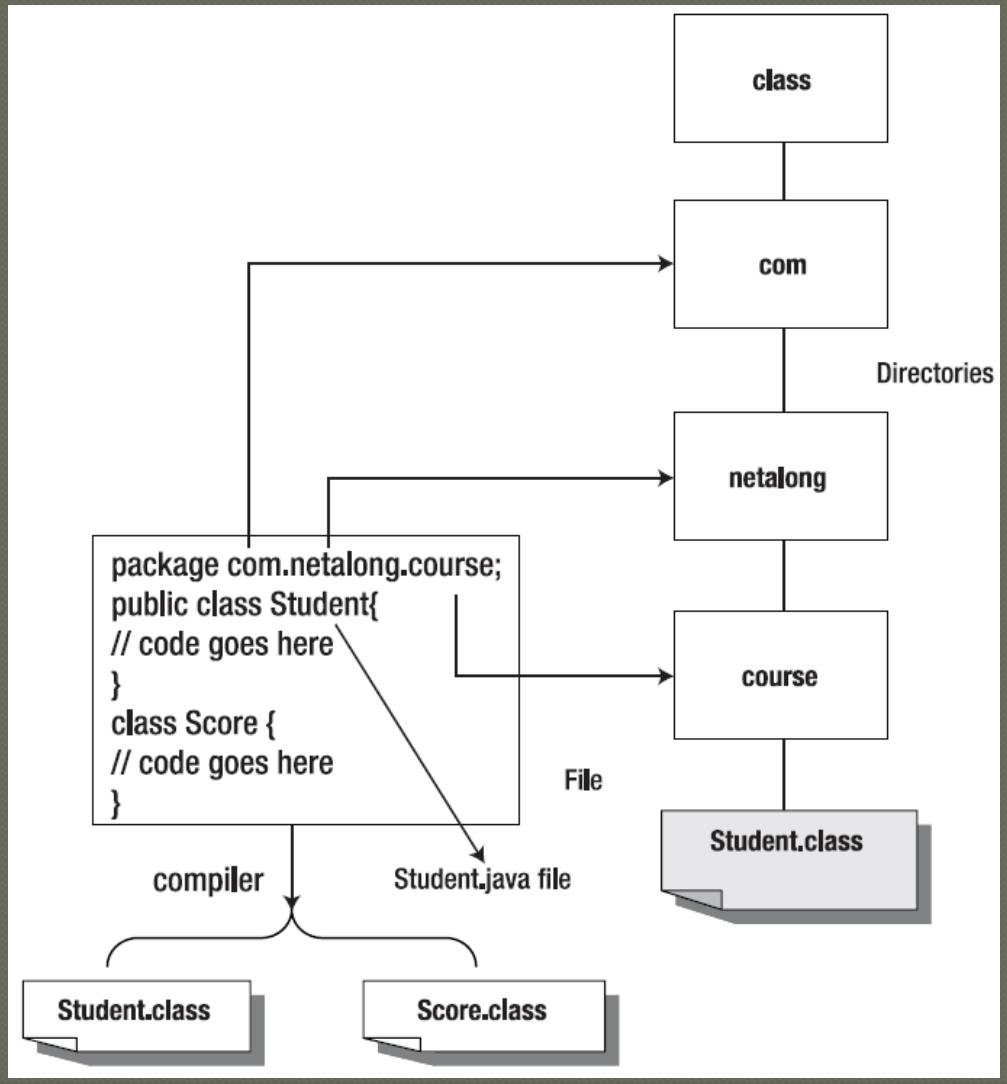
- You can import all the classes in the package:

```
import course.*;
```

Advantages

- It makes it easier to find and use classes.
- It avoids naming conflicts. Two classes with the same name existing in two different packages do not have a name conflict, as long as they are referenced by their fully qualified name.
- It provides access control.

A Package Name and the Corresponding Directory Structure

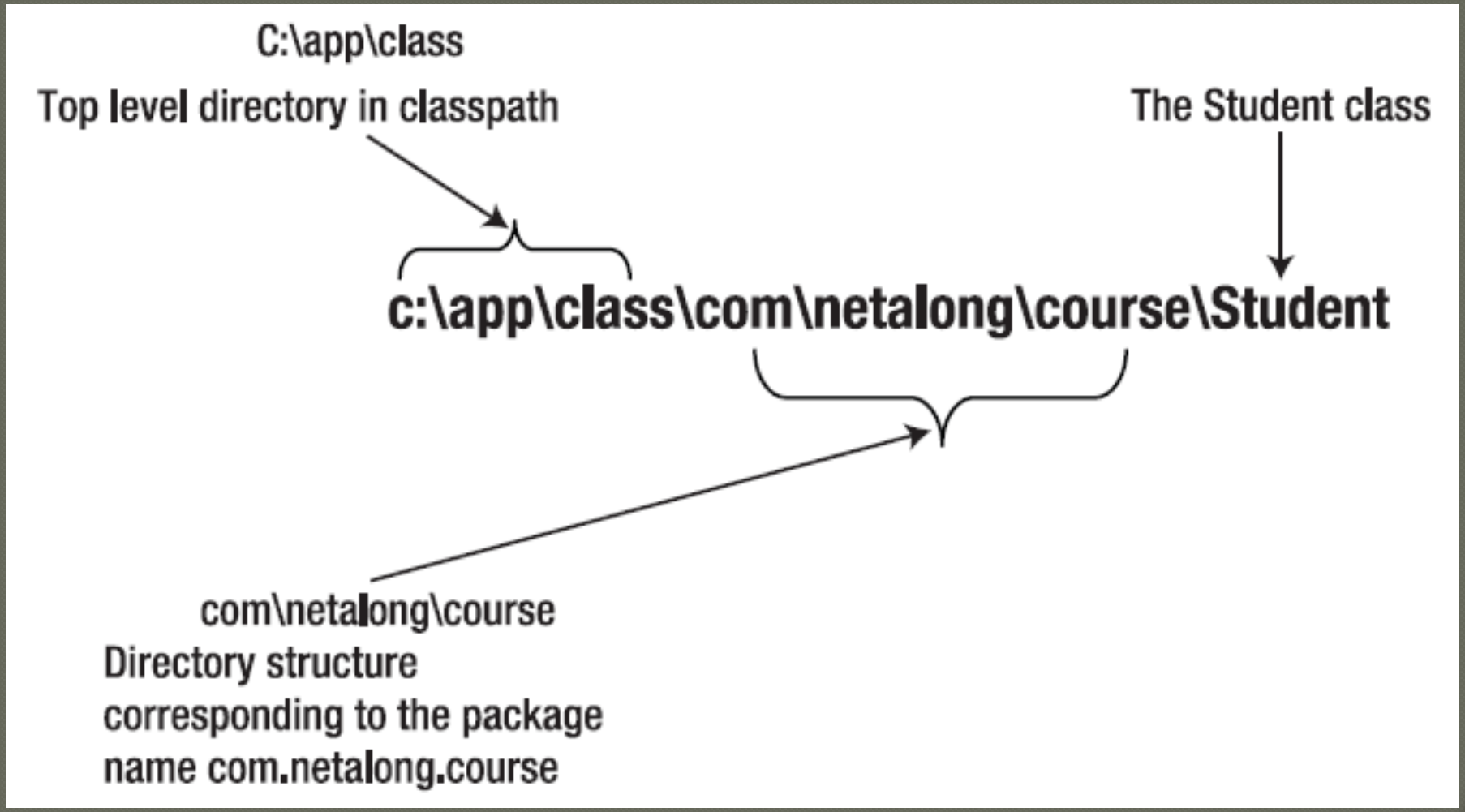


classpath

- When the compiler or the interpreter encounters a class name in your code, they look for classes in each directory or a JAR (Java Archive) file listed in the `classpath`
- Let's assume that we put the class files in the `c:\app\class\com\netalong\course` directory, the `classpath` must include this path name:

```
C:\app\class
```

How the Compiler and the Interpreter Construct a Full Path Name



The JAR Files

- All the directories of an application can be compressed into what is called a JAR file.
- A JAR file as a tree of directories.
- can be created with the `jar` command:

```
jar -cf myApp.jar topDir
```

- list directories and files in this JAR file:

```
jar -tf myApp.jar
```

- execute your application contained in the JAR file:

```
java -jar myApp.jar
```

Contents

Passing Arguments into Methods

- The values of parameters are called arguments and can be passed during a method call
- Whether the passed variable is a primitive or a reference variable, it is always the copy of the variable – pass by value

Passing a Primitive Variable

- A primitive variable holds a data item as its value
- When a primitive variable is passed as an argument in a method call, only the copy of the original variable is passed
- Any change to the passed variable in the called method will not affect the variable in the calling method

Listing 4-4. *Student.java*

```
1. class Student {
2. public static void main (String [] args) {
3.  int score = 75;
4.  Student st = new Student();
5.  st.modifyStudent(score);
6.  System.out.println("The original student score: " + score);
7. }
8. void modifyStudent(int i){
9.     i = i+10;
10.    System.out.println("The modified student score: " + i);
11. }
12.}
```

```
The modifed student score: 85
The original student score: 75
```

Passing a Reference Variable

- An object reference variable points to an object
- When you pass a reference variable in a method, you pass a copy of it
- The copy of the reference variable points to the same object to which the original variable points → the called method can change the object properties by using the passed reference

Contents

- Organizing Your Java Application
- Passing Arguments into Methods
- **Using Access Modifiers**
- Understanding Usage Modifiers
- Modifiers: The Big Picture
- Understanding Garbage Collection in Java

Using Access Modifiers

- Access modifiers determine the accessibility scope of the Java elements they modify
- The access modifiers may be used with a class and its members
- The Java language offers three explicit access modifiers, `public`, `protected`, and `private`, and a default modifier

The public Modifier

- The public modifier makes a Java element (a class or a class member) most accessible

Listing 4-6. *InstanceTest.java*

```
1. class MyClass {
2.   public int  myNumber = 10;
3.      public int getMyNumber(){
4.       return myNumber;
5.     }
6.   }
7.  class InstanceTest {
8.    public static void main(String[] args) {
9.        MyClass  mc = new MyClass();
10.      System.out.println (" The value of myNumber is " + mc.myNumber);
11.      System.out.println (" The value returned by the method is " +
              mc.getMyNumber());
12.   }
13.  }
```

The `private` Modifier

- The `private` modifier makes a Java element (a inner class or a class member) least accessible
- A `private` member of a class may only be accessed from the code inside the same class in which this member is declared
- A top-level class cannot be declared `private`; it can only be `public`, or default

Listing 4-7. *TestPrivateTest.java*

```
1. class PrivateTest {
2.
3.   // public int  myNumber = 10;
4.   private int  myNumber = 10;
5.  public int getMyNumber(){
6.     return myNumber;
7.   }
8. }
9. class SubPrivateTest extends PrivateTest {
10.    public void printSomething(){
11.     System.out.println (" The value of myNumber is " + this.myNumber);
12.      System.out.println (" The value returned by the method is " +
                this.getMyNumber());
13.    }
14. }
15. class TestPrivateTest{
16.   public static void main(String[] args) {
17.      SubPrivateTest spt = new SubPrivateTest();
18.      spt.printSomething();
19.  }
20. }
```

The protected Modifier

- This modifier applies only to class members
- A class member declared `protected` is accessible to the following elements:
  - All the classes in the same package that contains the class that owns the protected member.
  - All the subclasses of the class that owns the protected member.

```
1. package networking;
2. class Communicator {
3.          void sendData() {}
4.          protected void receiveData() {}
5. }
```

```
1. package internetworking;
2. import networking.*;
3. class Client extends Communicator {
4.    void  communicate(){
5.       receiveData();
6.    }
7. }
```

The Default Modifier

- When you do not specify any access modifier for an element, it is assumed that the access is *default*.
- A class or a class member declared *default* is accessible from anywhere in the same package in which the accessed class exists
- A method may not be overridden to be less accessible

Table 4-1. *Access Level that a Class Has Granted to Other Classes by Using Different Modifiers*

Access Modifier	Class	Subclass	Package	World
`private`	Yes	No	No	No
`protected`	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes
Default	Yes	No	Yes	No

Contents

Understanding Usage Modifiers

- modify the way a class or a class member is to be used:
  - `final`
  - `static`
  - `abstract`
  - `native`
  - `transient`
  - `volatile`
  - `synchronized`

The `final` Modifier

- may be applied to a class, a method, or a variable:
  - the value of the variable is constant
  - the class cannot be extended
  - the method cannot be overridden

The static Modifier

- can be applied to variables, methods, and a block of code
- The static elements of a class are visible to all the instances of the class
- If one instance of the class makes a change to a static element, all the instances will see that change

Listing 4-10. *RunStaticCodeExample.java*

```
1.  class StaticCodeExample {
2.  static int counter=0;
3.  static {
4.       counter++;
5.       System.out.println("Static Code block: counter: " + counter);
6.  }
7.  StaticCodeExample() {
8.          System.out.println("Constructor:  counter: " + counter);
9.  }
10.  static {
11.               System.out.println("This is another static block");
12.              }
13.}
14.  public class  RunStaticCodeExample {
15.  public static void main(String[] args) {
16.       StaticCodeExample sce = new StaticCodeExample();
17.       System.out.println("main: " + sce.counter);
18.  }
19.}
```

The abstract Modifier

- may be applied to a class or a method
- A class that is declared `abstract` cannot be instantiated
- A class must be declared `abstract`:
  - may have one or more abstract methods
  - may have inherited one or more abstract methods from its superclass, and has not provided implementation for all or some of them
  - implements an interface, but does not provide implementation for at least one method in the interface.

Listing 4-11. *RunShape.java*

```
1. abstract class Shape {
2.   abstract void draw();  //Note that there are no curly braces here
3.   void message() {
4.       System.out.println("I cannot live without being a parent.");
5.   }
6. }
7.  class Circle extends Shape {
8.   void draw() {
9.       System.out.println("Circle drawn.");
10.   }
11. }
12. class Cone extends Shape {
13.   void draw() {
14.       System.out.println("Cone drawn.");
15.   }
16.  }
17.  public class RunShape {
18.   public static void main(String[] args) {
19.       Circle circ = new Circle();
20.       Cone cone = new Cone();
21.       circ.draw();
22.       cone.draw();
23.       cone.message();
24.   }
25. }
```

Other Usage Modifiers

- `native`: to use a method that exists outside of the JVM
- `transient`: to instruct the JVM not to store the variable when the object in which it is declared is being serialized
- `volatile`: tells the accessing thread that it should synchronize its private copy of the variable with the master copy in the memory
- `synchronized`: to control access to critical sections in the program

Contents

Modifiers: The Big Picture

Table 4-2. *Summary of Modifiers Used by Java Classes and Class Members*

Modifier	Top-Level Class	Variable	Method	Constructor	Code Block
`public`	Yes	Yes	Yes	Yes	No
`private`	No	Yes	Yes	Yes	No
`protected`	No	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	Yes	N/A
`final`	Yes	Yes	Yes	No	No
`static`	No	Yes	Yes	No	Yes
`abstract`	Yes	No	Yes	No	No
`native`	No	No	Yes	No	No
`transient`	No	Yes	No	No	No
`volatile`	No	Yes	No	No	No
`synchronized`	No	No	Yes	No	Yes

Contents

- Organizing Your Java Application
- Passing Arguments into Methods
- Using Access Modifiers
- Understanding Usage Modifiers
- Modifiers: The Big Picture
- **Understanding Garbage Collection in Java**

Understanding Garbage Collection

- When you create an object, the object is put on the heap
- After an object in memory has been used and is no longer needed, it is sensible to free memory from that object: *garbage collection*
- Garbage collection is done automatically by what is called the *garbage collector*

Understanding the Garbage Collector

- automates memory management by freeing up the memory from objects that are no longer in use
- Make a request to the garbage collector:

```
System.gc();
Runtime.getRuntime().gc();
```

- An object is considered eligible for garbage collection when there is no reference pointing to it

The finalize() Method

- This method will be called by the garbage collector before deleting any object
- inherited from the `Object` class

```
protected void finalize() {
 super.finlaize();
 // clean up code follows.
}
```