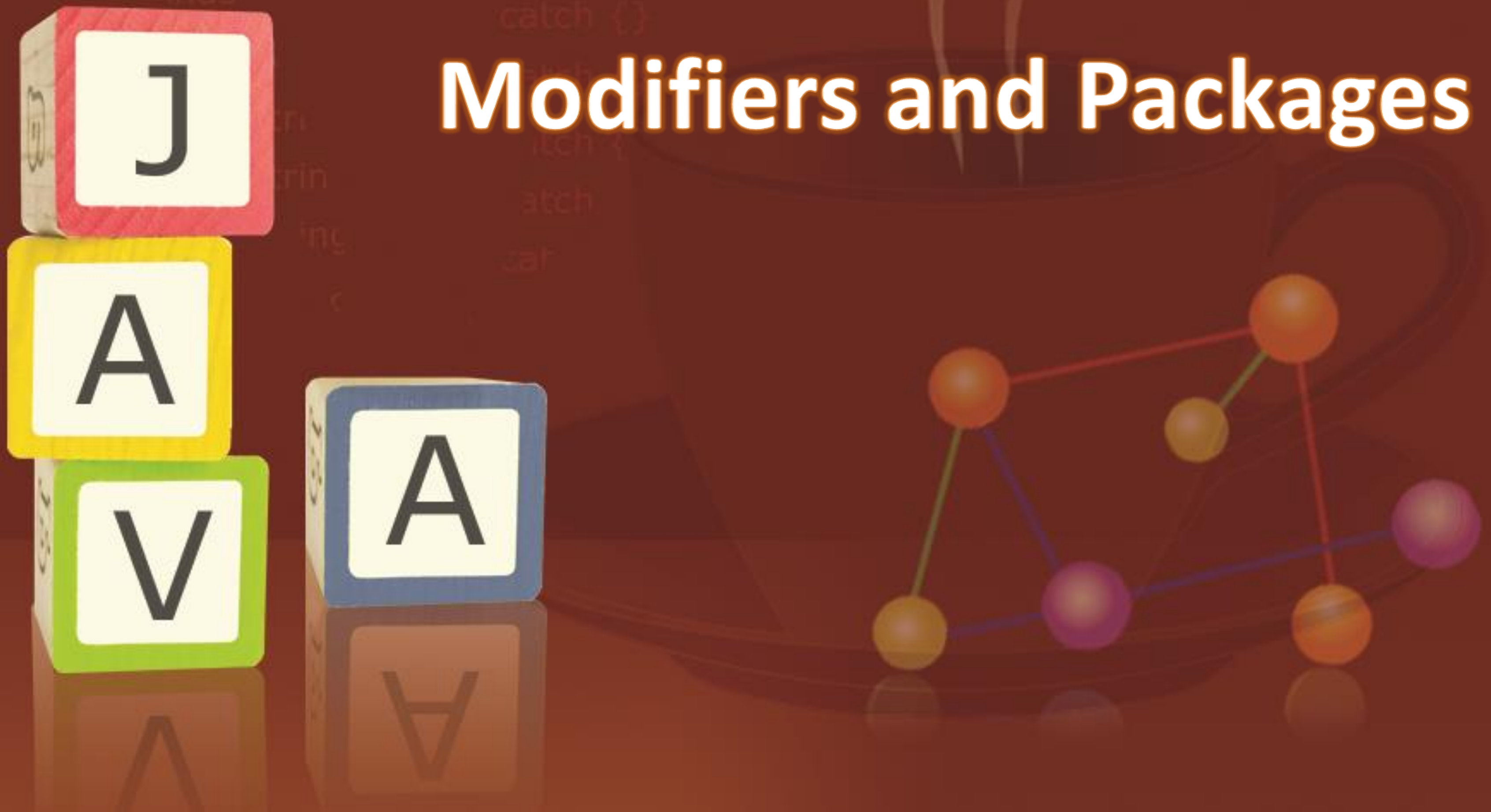
Fundamentals of Java

Session: 9

Modifiers and Packages



Objectives

- Describe field and method modifiers
- Explain the different types of modifiers
- Explain the rules and best practices for using field modifiers
- Describe class variables
- Explain the creation of static variables and methods
- Describe package and its advantages
- Explain the creation of user-defined package
- Explain the creation of .jar files for deployment

Introduction

- Java is a tightly encapsulated language.
- Java provides a set of access specifiers such as `public`, `private`, `protected`, and `default` that help to restrict access to class and class members.
- Java provides additional field and method modifiers to further restrict access to the members of a class to prevent modification by unauthorized code.
- Java provides the concept of class variables and methods to create a common data member that can be shared by all objects of a class as well as other classes.
- Java provides packages that can be used to group related classes that share common attributes and behavior.
- The entire set of packages can be combined into a single file called the `.jar` file for deployment on the target system.

Field and Method Modifiers

Field and method modifiers are keywords used to identify fields and methods that provide controlled access to users.

Some of these can be used in conjunction with access specifiers such as `public` and `protected`.

- The different field modifiers that can be used are as follows:

  - `volatile`
  - `native`
  - `transient`
  - `final`

‘volatile’ Modifier 1-2

The `volatile` modifier allows the content of a variable to be synchronized across all running threads.

A thread is an independent path of execution of code within a program. Many threads can run concurrently within a program.

The `volatile` modifier is applied only to fields.

Constructors, methods, classes, and interfaces cannot use this modifier.

The `volatile` modifier is not frequently used.

While working with a multithreaded program, the `volatile` keyword is used.

'volatile' Modifier 2-2

When multiple threads of a program are using the same variable, in general, each thread has its own copy of that variable in the local cache.

In such a case, if the value of the variable is updated, it updates the copy in the local cache and not the main variable present in the memory.

The other thread using the same variable does not get the updated value.

- To avoid this problem, a variable is declared as `volatile` to indicate that it will not be stored in the local cache.
- Whenever a thread updates the values of the variable, it updates the variable present in the main memory.
- This helps other threads to access the updated value.
- For example,

```
private volatile int testValue; // volatile variable
```

'native' Modifier 1-3

The `native` modifier is used only with methods.

It indicates that the implementation of the method is in a language other than Java such as C or C++.

Constructors, fields, classes, and interfaces cannot use this modifier.

The methods declared using the `native` modifier are called native methods.

The Java source file typically contains only the declaration of the native method and not its implementation.

The implementation of the method exists in a library outside the JVM.

- Before invoking a native method, the library that contains the method implementation must be loaded by making the following system call:

```
System.loadLibrary("libraryName");
```

- To declare a native method, the method is preceded with the `native` modifier.
- The implementation is not provided for the method. For example,

```
public native void nativeMethod();
```

‘native’ Modifier 2-3

- After declaring a native method, a complex series of steps are used to link it with the Java code.
- Following code snippet demonstrates an example of loading a library named `NativeMethodDefinition` containing a native method named `nativeMethod()`:

```
class NativeModifier {
  native void nativeMethod(); // declaration of a native method
  /**
   * static code block to load the library
   */
  static {
    System.loadLibrary(“NativeMethodDefinition”);
  }
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    NativeModifier objNative = new NativeModifier(); // line1
    objNative.nativeMethod(); // line2
  }
}
```

‘native’ Modifier 3-3

- Notice that a `static` code block is used to load the library.
- The `static` keyword indicates that the library is loaded as soon as the class is loaded.
- This ensures that the library is available when the call to the native method is made. The native method can be used in the same way as a non-native method.
- Native methods allow access to existing library routines created outside the JVM.
- However, the use of native methods introduces two major problems.

Impending security risk

- The native method executes actual machine code, and therefore, it can gain access to any part of the host system.
- That is, the native code is not restricted to the JVM execution environment.
- This may lead to a virus infection on the target system.

Loss of portability

- The native code is bundled in a DLL, so that it can be loaded on the machine on which the Java program is executing.
- Each native method is dependent on the CPU and the OS.
- This makes the DLL inherently non-portable.
- This means, that a Java application using native methods will run only on a machine in which a compatible DLL has been installed.

‘transient’ Modifier 1-2

When a Java application is executed, the objects are loaded in the Random Access Memory (RAM).

Objects can also be stored in a persistent storage outside the JVM so that it can be used later.

This determines the scope and life span of an object.

The process of storing an object in a persistent storage is called serialization.

For any object to be serialized, the class must implement the `Serializable` interface.

If `transient` modifier is used with a variable, it will not be stored and will not become part of the object’s persistent state.

The `transient` modifier is useful to prevent security sensitive data from being copied to a source in which no security mechanism has been implemented.

The `transient` modifier reduces the amount of data being serialized, improves performance, and reduces costs.

‘transient’ Modifier 2-2

- The `transient` modifier can only be used with instance variables.
- It informs the JVM not to store the variable when the object, in which it is declared, is serialized.
- Thus, when the object is stored in persistent storage, the instance variable declared as `transient` is not persisted.
- Following code snippet depicts the creation of a `transient` variable:

```
class Circle {

  transient float PI; // transient variable that will not persist
  float area; // instance variable that will persist

}
```

‘final’ Modifier 1-5

The `final` modifier is used when modification of a class or data member is to be restricted.

The `final` modifier can be used with a variable, method, and class.

Final Variable

A variable declared as `final` is a constant whose value cannot be modified.

A `final` variable is assigned a value at the time of declaration.

A compile time error is raised if a `final` variable is reassigned a value in a program after its declaration.

- Following code snippet shows the creation of a `final` variable:

```
final float PI = 3.14;
```

- The variable **`PI`** is declared `final` so that its value cannot be changed later.

‘final’ Modifier 2-5

Final Method

A method declared `final` cannot be overridden or hidden in a Java subclass.

The reason for using a `final` method is to prevent subclasses from changing the meaning of the method.

A `final` method is commonly used to generate a random constant in a mathematical application.

- Following code snippet depicts the creation of a `final` method:

```
final float getCommission(float sales){
  System.out.println(“A final method. . .”);
}
```

- The method **`getCommission()`** can be used to calculate commission based on monthly sales.
- The implementation of the method cannot be modified by other classes as it is declared as `final`.
- A `final` method cannot be declared abstract as it cannot be overridden.

'final' Modifier 3-5

Final Class

A class declared `final` cannot be inherited or subclassed.

Such a class becomes a standard and must be used as it is.

The variables and methods of a class declared `final` are also implicitly `final`.

- The reason for declaring a class as `final` is to limit extensibility and to prevent the modification of the class definition.
- Following code snippet shows the creation of a `final` class:

```
public final class Stock {
...
}
```

- The class **`Stock`** is declared `final`.
- All data members within this class are implicitly `final` and cannot be modified by other classes.

‘final’ Modifier 4-5

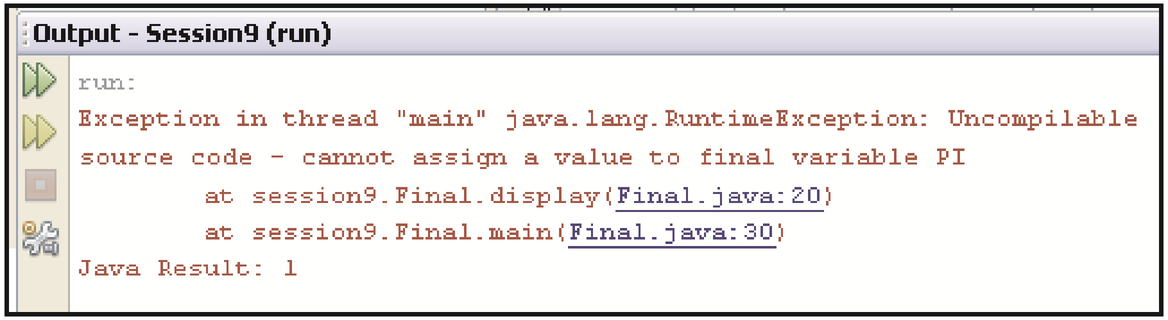
- Following code snippet demonstrates an example of creation of a `final` class:

```
package session9;
public class Final {

  // Declare and initialize a final variable
  final float PI = 3.14F; // variable to store value of PI

  /**
   * Displays the value of PI
   *
   * @param pi a float variable storing the value of PI
   * @return void
   */
  public void display(float pi) {
    PI = pi; // generates compilation error
    System.out.println(“The value of PI is:”+PI);
  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
```

‘final’ Modifier 5-5

```
    // Instantiate the Final class
    final Final objFinal = new Final();

    // Invoke the display() method
    objFinal.display(22.7F);
  }
}
```

- The class **`Final`** consists of a `final float` variable **`PI`** set to **`3.14`**.
- The method **`display()`** is used to set a new value passed by the user to **`PI`** .
- This leads to compilation error ‘**`cannot assign a value to final variable PI`**’.
- If the user chooses to run the program anyway, the following runtime error is issued as shown in the following figure:



Output - Session9 (run)

```
run:
Exception in thread "main" java.lang.RuntimeException: Uncompilable
source code - cannot assign a value to final variable PI
        at session9.Final.display(Final.java:20)
        at session9.Final.main(Final.java:30)
Java Result: 1
```

- To remove the error, the method signature should be changed and the statement, **`PI = pi;`** should be removed.

Rules and Best Practices for Using Field Modifiers

- Some of the rules for using field modifiers are as follows:

`Final` fields cannot be `volatile.`

`Native` **methods in Java cannot have a body.**

Declaring a `transient` **field as** `static` **or** `final` **should be avoided as far as possible.**

`Native` **methods violate Java's platform independence characteristic. Therefore, they should not be used frequently.**

A `transient` **variable may not be declared as** `final` **or** `static`**.**

Class Variables

Consider a situation, where in a user wants to create a counter that keeps track of the number of objects accessing a particular method.

In such a scenario, a variable is required that can be shared among all the objects of a class and any changes made to the variable are updated only in one common copy.

Java provides the implementation of such a concept of class variables by using the `static` keyword.

Declaring Class Variables

- Class variables are also known as `static` variables.
- Note that the `static` variables are not constants.
- Such variables are associated with the class rather than with any object.
- All instances of the class share the same value of the class variable.
- The value of a `static` variable can be modified using class methods or instance methods.
- Unlike instance variable, there exists only one copy of a class variable for all objects in one fixed location in memory.
- A `static` variable declared as **`final`** becomes a constant whose value cannot be modified.

- For example,

```
static int PI=3.14;  // static variable-can be modified
static final int PI=3.14;  // static constant-cannot be
                             modified
```

Creating Static Variables, Static Methods, and Static Blocks 1-6

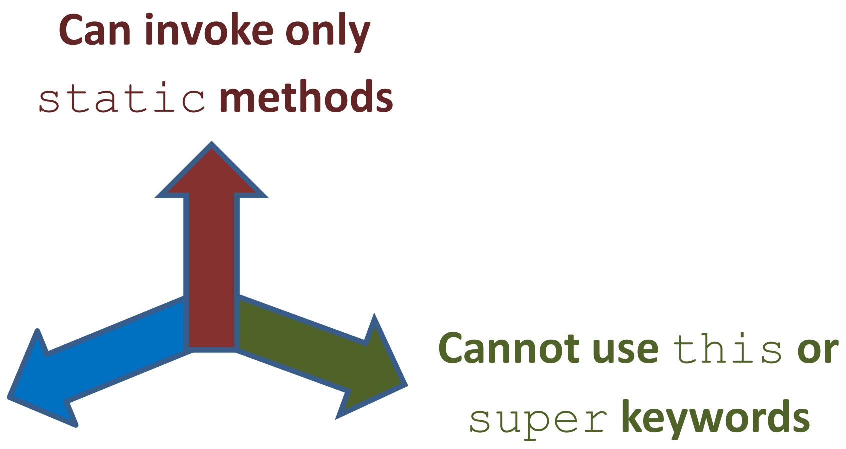
One can also create `static` methods and `static` initializer blocks along with `static` variables.

`Static` variables and methods can be manipulated without creating an instance of the class.

This is because there is only one copy of a `static` data member that is shared by all objects.

A `static` method can only access `static` variables and not instance variables.

- Methods declared as `static` have the following restrictions:


Can invoke only static methods
Can access only static data
Cannot use this or super keywords

Creating Static Variables, Static Methods, and Static Blocks 2-6

Generally, a constructor is used to initialize variables.

However, a `static` block can also be used to initialize `static` variables because `static` block is executed even before the `main()` method is executed.

It is used when a block of code needs to be executed during loading of the class by JVM.

Execution of Java code starts from `static` blocks and not from `main()` method. It is enclosed within {} braces.

There can be more than one `static` block in a program. They can be placed anywhere in the class.

A `static` initialization block can reference only those class variables that have been declared before it.

Creating Static Variables, Static Methods, and Static Blocks 3-6

- Following code snippet demonstrates an example of `static` variables, `static` method, and `static` block:

```
package session9;
public class StaticMembers {
  // Declare and initialize static variable
  public static int staticCounter = 0;

    // Declare and initialize instance variable
    int instanceCounter = 0;
    /**
     * static block
     *
     */
    static{
      System.out.println("I am a static block");
    }

  /**
   * Static method
   *
   * @return void
```

Creating Static Variables, Static Methods, and Static Blocks 4-6

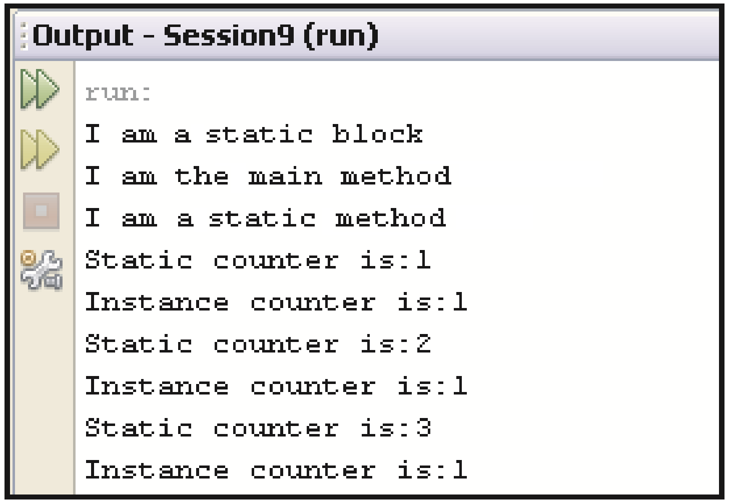
```
 */
public static void staticMethod(){
  System.out.println(“I am a static method”);
}

/**
 * Displays the value of static and instance counters
 *
 * @return void
 */
public void displayCount(){

  //Increment the static and instance variable
  staticCounter++;
  instanceCounter++;

  // Print the value of static and instance variable
  System.out.println(“Static counter is:”+ staticCounter);
  System.out.println(“Instance counter is:”+ instanceCounter);
}
```

Creating Static Variables, Static Methods, and Static Blocks 5-6

```
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {

     System.out.println("I am the main method");
     // Invoke the static method using the class name
     StaticMembers.staticMethod();

     // Create first instance of the class
     StaticMembers objStatic1 = new StaticMembers();
     objStatic1.displayCount();

     // Create second instance of the class
     StaticMembers objStatic2 = new StaticMembers();
     objStatic2.displayCount();

     // Create third instance of the class
     StaticMembers objStatic3 = new StaticMembers();
     objStatic3.displayCount();
   }
 }
```

Creating Static Variables, Static Methods, and Static Blocks 6-6

- Following figure shows the output of the program:



```
run:
I am a static block
I am the main method
I am a static method
Static counter is:1
Instance counter is:1
Static counter is:2
Instance counter is:1
Static counter is:3
Instance counter is:1
```

- From the figure it is clear that the `static` block is executed even before the `main()` method.
- Also, the value of `static` counter is incremented to 1, 2, 3, ..., and so on whereas the value of instance counter remains 1 for all the objects.
- This is because a separate copy of the instance counter exists for each object.
- However, for the `static` variable, only one copy exists per class.
- Every object increments the same copy of the variable, **`staticCounter`**.
- Thus, by using a `static` counter, a user can keep track of the number of instances of a class.

Packages 1-2

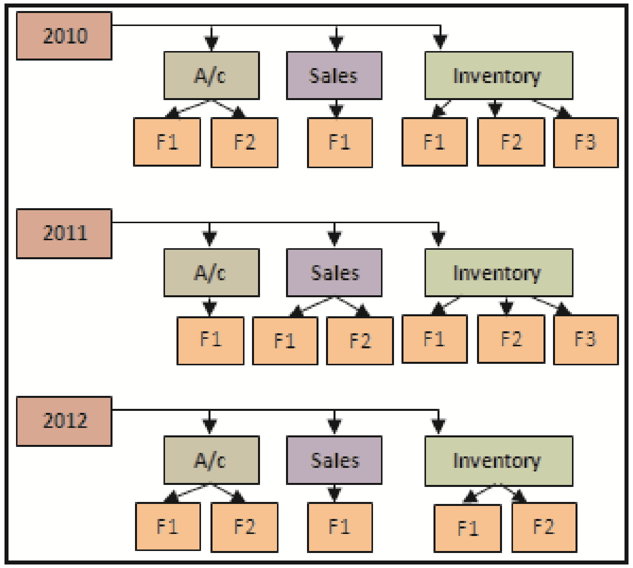
Consider a situation where in a user has about fifty files of which some are related to sales, others are related to accounts, and some are related to inventory.

Also, the files belong to different years. All these files are kept in one section of a cupboard.

Now, when a particular file is required, the user has to search the entire cupboard. This is very time consuming and difficult.

For this purpose, the user creates separate folders and divides the files according to the years and further groups them according to the content.

- This is depicted in the following figure:



Packages 2-2

- Similarly, in Java, one can organize the files using packages.

A package is a namespace that groups related classes and interfaces and organizes them as a unit.

- For example, one can keep source files in one folder, images in another, and executables in yet another folder. Packages have the following features:

  - A package can have sub packages.
  - A package cannot have two members with the same name.
  - If a class or interface is bundled inside a package, it must be referenced using its fully qualified name, which is the name of the Java class including its package name.
  - If multiple classes and interfaces are defined within a package in a single Java source file, then only one of them can be public.
  - Package names are written in lowercase.
  - Standard packages in the Java language begin with `java` or `javax`.

Advantages of Using Packages

- One can easily determine that these classes are related.
- One can know where to find the required type that can provide the required functions.
- The names of classes of one package would not conflict with the class names in other packages as the package creates a new namespace.
- For example, **`myPackage1.Sample`** and **`myPackage2.Sample`**.
- One can allow classes within one package to have unrestricted access to one another while restricting access to classes outside the package.
- Packages can also store hidden classes that can be used within the package, but are not visible or accessible outside the package.
- Packages can also have classes with data members that are visible to other classes, but not accessible outside the package.
- When a program from a package is called for the first time, the entire package gets loaded into the memory.
- Due to this, subsequent calls to related subprograms of the same package do not require any further disk Input/Output (I/O).

Type of Packages 1-11

- The Java platform comes with a huge class library which is a set of packages.
- These classes can be used in applications by including the packages in a class.
- This library is known as the Application Programming Interface (API).
- Every Java application or applet has access to the core package in the API, the `java.lang` package.
- For example,

The `String` class stores the state and behavior related to character strings.

The `File` class allows the developer to create, delete, compare, inspect, or modify a file on the file system.

The `Socket` class allows the developer to create and use network sockets.

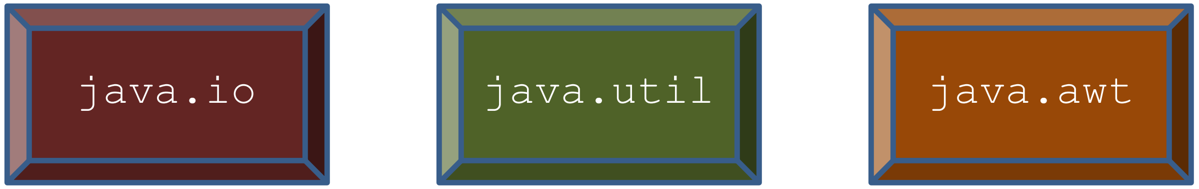
The various Graphical User Interface (GUI) control classes such as `Button`, `Checkbox`, and so on provide ready to use GUI controls.

Type of Packages 2-11

- The different types of Java packages are as follows:

Predefined packages
User-defined packages

- The predefined packages are part of the Java API. Predefined packages that are commonly used are as follows:



Type of Packages 3-11

- To create user-defined packages, perform the following steps:

1

- Select an appropriate name for the package by using the following naming conventions:

  - Package names are usually written in all lower case to avoid conflict with the names of classes or interfaces.
  - Companies usually attach their reversed Internet domain name as a prefix to their package names.
  - For example, **`com.sample.mypkg`** for a package named **`mypkg`** created by a programmer at sample.com.
  - Naming conflicts occurring in the projects of a single company are handled according to the naming conventions specific to that company.
  - This is done usually by including the region name or the project name after the company name.
  - For example, **`com.sample.myregion.mypkg`**.
  - Package names should not begin with `java` or `javax` as they are used for packages that are part of Java API.

Type of Packages 4-11

- In certain cases, the Internet domain name may not be a valid package name.
- For example, if the domain name contains special characters such as hyphen, if the package name consists of a reserved Java keyword such as `char`, or if the package name begins with a digit or some other character that is illegal to use as the beginning of a Java package name.
- In such a case, it is advisable to use an underscore as shown in the following table:

Domain Name	Suggested Package Name
`sample-name.sample.org`	`org.sample.sample_name`
`sample.int`	`int_.sample`
`007name.sample.com`	`com.sample._007name`

2
- Create a folder with the same name as the package.

3
- Place the source files in the folder created for the package.

Type of Packages 5-11

4

- Add the package statement as the first line in all the source files under that package as depicted in the following code snippet:

```
package session9;
class StaticMembers{
 public static void main(String[] args)
  {}
}
```

- Note that there can only be one package statement in a source file.

5

- Save the source file **`StaticMembers.java`** in the package **`session9`**.

6

- Compile the code as follows:

```
javac StaticMembers.java
```

OR

- Compile the code with `–d` option as follows:

```
javac –d . StaticMembers.java
```

- where, `–d` stands for directory and '`.`' stands for current directory.
- The command will create a sub-folder named **`session9`** and store the compiled class file inside it.

Type of Packages 6-11

7

- From the parent folder of the source file, execute it using the fully qualified name as follows:

```
java session9.StaticMembers
```

- Java allows the user to import the classes from predefined as well as user-defined packages using an import statement.
- However, the access specifiers associated with the class members will determine if the class members can be accessed by a class of another package.
- A member of a `public` class can be accessed outside the package by doing any of the following:
  - Referring to the member class by its fully qualified name, that is, `package-name.class-name`.
  - Importing the package member, that is, `import package-name.class-name`.
  - Importing the entire package, that is, `import package-name.*`.

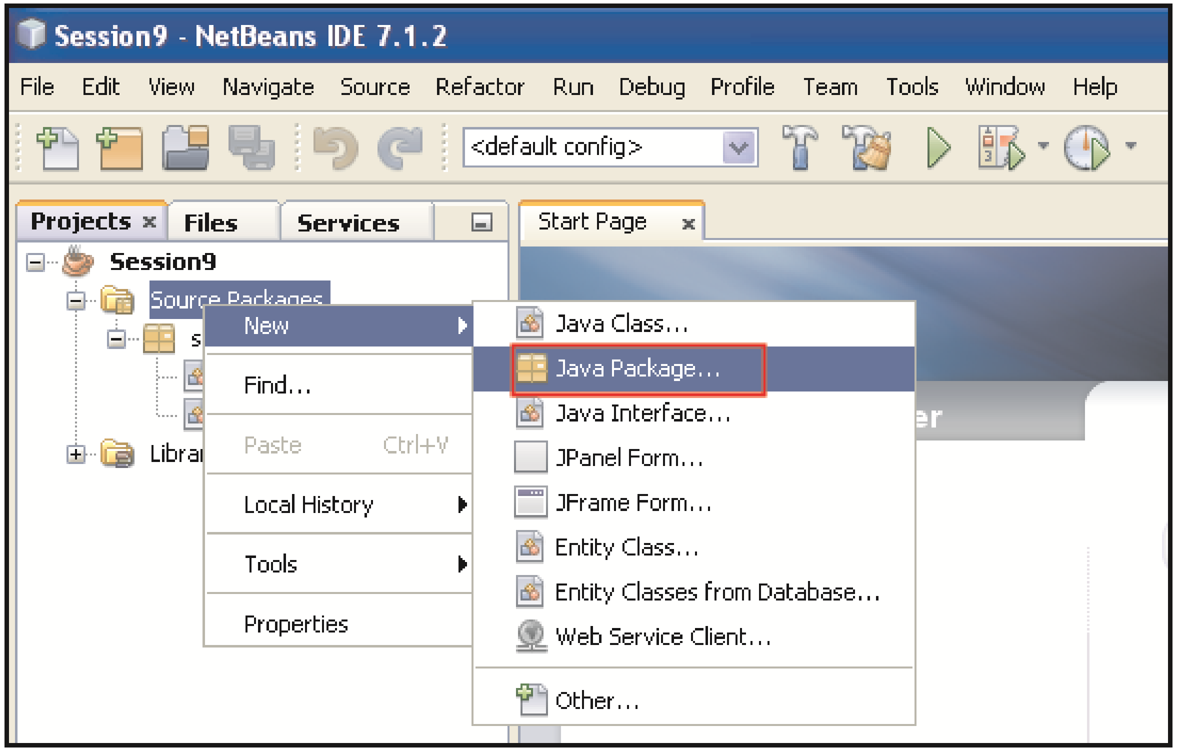
Type of Packages 7-11

- To create a new package using NetBeans IDE, perform the following steps:

1
- Open the project in which the package is to be created.
- In this case **`Session9`** project has been chosen.

2
- Right-click **Source Packages → New → Java Package** to display the **New Java Package** dialog box.

  - For example, in the following figure, the project **`Session9`** is opened and the **Java Package** option is selected:



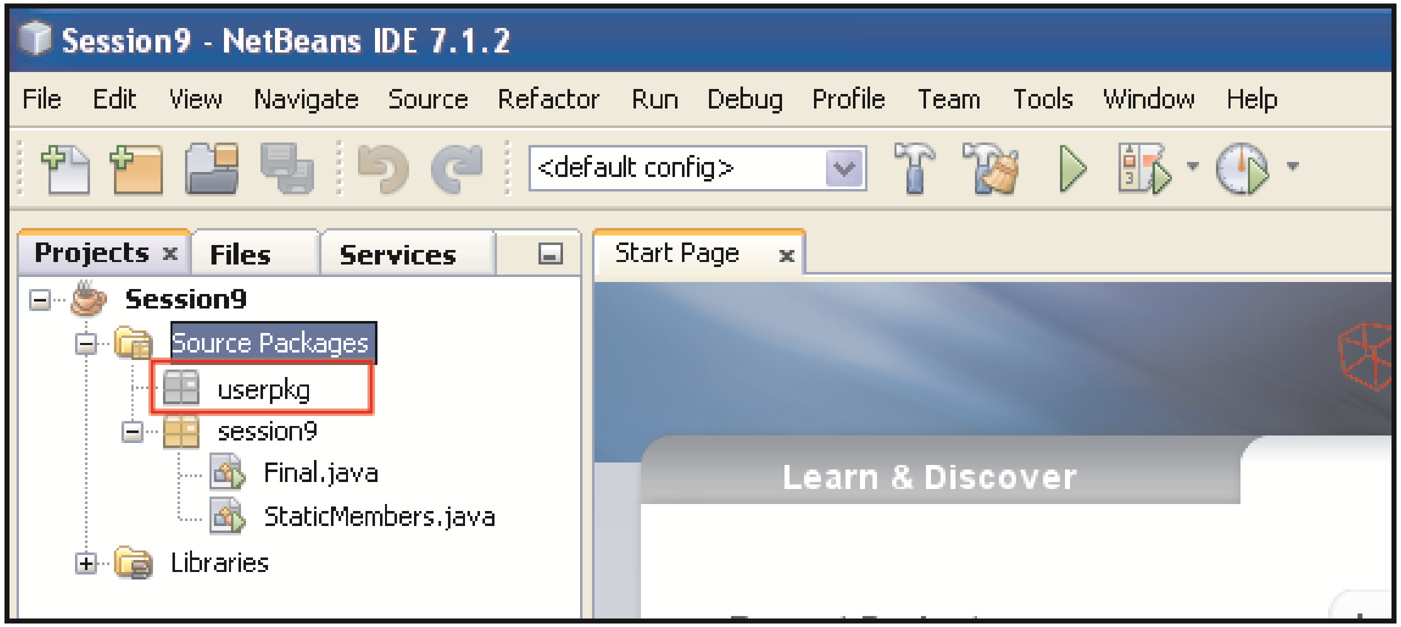
Type of Packages 8-11

3

- Type **`userpkg`** in the **Package Name** box of the **New Java Package** dialog box that is displayed.

4

- Click **Finish**. The **userpkg** package is created as shown in the following figure:



5

- Right-click **`userpkg`** and select **New → Java Class** to add a new class to the package.

6

- Type **`UserClass`** as the **Class Name** box of the **New Java Class** dialog box and click **Finish.**

Type of Packages 9-11

7

- Type the code in the class as depicted in the following code snippet:

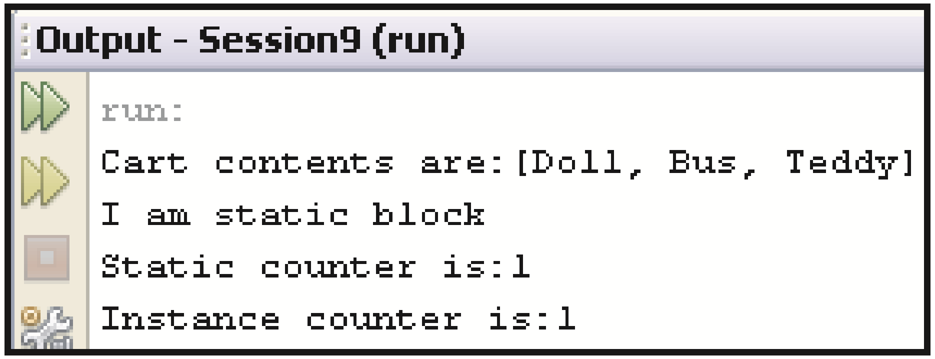
```
package userpkg;
// Import the predefined and user-defined packages
import java.util.ArrayList;
import session9.StaticMembers;

  public class UserClass {

  // Instantiate ArrayList class of java.util package
  ArrayList myCart = new ArrayList(); // line 1

  /**
   * Initializes an ArrayList
   *
   * @return void
   */
  public void createList() {

    // Add values to the list
    myCart.add("Doll");
```

Type of Packages 10-11

```
      myCart.add("Bus");
      myCart.add("Teddy");
      // Print the list
      System.out.println("Cart contents are:"+ myCart);
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

      // Instantiate the UserClass class
      UserClass objUser = new UserClass();
      objUser.createList(); // Invoke the createList() method

      // Instantiate the StaticMembers class
      StaticMembers objStatic = new StaticMembers();
      objStatic.displayCount(); // Invoke the displayCount() method
    }
  }
```

Type of Packages 11-11

- The two import statements namely, `java.util.ArrayList` and **`session9.StaticMembers`** are used to import the packages `java.util` and **`session9`** into the **`UserClass`** class.
- Following figure shows the output of the program:



Output - Session9 (run)

```
run:
Cart contents are:[Doll, Bus, Teddy]
I am static block
Static counter is:1
Instance counter is:1
```

- The output shows the execution of the `static` block of **`StaticMembers`** class also.
- This is because the `static` block is executed as soon as the class is launched.

Creating .jar Files for Deployment 1-16

- All the source files of a Java application are bundled into a single archive file called the Java Archive (JAR).
- The `.jar` file contains the class files and additional resources associated with the application.
- The `.jar` file format provides several advantages as follows:

Security	• The `.jar` file can be digitally signed so that only those users who recognize your signature can optionally grant the software security privileges that the software might not otherwise have.
Decrease in Download Time	• The source files bundled in a `.jar` file can be downloaded to a browser in a single HTTP transaction without having to open a new connection for each file.
File Compression	• The `.jar` format compresses the files for efficient storage.

Creating .jar Files for Deployment 2-16

Packaging for Extensions

- The extension framework in Java allows adding additional functionality to the Java core platform.
- The `.jar` file format defines the packaging for extensions. For example, Java 3D and Java Mail extensions developed by Sun Microsystems.

Package Sealing

- Java provides an option to seal the packages stored in the `.jar` files so that the packages can enforce version consistency.
- When a package is sealed within a `.jar` file, it implies that all classes defined in that package must be available in the same `.jar` file.

Package Versioning

- A `.jar` file can also store additional information about the files, such as vendor and version information.

Portability

- The `.jar` files are packaged in a ZIP file format.
- This enables the user to use them for tasks such as lossless data compression, decompression, archiving, and archive unpacking.

Creating .jar Files for Deployment 3-16

- To perform basic tasks with `.jar` files, one can use the Java Archive Tool.
- This tool is provided with the JDK.
- The Java Archive Tool is invoked by using the `jar` command.
- The basic syntax for creating a `.jar` file is as follows:

Syntax

```
jar cf jar-file-name input-file-name(s)
```

where,

`c`: indicates that the user wants to CREATE a `.jar` file.

`f`: indicates that the output should go to a file instead of `stdout`.

`jar-file-name`: represents the name of the resulting `.jar` file. Any name can be used for a `.jar` file. The `.jar` extension is provided with the file name, though it is not required.

`input-file-name(s)`: represents a list of one or more files to be included in the `.jar` separated by a space. This argument can contain the wildcard symbol '`*`' as well. If any of the input files specified is a directory, the contents of that directory are added to the `.jar` recursively.

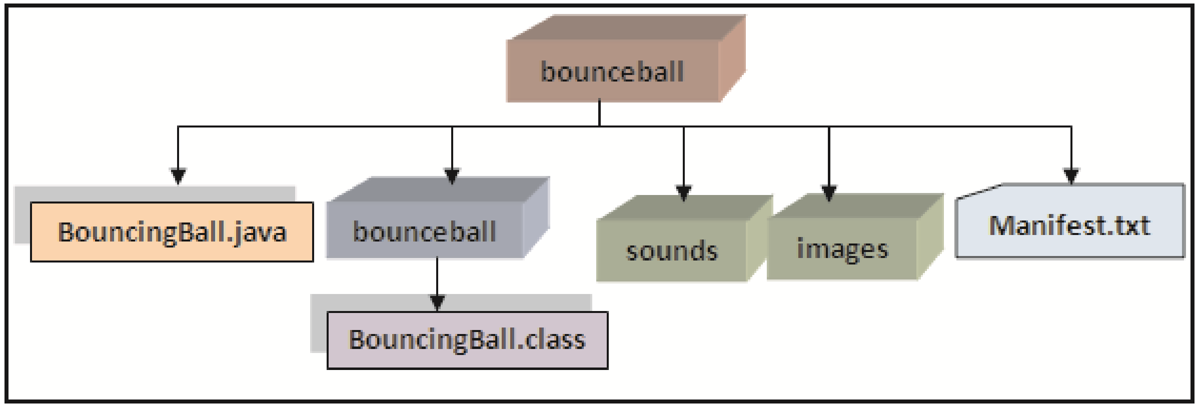
Creating .jar Files for Deployment 4-16

- The options `c` and `f` can be used in any order, but without any space in between.
- The `jar` command generates a compressed `.jar` file and places it by default in the current directory.
- Also, it will generate a default manifest file for the `.jar` file.
- The metadata in the `.jar` file such as entry names, contents of the manifest, and comments must be encoded in UTF8.
- Some of the other options, apart from `cf`, available with the `jar` command are listed in the following table:

Option	Description
`v`	Produces VERBOSE output on stdout while the `.jar` is begin built. The output displays the name of each of the files that are included in the `.jar` file.
`0 (zero)`	Indicates that the `.jar` file must not be compressed.
`M`	Indicates that the default manifest file must not be created.
`m`	Allows inclusion of manifest information from an existing manifest file. `jar cmf existing-manifest-name jar-file-name input-file-name(s)`
`-c`	Used to change directories during execution of the command.

Creating .jar Files for Deployment 5-16

- When a `.jar` file is created, the time of creation is stored in the `.jar` file.
- Therefore, even if the contents of the `.jar` file are not changed, if the `.jar` file is created multiple times, the resulting files will not be exactly identical.
- For this reason, it is advisable to use versioning information in the manifest file instead of creation time, to control versions of a `.jar` file.
- Consider the following files of a simple **`BouncingBall`** game application as shown in the following figure:



- The figure shows the **`BouncingBall`** application with the source file **`BouncingBall.java`**, class file **`BouncingBall.class`**, **`sounds`** directory, **`images`** directory, and **`Manifest.txt`** file.
- **`sounds`** and **`images`** are subdirectories that contain the sound files and animated .gif images used in the application.

Creating .jar Files for Deployment 6-16

- To create a `.jar` of the application using command line, perform the following steps:

1
- Create the directory structure as shown in the earlier figure.

2
- Create a text file with the code depicted in the following code snippet:

```
package bounceball;
public class BouncingBall {

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    System.out.println("This is the bouncing ball game");
  }
}
```

3
- Save the file as **`BouncingBall.java`** in the source package **`bounceball`**.

Creating .jar Files for Deployment 7-16

4

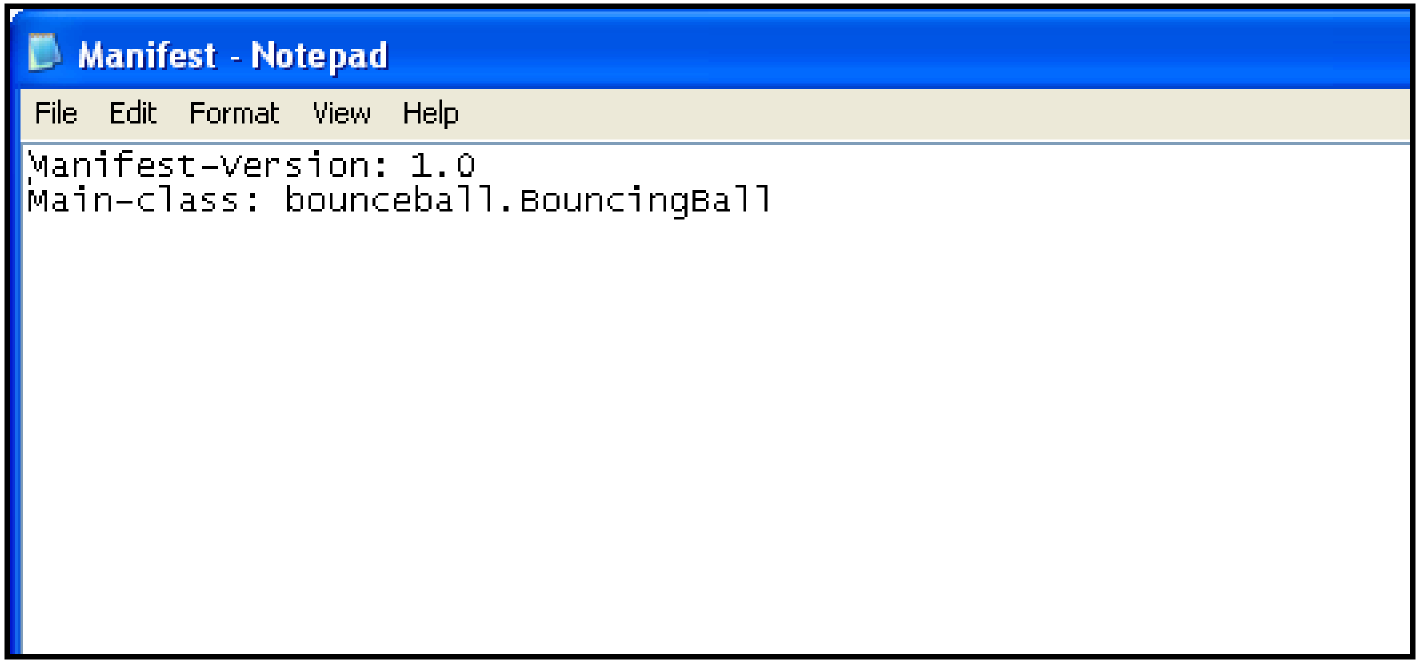
- Compile the `.java` file at command prompt by writing the following command:

```
javac -d . BouncingBall.java
```

- The command will create a subfolder with the same name **`bounceball`** and store the class file **`BouncingBall.java`** in that directory.

5

- Create a text file with the `Main-class` attribute as shown in the following figure:



```
Manifest-Version: 1.0
Main-class: bounceball.BouncingBall
```

Creating .jar Files for Deployment 8-16

6

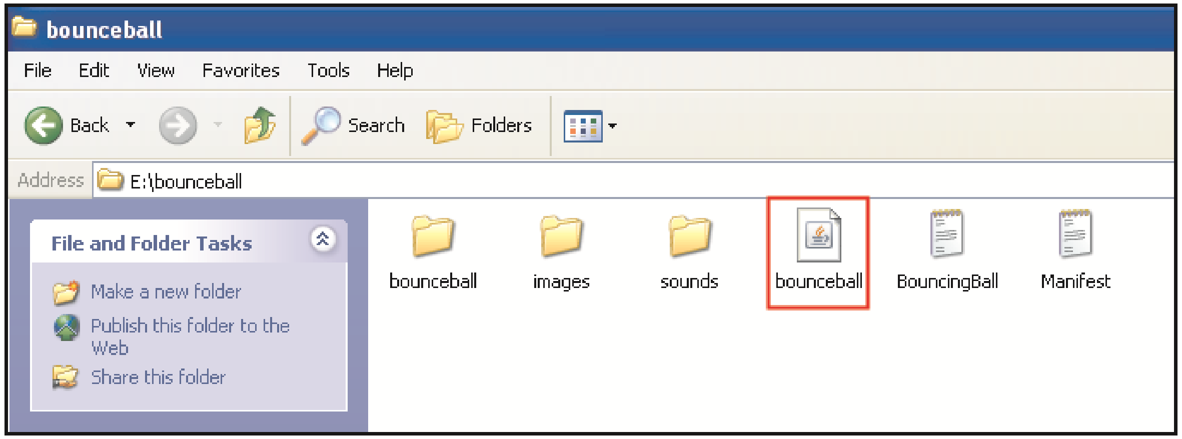
- Save the file as **`Manifiest.txt`** in the source **`bounceball`** folder.
  - The **`Manifest.txt`** file will be referred by the Jar tool for the Main class during `.jar` creation.
  - This will inform the Jar tool about the starting point of the application.

7

- To package the application in a single `.jar` named **`BouncingBall.jar`**, write the following command:

```
jar cvmf Manifest.txt bounceball.jar
  bounceball/BouncingBall.class sounds images
```

  - This will create a **`bounceball.jar`** file in the source folder as shown in the following figure:



Creating .jar Files for Deployment 9-16

- The command options `cvmf` indicate that the user wants to create a `.jar` file with verbose output using the existing manifest file, **`Manifest.txt`**.
- The name of the output file is specified as **`bounceball.jar`** instead of `stdout`.
- Further, the name of the class file is provided with its location as **`bounceball/BouncingBall.class`** followed by the directory names **`sounds`** and **`images`** so that the respective files under these directories are also included in the `.jar` file.

8

- To execute the `.jar` file at command prompt, type the following command:

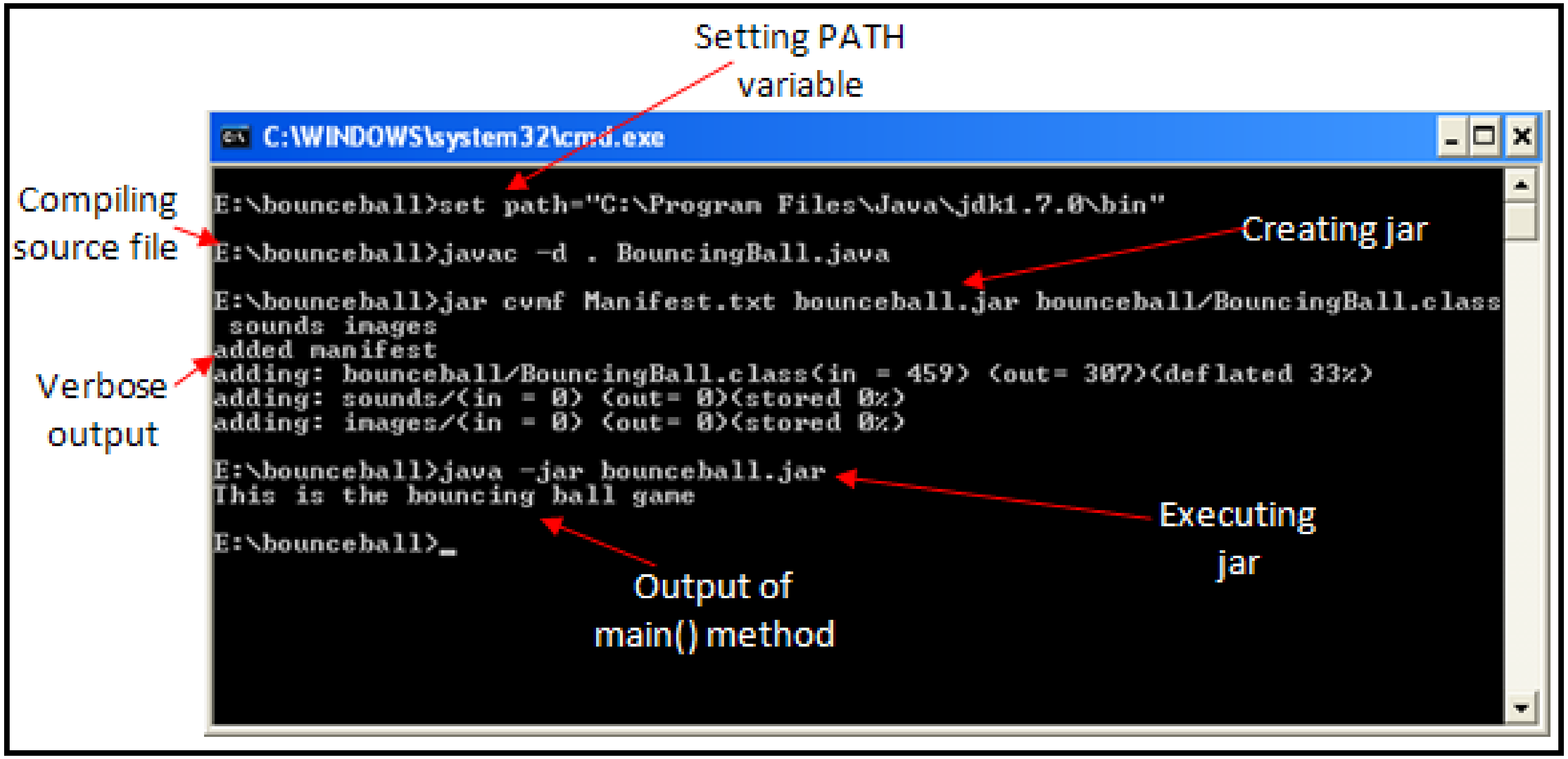
```
java -jar bounceball.jar
```

- The command will execute the `main()` method of the class of the `.jar` file and print the following output:

```
This is the bouncing ball game.
```

Creating .jar Files for Deployment 10-16

- Following figure shows the entire series of steps for creation of `.jar` file with the verbose output and the final output after `.jar` file execution:



```
E:\bounceball>set path="C:\Program Files\Java\jdk1.7.0\bin"

E:\bounceball>javac -d . BouncingBall.java

E:\bounceball>jar cvmf Manifest.txt bounceball.jar bounceball/BouncingBall.class
 sounds images
added manifest
adding: bounceball/BouncingBall.class(in = 459) (out= 307)(deflated 33%)
adding: sounds/(in = 0) (out= 0)(stored 0%)
adding: images/(in = 0) (out= 0)(stored 0%)

E:\bounceball>java -jar bounceball.jar
This is the bouncing ball game

E:\bounceball>_
```

Creating .jar Files for Deployment 11-16

Since sounds and images are directories, the Jar tool will recursively place the contents in the `.jar` file.

The resulting `.jar` file **`BouncingBall.jar`** will be placed in the current directory.

The use of the option '`v`' will shows the verbose output of all the files that are included in the `.jar` file.

In the example, the files and directories in the archive retained their relative path names and directory structure.

One can use the `-c` option to create a `.jar` file in which the relative paths of the archived files will not be preserved.

- For example, suppose one wants to put sound files and images used by the **`BouncingBall`** program into a `.jar` file, and that all the files should be on the top level, with no directory hierarchy.

- One can accomplish this by executing the following command from the parent directory of the sounds and images directories:

```
jar cf SoundImages.jar -c sounds . -c images .
```

Creating .jar Files for Deployment 12-16

- Here, ‘**-c sounds**’ directs the Jar tool to the sounds directory and the ‘.’ following ‘**-c sounds**’ directs the Jar tool to archive all the contents of that directory.

- Similarly, ‘**-c images .**’ performs the same task with the images directory.

- The resulting .jar file would consist of all the sound and image files of the sounds and images folder as follows:

```
META-INF/MANIFEST.MF
beep.au
failure.au
ping.au
success.au
greenball.gif
redball.gif
table.gif
```

- However, if the following command is used without the -c option:

```
jar cf SoundImages.jar sounds images
```

Creating .jar Files for Deployment 13-16

- The resulting `.jar` file would have the following contents:

```
META-INF/MANIFEST.MF
sounds/beep.au
sounds/failure.au
sounds/ping.au
sounds/success.au
images/greenball.gif
images/redball.gif
images/table.gif
```

- Following table shows a list of frequently used `jar` command options:

Task	Command
To create a `.jar` file	`jar cf jar-file-name input-file-name(s)`
To view contents of a `.jar` file	`jar tf jar-file-name`
To extract contents of a `.jar` file	`jar xf jar-file-name`
To run the application packaged into the `.jar` file. **`Manifest.txt`** file is required with `Main-class` header attribute.	`java -jar jar-file-name`

Creating .jar Files for Deployment 14-16

- To create a `.jar` file of the application using NetBeans IDE, perform the following steps:

1
- Create a new package **bounceball** in the **Session9** application.

2
- Create a new java class named **BouncingBall.java** within the **bounceball** package.

3
- Type the code depicted in the following code snippet in the **BouncingBall** class:

```
package bounceball;
  public class BouncingBall {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args)
  {
    System.out.println("This is the bouncing ball game");
  }
}
```

4
- Set **BouncingBall.java** as the main class in the **Run** properties of the application.

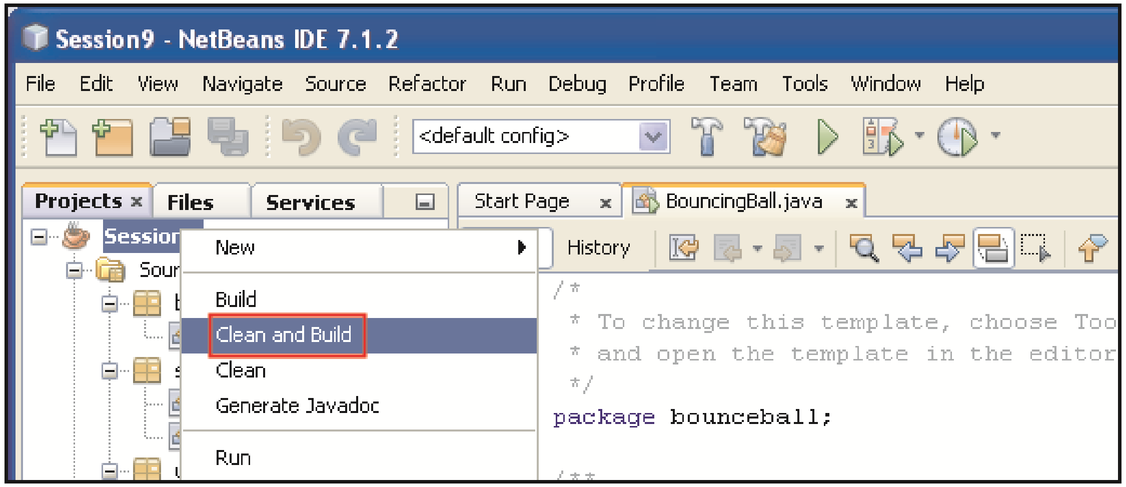
Creating .jar Files for Deployment 15-16

5

- Run the application by clicking the **Run** icon on the toolbar.
- The output will be shown in the **Output** window.

6

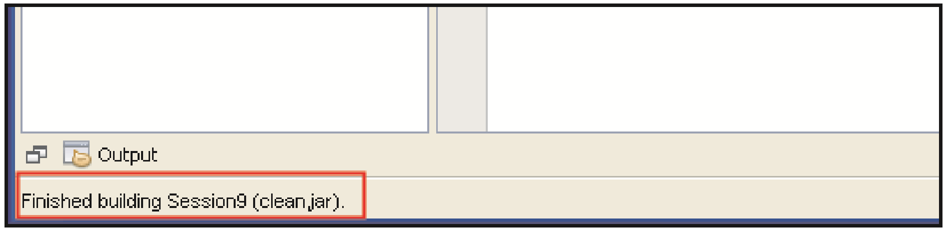
- To create the `.jar` file, right-click the **Session9** application and select **Clean and Build** option as shown in the following figure:


Session9 - NetBeans IDE 7.1.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects Files Services
Start Page BouncingBall.java
History
New
Build
Clean and Build
Clean
Generate Javadoc
Run

```
/*
 * To change this template, choose Too
 * and open the template in the editor
 */
package bounceball;
```

- The IDE will build the application and generate the `.jar` file.

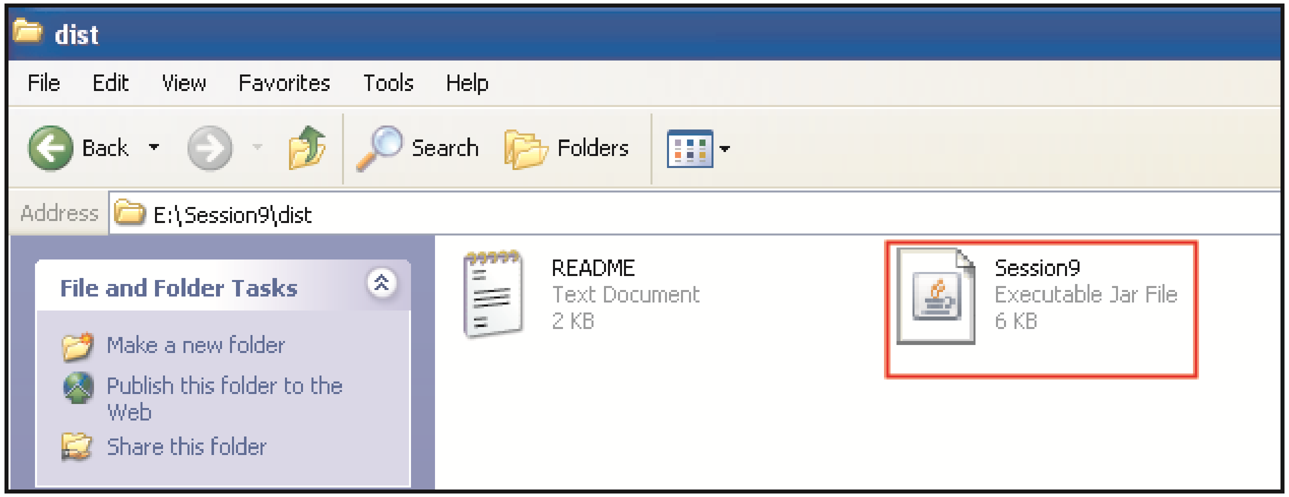
- A message as shown in the following figure will be displayed to the user in the status bar, once `.jar` file generation is finished:



```
Finished building Session9 (clean,jar).
```

Creating .jar Files for Deployment 16-16

- The **Clean and Build** command creates a new dist folder into the application folder and places the `.jar` file into it as shown in the following figure:



- User can load this `.jar` file in any device that has a JVM and execute it by double-clicking the file or run it at command prompt by writing the following command:

```
java -jar Session9.jar
```

Summary

- Field and method modifiers are used to identify fields and methods that have controlled access to users.
- The volatile modifier allows the content of a variable to be synchronized across all running threads.
- A thread is an independent path of execution within a program.
- The native modifier indicates that the implementation of the method is in a language other than Java such as C or C++.
- The transient modifier can only be used with instance variables. It informs the JVM not to store the variable when the object, in which it is declared, is serialized.
- The final modifier is used when modification of a class or data member is to be restricted.
- Class variables are also known as static variables and there exists only one copy of that variable for all objects.
- A package is a namespace that groups related classes and interfaces and organizes them as a unit.
- All the source files of a Java application are bundled into a single archive file called the Java Archive (JAR).