

Professional Programming in Java

Session: 14

Java Class Design and Advanced Class Design



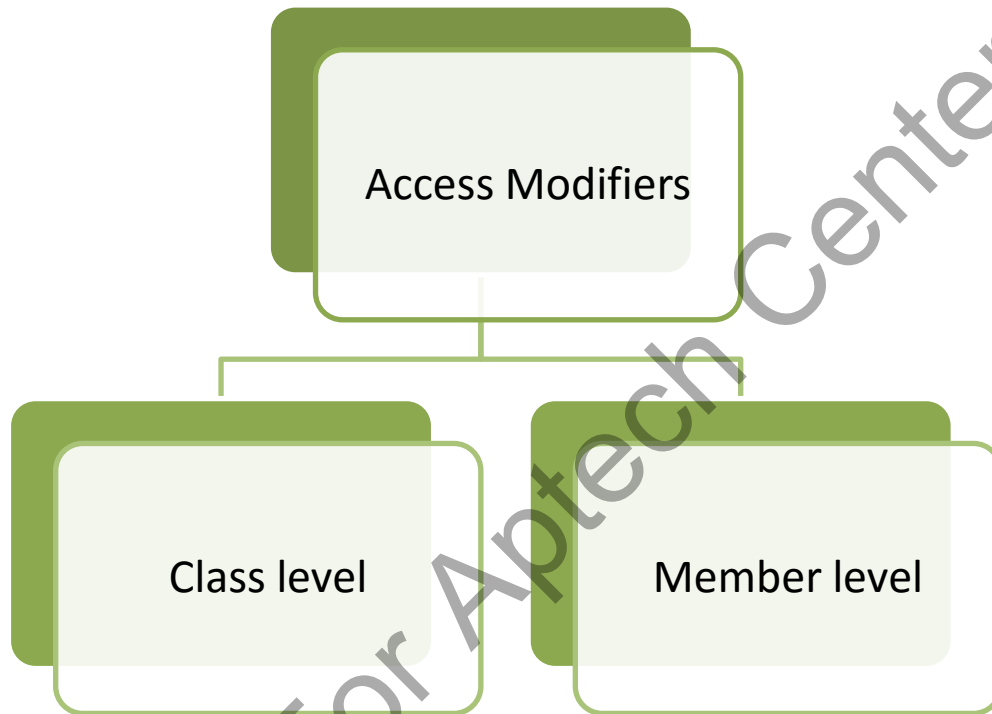


- ◆ Explain Java access modifiers
- ◆ Explain advanced OOP concepts in Java
- ◆ Describe packages
- ◆ Define abstract class
- ◆ Explain the use of static and final keywords
- ◆ Identify different types of inner classes
- ◆ Explain advanced types





- ◆ Access control determines how a class and its member variables and methods are accessible and used by other classes and objects.





◆ Access modifiers in Java:

public

- Can be applied to classes, member variables, and methods.
- Accessible from within the same class.
- Is applied using the public keyword.

protected

- Can be applied to member variables and methods.
- Accessible only to the class in which they are declared and its subclasses.
- Is applied using the protected keyword.

private

- Can be applied to member variables and methods.
- Accessible only to the class in which they are declared.
- Is applied using the private keyword.



- ◆ Following table shows the access levels of the different access modifiers:

Access Modifier	Within Class	Within Package	Subclass Outside Package	Global
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
friendly or package	Yes	Yes	No	No
private	Yes	No	No	No



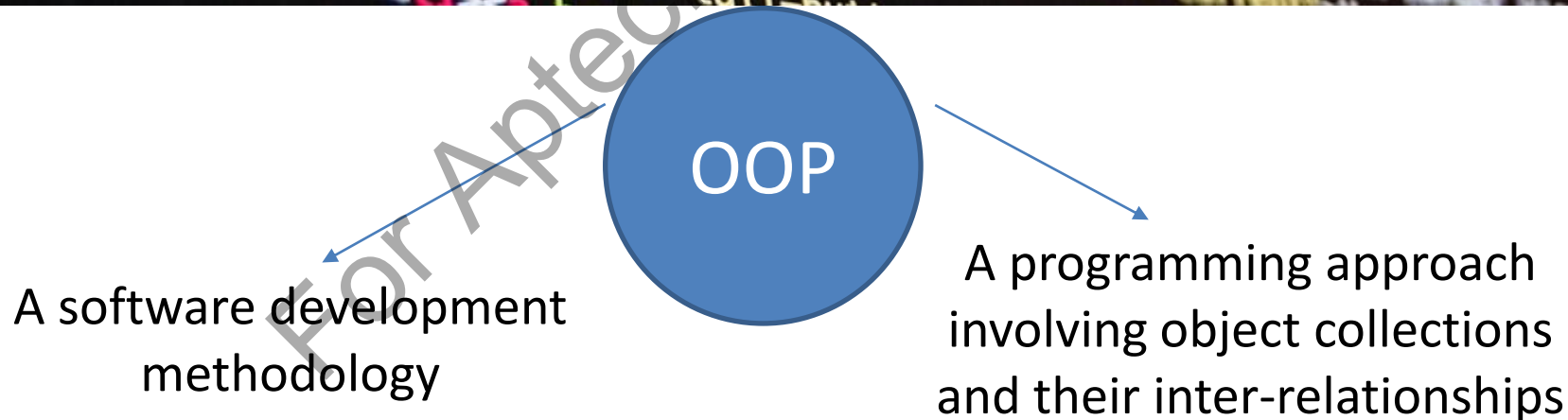
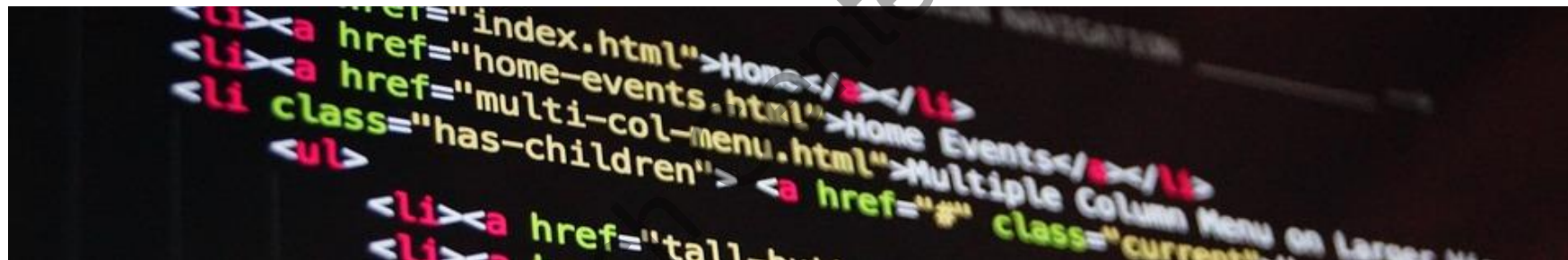
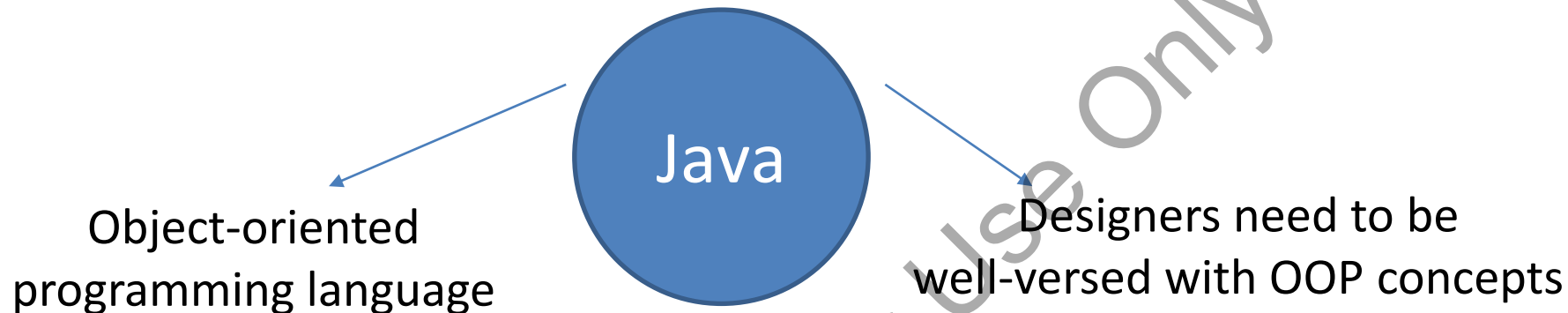
- ◆ Best practices to be followed when applying access control to Java classes, variables, and methods include:

Declare member variables and methods as private.

Declare only those methods required to create objects as public.

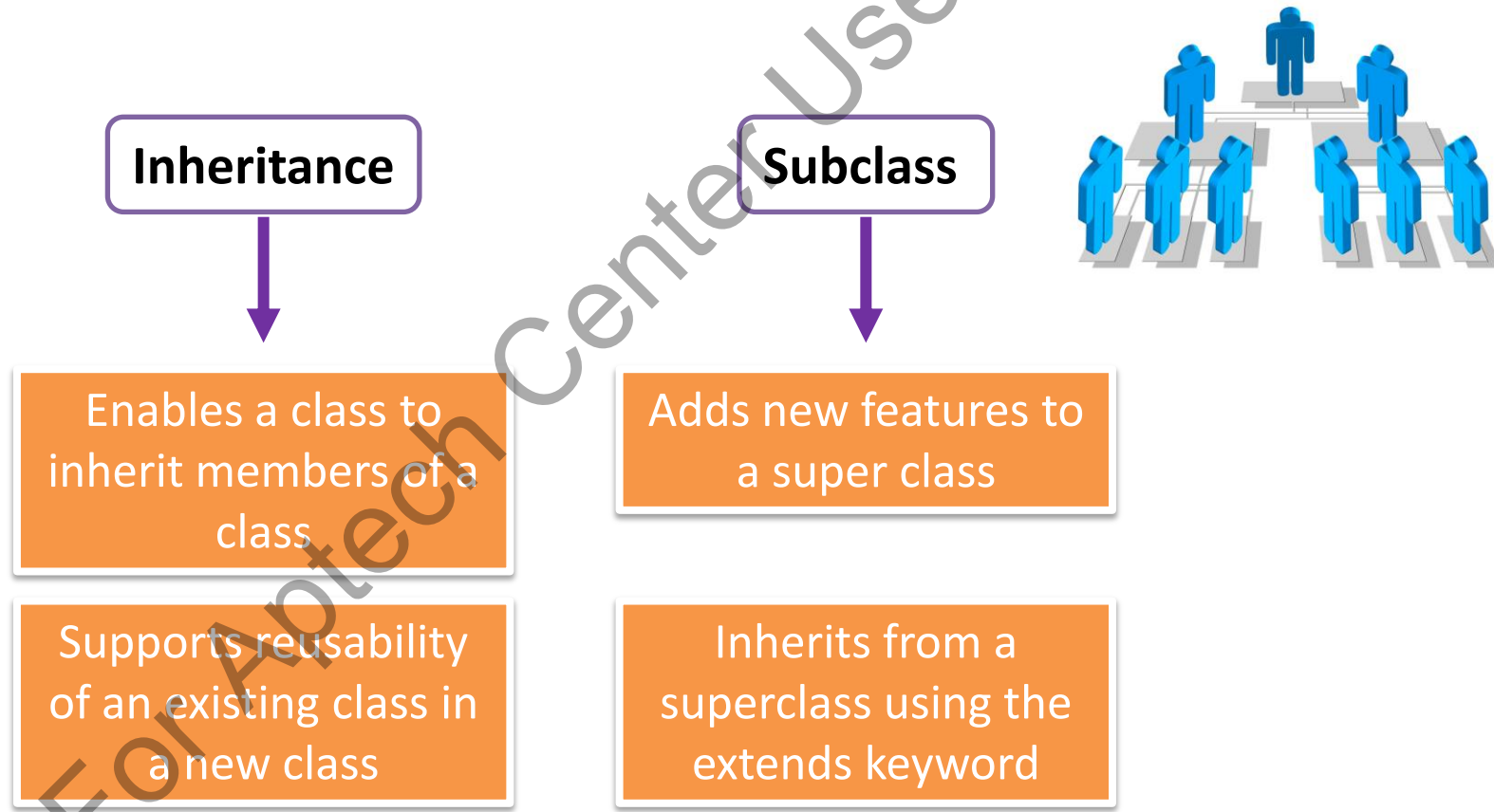
Do not declare member variables as public, except for constants.

Declare variables and methods as protected if there is chance that a subclass might need them in the future.





- ◆ In Java, there are various classes which can be derived or inherited.



- ◆ Java supports multi-level inheritance among classes.



- ◆ The code shows an example of multi-level inheritance hierarchy.

Code Snippet

```
package com.classdesign.demo;  
public class Movie {  
    String language="English";  
    String type="Full length movie";  
    void getMovie() {  
        System.out.println("Language "+ language);  
        System.out.println("Type: "+ type);  
    }  
}
```

For Aptech Center Use Only

Multi-level Inheritance [3-4]



- ◆ The code shows an `ActionMovie` class that extends `Movie`, forming an inheritance hierarchy.

Code Snippet

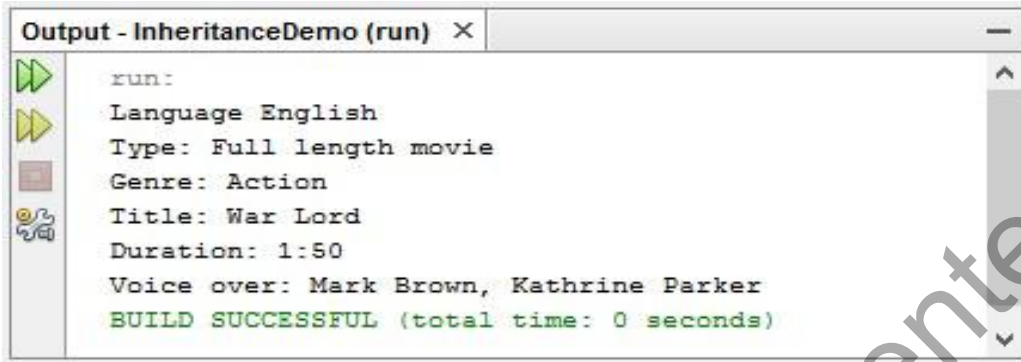
```
package com.classdesign.demo;

public class ActionMovie extends Movie {
    String title = "War Lord";
    String duration = "1:50";
    String genre="Action";
    void getActionMovie() {
        System.out.println("Genre: " + genre);
        System.out.println("Title: " + title);
        System.out.println("Duration: " + duration);
    }
}
```

Multi-level Inheritance [4-4]



Following is the output of the code:

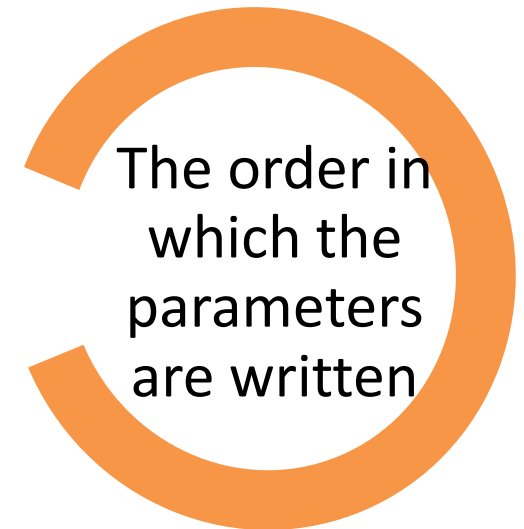
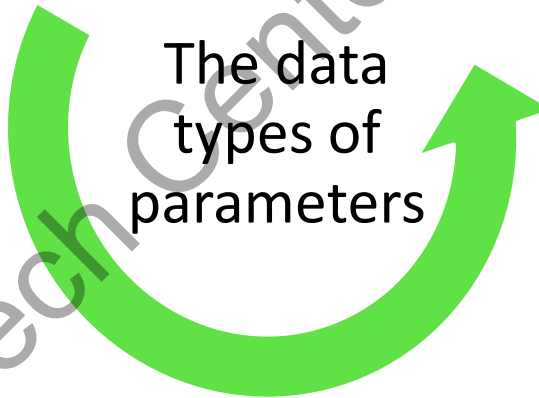
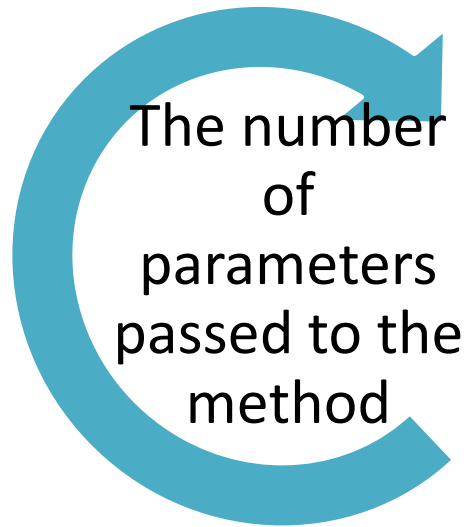


```
run:
Language: English
Type: Full length movie
Genre: Action
Title: War Lord
Duration: 1:50
Voice over: Mark Brown, Kathrine Parker
BUILD SUCCESSFUL (total time: 0 seconds)
```

For Aptech Center Use Only



In object-oriented programming, every method has a signature which comprises:



For Aptech Center Use Only



- ◆ The code overloads the `add()` method to calculate the square of the `int` and `float` values passed as parameters.

Code Snippet

```
package com.classdesign.demo;
public class MethodOverloadingDemo {
    static int add(int intNum1, int Num2) {
        return intNum1 + Num2;
    }
    static float add(float floatNum1, float floatNum2) {
        return floatNum1 + floatNum2;
    }
    public static void main(String[] args) {
        System.out.println("Sum of int value: " + add(5, 15));
        System.out.println("Sum of float value: " + add(5.95f,
            15.30f));
    }
}
```

Method Overloading [3-4]



Following is the output of the code:

A screenshot of an IDE's output window titled "Output - MethodOverloadingDemo (run)". The window contains the following text:

```
run:  
Sum of int value: 20  
Sum of float value: 21.25  
BUILD SUCCESSFUL (total time: 1 second)
```

The window has a standard toolbar on the left with icons for running, stepping through, and other debugging actions, and a scrollbar on the right.

For Aptech Center Use Only



- ◆ **Example:** The built-in `abs()` method of the `Math` class present in the `java.lang` package

Returns the absolute value of the numeric parameter passed to it.

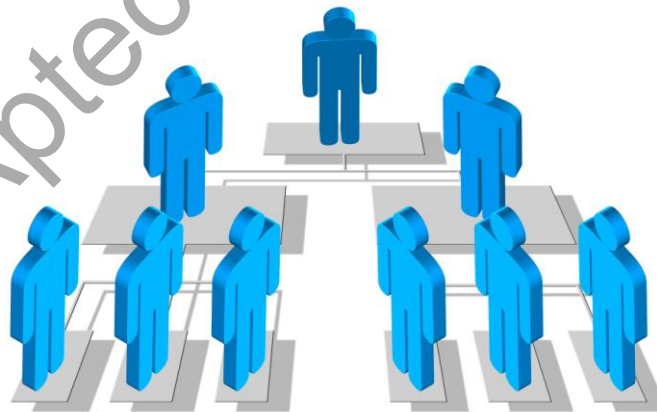
Has several overloaded versions such as `abs(int num)`, `abs(float num)`, and `abs(double num)`.

Java determines the correct overloaded `abs()` method to invoke.



Method overriding is:

- ◆ Process of creating a method in the subclass that has the same return type and signature as a method defined in the superclass.
- ◆ A form of dynamic polymorphism.





- ◆ The code shows an example of method overriding to implement dynamic polymorphism.

Code Snippet

```
package com.classdesign.demo;
class Animal {
    void getMessage() {
        System.out.println("Message from Animal.");
    }
}
class Dog extends Animal {
    @Override
    void getMessage() {
        System.out.println("Bow-wow! Message from Dog.");
    }
}
```

Method Overriding [3-4]



```
class Cat extends Animal {
    @Override
    void getMessage() {
        System.out.println("Meow! Message from Cat.");
    }
}

public class MethodOverridingDemo {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();
        animal1.getMessage();
        animal2.getMessage();
    }
}
```

For Aptech Center Use Only

Method Overriding [4-4]



Following is the output of the code:

```
Output - MethodOverridingDemo (run) X
run:
Bow-wow! Message from Dog.
Meow! Message from Cat.
BUILD SUCCESSFUL (total time: 1 second)
```

For Aptech Center Use Only

super () Constructor Call and super Keyword [1-4]



- ◆ Syntax of `super ()` constructor call is as follows:

Syntax

```
super ();
```

Or

```
super (parameter list);
```

Note: `super ()` call must be the first statement inside the constructor making the call.

- ◆ Syntax of `super` keyword is as follows:

Syntax

```
super.<method_name>;
```

super () Constructor Call and super Keyword

[2-4]



- ◆ The code shows the use of the `super ()` constructor call and the `super` keyword.

Code Snippet

```
package com.classdesign.demo;
class SuperClass {
    public SuperClass() {
        System.out.println("Message from SuperClass default
        constructor.");
    }
    void print() {
        System.out.println("Message from SuperClass print().");
    }
}
public class SuperCallDemo extends SuperClass {
    public SuperCallDemo() {
        super();
    }
}
```

super () Constructor Call and super Keyword

[3-4]



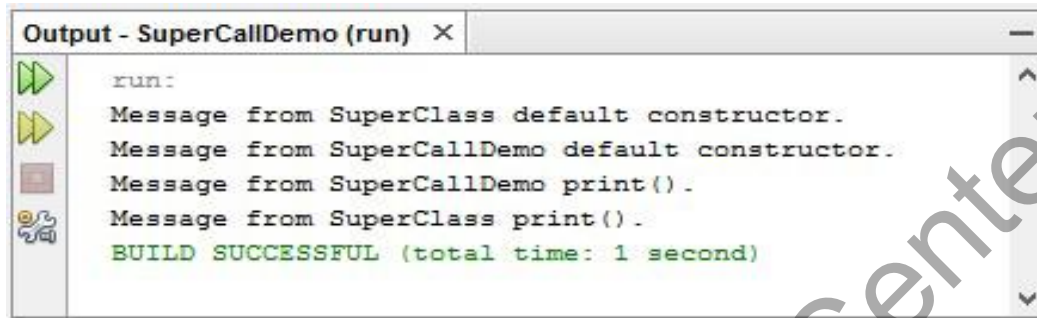
```
        System.out.println("Message from SuperCallDemo default  
        constructor.");  
    }  
    @Override  
    void print() {  
        System.out.println("Message from SuperCallDemo print().");  
        super.print();  
    }  
    public static void main(String[] args) {  
        SuperClass obj = new SuperCallDemo();  
        obj.print();  
    }  
}
```

For Aptech Center Use Only

super () Constructor Call and super Keyword [4-4]



Following is the output of the code:



```
run:
Message from SuperClass default constructor.
Message from SuperCallDemo default constructor.
Message from SuperCallDemo print().
Message from SuperClass print().
BUILD SUCCESSFUL (total time: 1 second)
```

For Aptech Center Use Only

Package and Import Statements [1-2]



Sydney in Australia



Sydney in Canada

OR



- ◆ To avoid conflicts due to identical class names, keep the classes in different packages.
- ◆ In Java, a **package** is used to group classes and interfaces logically and prevent name clashes between those with identical names.

Package and Import Statements [2-2]



`java.lang`

- Bundles the classes that are fundamental to the design of the Java programming language.

`java.io`

- Bundles the classes to perform input/output operations.

`java.collections`

- Bundles the classes and interfaces that are part of the Java collection framework.

`java.net`

- Bundles the classes to implement networking applications.

`java.time`

- Bundles the classes to work with date and time.



- ◆ A package in Java is declared with the **package** keyword.

Rules to declare a package

A package must be declared with class outside its namespace.

Name must match the directory structure where the corresponding bytecode resides.



- ◆ The code shows the use of a user-defined package for a Java class.

Code Snippet

```
package com.classdesign.demoA;  
public class ClassA {  
    public void getMessage(){  
        System.out.println("Message from ClassA in  
        com.classdesign.demoA package ");  
    }  
}
```

For Aptech Center Use Only



- ◆ The code shows a Java class with the same name as the class in the earlier code snippet, but in a different package.

Code Snippet

```
package com.classdesign.demoB;  
public class ClassA {  
    public void getMessage() {  
        System.out.println("Message from ClassA in  
        com.classdesign.demoB package ");  
    }  
}
```

For Aptech Center Use Only



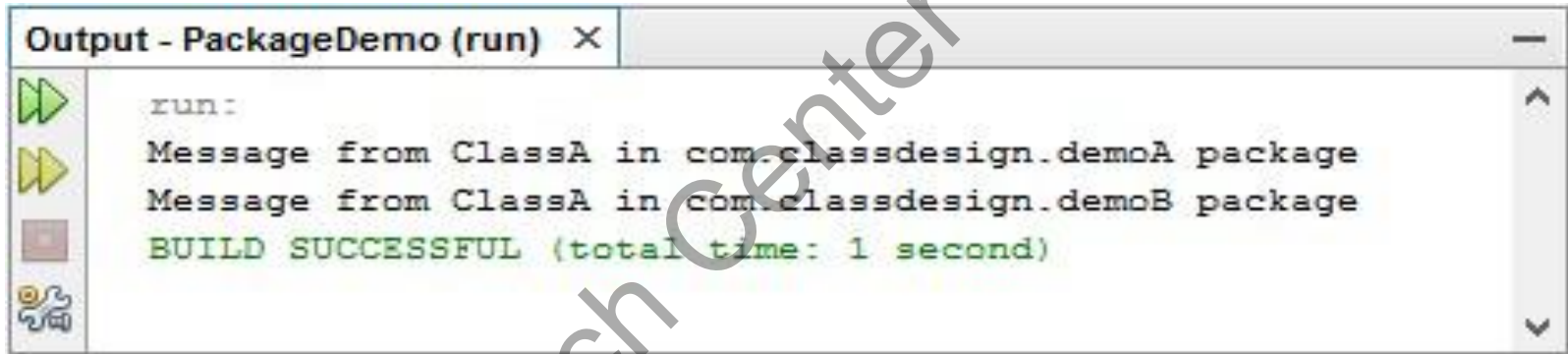
- ◆ The code shows a Java class using the two `ClassA` classes that are part of different packages.

Code Snippet

```
package com.classdesign.demo;  
public class PackageDemo {  
    public static void main(String[] args) {  
        com.classdesign.demoA.ClassA obj1 = new  
            com.classdesign.demoA.ClassA();  
        com.classdesign.demoB.ClassA obj2 = new  
            com.classdesign.demoB.ClassA();  
        obj1.getMessage();  
        obj2.getMessage();  
    }  
}
```



Following is the output of the code:



```
run:
Message from ClassA in com.classdesign.demoA package
Message from ClassA in com.classdesign.demoB package
BUILD SUCCESSFUL (total time: 1 second)
```



To avoid typing the fully qualified name each time a class needs to be used, Java enables **single-type import**.

Single-type import is importing a class once and thereafter, you can use the class with only the class name.



- ◆ The code shows importing the `ClassA` class of the `com.classdesign.demoA` package with a single-type import.

Code Snippet

```
package com.classdesign.demo;  
import com.classdesign.demoA.ClassA;  
public class PackageDemo {  
    public static void main(String[] args) {  
        ClassA obj1 = new ClassA();  
        obj1.getMessage();  
    }  
}
```

Note: Java does not allow importing classes with the same name in different packages through single type import.



- ◆ The code shows importing the entire `com.classdesign.demoA` package and using the `ClassA` class of that package.

Code Snippet

```
package com.classdesign.demo;  
import com.classdesign.demoA.*;  
public class PackageDemo {  
    public static void main(String[] args) {  
        ClassA obj1 = new ClassA();  
        obj1.getMessage();  
    }  
}
```

Note: You do not need to explicitly import the `java.lang` package as the compiler does it by default for all classes.



Abstract Class

Cannot be instantiated, but can only be sub-classed.

Contains one or more methods declared with the `abstract` keyword.

Does not contain implementation.

Used to declare classes that only define common properties and behavior of other classes.



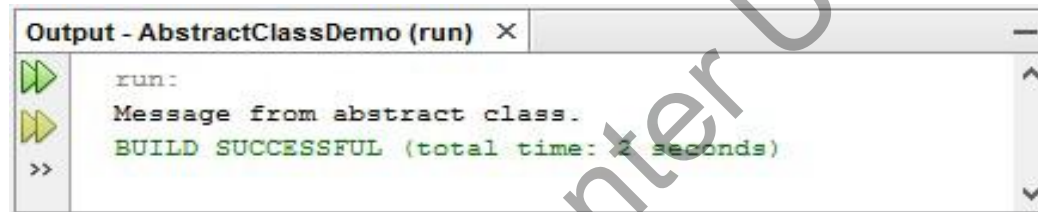
- ◆ The code shows how to create and use an abstract class.

Code Snippet

```
package com.classdesign.demo;
abstract class TestAbstract {
    abstract void getMessage();
}
public class AbstractClassDemo extends TestAbstract {
    void getMessage() {
        System.out.println("Message from abstract
        class.");
    }
    public static void main(String[] args) {
        TestAbstract abstractClass = new
        AbstractClassDemo();
        abstractClass.getMessage();
    }
}
```



Following is the output of the code:

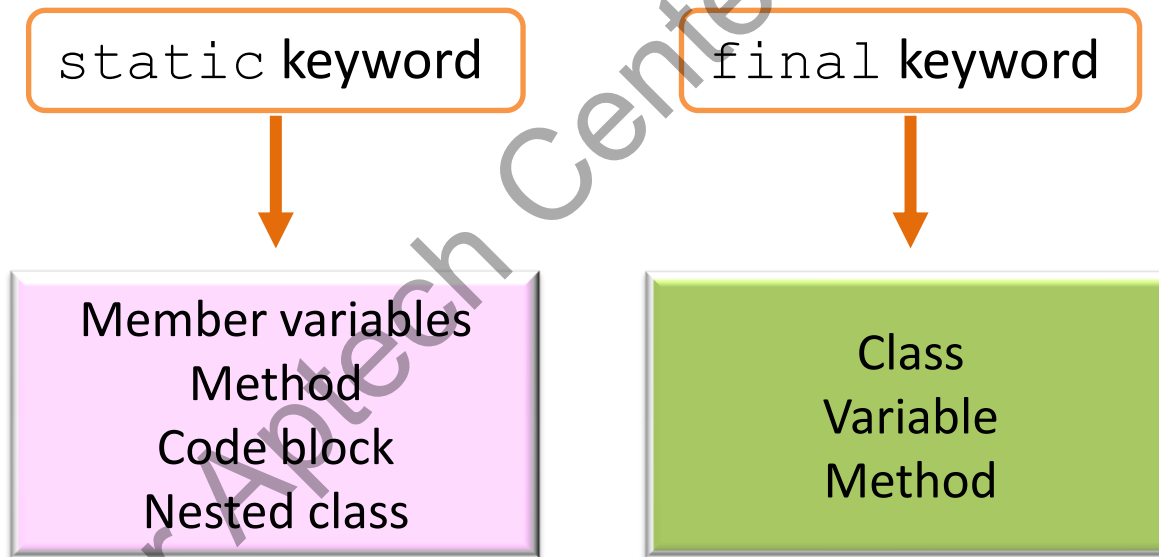


```
run:
Message from abstract class.
BUILD SUCCESSFUL (total time: 2 seconds)
```

For Aptech Center Use Only

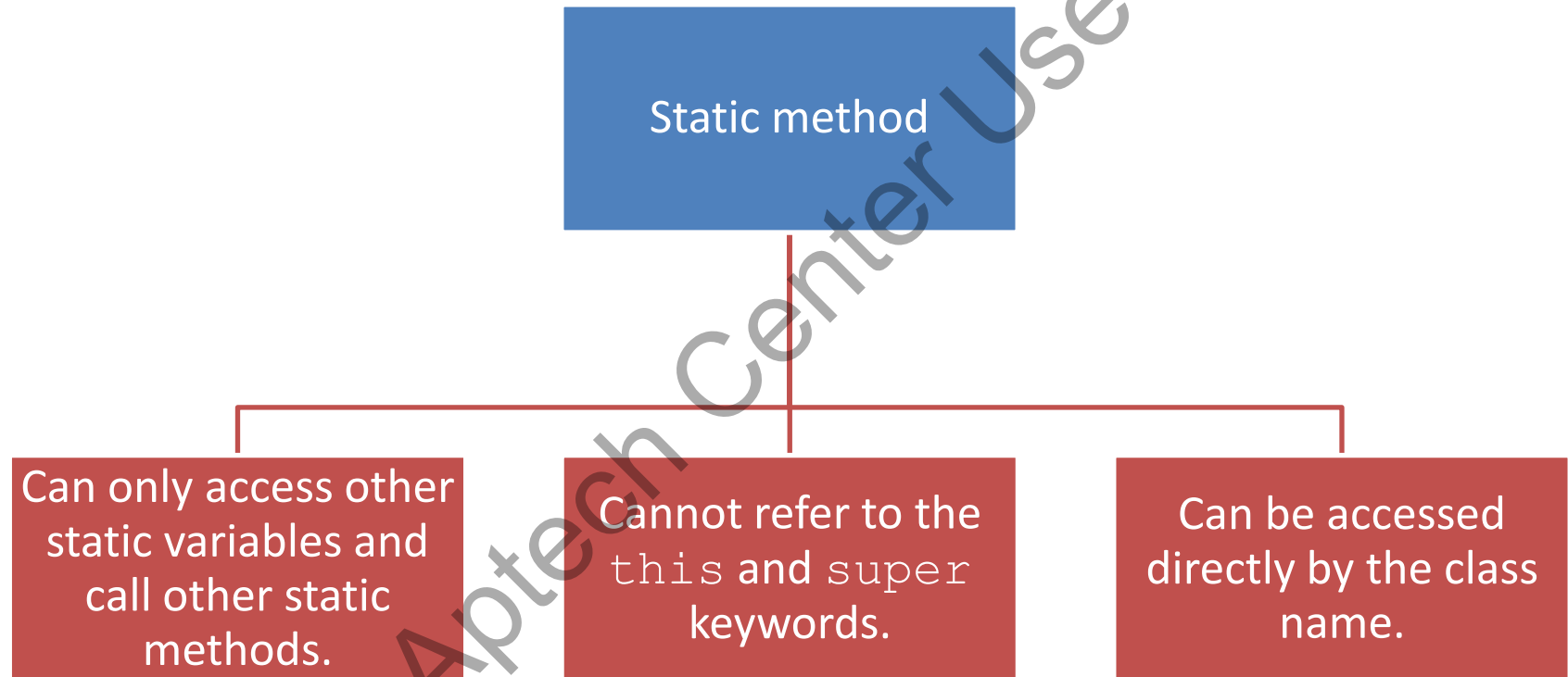


- ◆ In Java, `static` and `final` are two important keywords that are extensively used while writing Java code.





- ♦ Applying `static` keyword to methods of a class results in static methods.





- ◆ The code shows how to create and use a static variable.

Code Snippet

```
package com.classdesign.demo;  
public class StaticDemo {  
    static String message = "Hello! Message from static  
    variable";  
    public static void main(String[] args) {  
        String msg = StaticDemo.message;  
    }  
}
```

For Aptech Center Use Only



- ◆ The code shows how to create and use a static method.

Code Snippet

```
package com.classdesign.demo;  
public class StaticDemo {  
    private static String message = "Hello! Message  
    from static variable";  
    public static String getMessage() {  
        return StaticDemo.message;  
    }  
    public static void main(String[] args) {  
        String msg = StaticDemo.getMessage();  
        System.out.println(msg);  
    }  
}
```




◆ A static block:

- ◆ Is a normal block of code enclosed in curly braces { } and marked with the `static` keyword.
- ◆ Can be multiple static blocks inside the class body.

◆ For multiple static blocks:

- ◆ The blocks are called in the order that they appear in the class body.
- ◆ JVM executes the code of static blocks when it loads the class.
- ◆ JVM joins them into one single static block before executing it.

◆ Code inside a static block can refer static variables and methods.



- ◆ The code shows how to create and use a static block.

Code Snippet

```
package com.classdesign.demo;
public class StaticDemo {
    private static String message = "Hello! Message from
static variable";
    public static String getMessage(){
        return StaticDemo.message;
    }
    public static void main(String[] args) {
        String msg = StaticDemo.getMessage();
        System.out.println("Called from main: "+msg);
    }

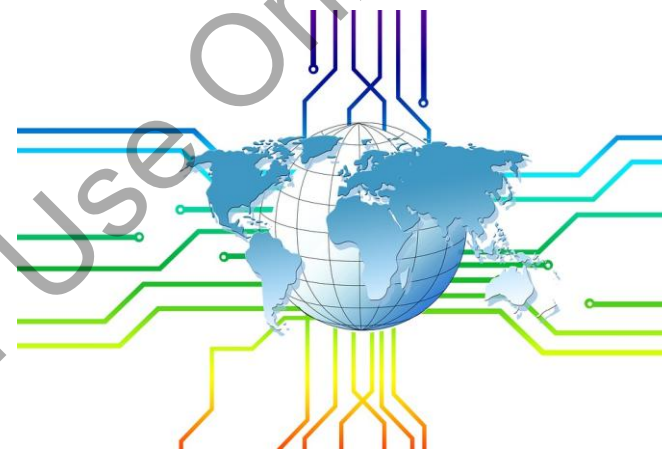
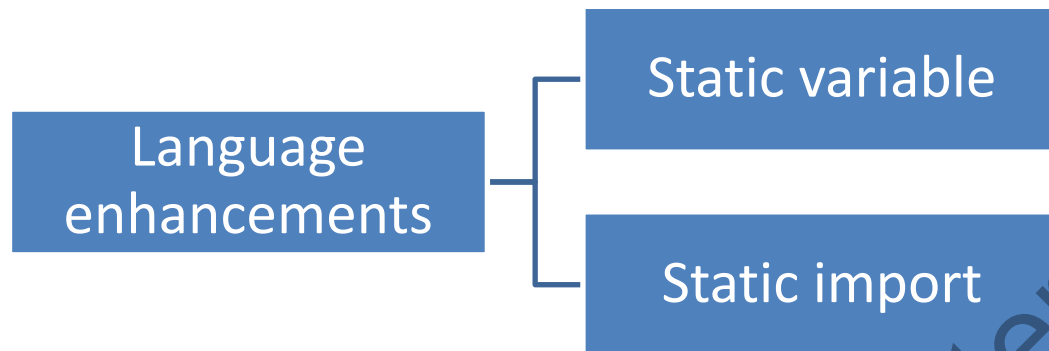
    static {
        System.out.println("Hello! Message from static
block");
        System.out.println("Called from static block: " +
StaticDemo.getMessage());
    }
}}
```



Following is the output of the code:



```
run:
Hello! Message from static block
Called from static block: Hello! Message from static variable
Called from main: Hello! Message from static variable
BUILD SUCCESSFUL (total time: 1 second)
```



Static import

- Allows importing static variables and methods of a class and use them, as they are declared in the same class.
- Can greatly reduce code size by allowing to use static variables and methods of another class without prefixing the class name.



- ◆ The code shows the use of static import.

Code Snippet

```
package com.classdesign.demo;
import static java.lang.Integer.MAX_VALUE;

public class StaticImportDemo {
    public static void main(String[] args) {
        /*Accessing static member with static import*/
        System.out.println(MAX_VALUE);

        /*Accessing static member without static import*/
        System.out.println(Integer.MAX_VALUE);
    }
}
```

Static Imports [3-4]



Following is the output of the code:

```
Output - StaticImportDemo (run) X
run:
2147483647
2147483647
BUILD SUCCESSFUL (total time: 1 second)
```

For Aptech Center Use Only



- ◆ Both the Integer and Long classes contain the static MAX_VALUE variable.
- ◆ With static import, it is possible to refer the variable directly as MAX_VALUE instead of Integer.MAX_VALUE, it is unclear of which class the variable belongs to.





Classes are declared final:

To prevent them from being sub-classed.

To confer security and efficiency benefits.

To standardize the behavior of a class by preventing it from being extended.





- ◆ The syntax for declaring a `final` class is as follows:

Syntax

```
<access_modifier> final <class_name>
```

- ◆ The code shows a final class.

Code Snippet

```
package com.classdesign.demo;  
    final class FinalClass{  
    }
```

For Aptech Center Use Only



- ◆ Variables and methods can also be declared as `final`.
- ◆ A variable declared as `final` cannot be assigned a different value.
- ◆ In Java, a constant that is used to map an exact and unchanging value to a variable name is declared as `final`.





- ◆ The syntax for declaring a `final` variable is as follows:

Syntax

```
<access_modifier> final <variable_name> = <value>;
```

- ◆ The syntax for declaring a `final` method is as follows:

Syntax

```
<access_modifier> final <return_type>  
<method_name> (<parameter_optional>)  
{ }
```

For Aptech Center Use Only



- ◆ The code shows a final variable and a final method.

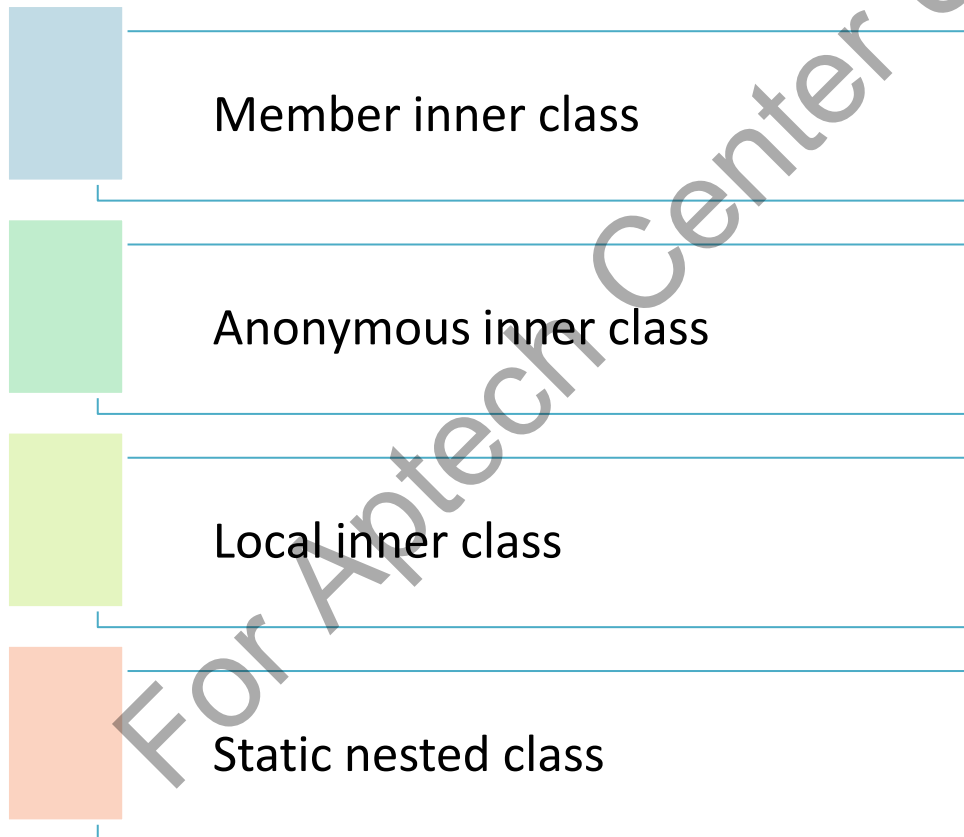
Code Snippet

```
package com.classdesign.demo;  
class FinalMemberDemo{  
    final int initialCount=10;  
    final void getMessage(){  
        System.out.println("Message from final method");  
    }  
}
```

For Aptech Center Use Only



- ◆ An inner or nested class is a class defined within the body of another class or interface.
- ◆ The different types of inner classes are:





- ◆ A non-static class defined inside a class but outside a method is known as a **member inner class**.
- ◆ The syntax for declaring a member inner class is as follows:

Syntax

```
class <outer_class>{  
    //code  
    class <inner_class>{  
        //code  
    }  
}
```

For Aptech Center Use Only



- ◆ The code shows the use of a member inner class.

Code Snippet

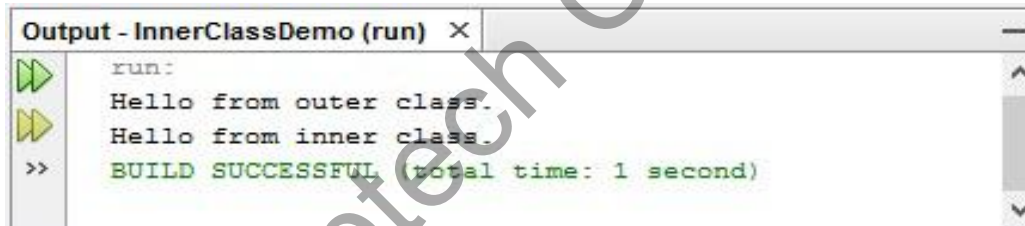
```
package com.classdesign.demo;
public class Outer {
    private String message="Hello from outer class.";
    class Inner{
        void getMessage() {
            System.out.println(message);
            System.out.println("Hello from inner class.");
        }
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.new Inner();
        inner.getMessage();
    }
}
```

Member Inner Class [3-3]



The code creates a top-level class with a member inner class. Observe that the member inner class is accessing the private variable of the enclosing outer class. The `main()` method creates an outer class instance and uses it to create an inner class instance. The `getMessage()` method is then called on the inner class instance.

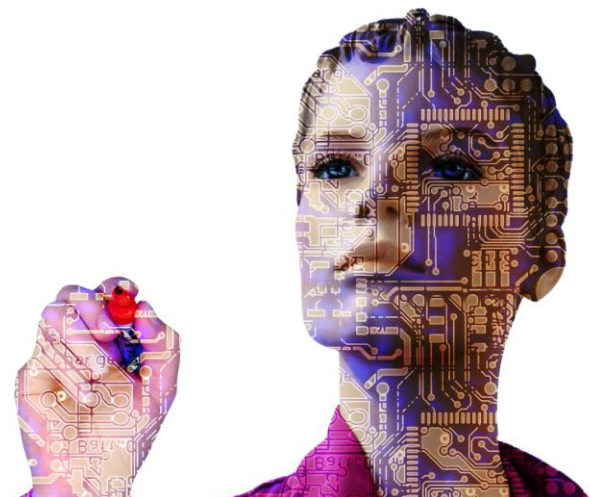
Following is the output of the code:



```
Output - InnerClassDemo (run) X
run:
Hello from outer class.
Hello from inner class.
BUILD SUCCESSFUL (total time: 1 second)
```




- ◆ An **anonymous inner class** is declared and instantiated at the same time.
- ◆ Inner classes are typically used to override the method of a concrete class, abstract class, or interface.





- ◆ The syntax of an anonymous inner class is as follows:

Syntax

```
<extended_type> <variable_name> = new < extended_type >() {  
    <access_modifier> <return_type> <>() {  
        .....  
        .....  
    }  
};
```

For Aptech Center Use Only

Anonymous Inner Class [3-4]



- ◆ The code shows the use of an anonymous inner class.

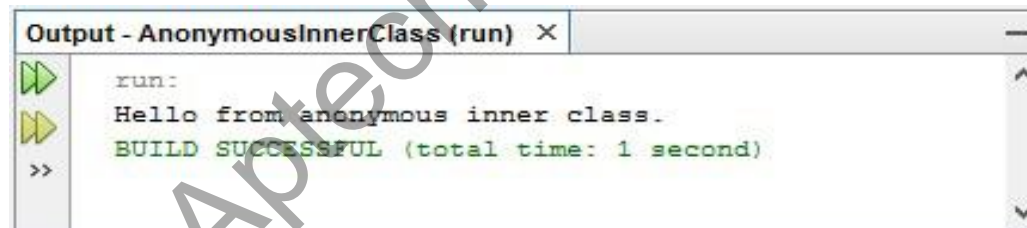
Code Snippet

```
package com.classdesign.demo;
abstract class Greet {
    abstract void getGreeting();
}
public class AnonymousInnerClass {
    public static void main(String[] args) {
        Greet g = new Greet()
        {
            void getGreeting() {
                System.out.println("Hello from anonymous
                Inner class.");
            }
        };
        g.getGreeting();
    }
}
```



The code creates an abstract `Greet` class with an abstract `getGreeting()` method. The `main()` method of the class, `AnonymousInnerClass` uses an anonymous inner class to instantiate a `Greet` object by overriding the `getGreeting()` method. Finally, the `getGreeting()` method is invoked on the `Greet` object named `g`.

Following is the output of the code:



```
Output - AnonymousInnerClass (run) X
run:
Hello from anonymous inner class.
BUILD SUCCESSFUL (total time: 1 second)
```



◆ A local inner class:

- is scoped within the method.
- Stops exiting once the method exits.
- Can be instantiated only from within the method where it is declared.





- ◆ The syntax of a local inner class is as follows:

Syntax

```
<access_modifier> <return_type> <method_name>() {  
    class <local_inner_name>{  
        //Code  
    }  
}
```

For Aptech Center Use Only



- ◆ The code shows the use of a local inner class.

Code Snippet

```
package com.classdesign.demo;

class LocalInnerClassDemo{
    void display(){
        String msg="Hello";
        class Local{
            void getMessage(){
                System.out.println("Message from local inner class: "+msg);
            }
        }
        Local l = new Local();
        l.getMessage();
    }
    public static void main(String args[]){
        LocalInnerClassDemo obj = new LocalInnerClassDemo();
        obj.display();
    }
}
```



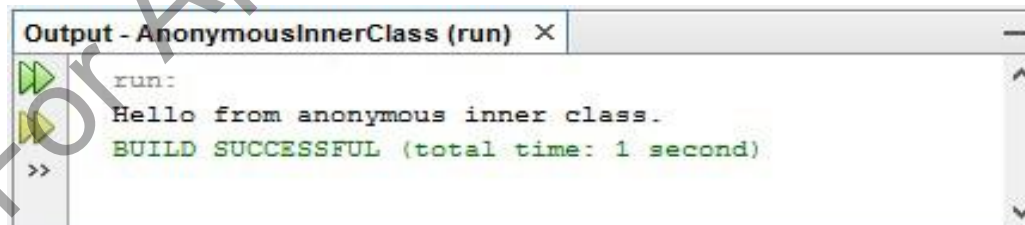
The code creates a local inner class named `Local` inside the `display()` method. The `display()` method instantiates the local inner class and invokes the `getMessage()` method on it.

In the `display()` method, observe that the local inner class refers to the `msg` local variable.



The ability to access non-final local variables has been introduced in Java 8.

Following is the output of the code:



```
Output - AnonymousInnerClass (run) X
run:
Hello from anonymous inner class.
BUILD SUCCESSFUL (total time: 1 second)
```




Static inner class

Similar to a member inner class.

Created before an instance of the outer class.

Accessed directly with the class name of the outer class.



- ◆ The syntax for declaring a static inner class is as follows:

Syntax

```
class <outer_class>{  
    //code  
    static class <inner_class>{  
        //code  
    }  
}
```

For Aptech Center Use Only



- ◆ The code shows the use of a static inner class.

Code Snippet

```
package com.classdesign.demo;
public class Outer {
    static String msg="Hello";
    static class StaticInner{
        static void display(){
            System.out.print("Message from static inner class:
                "+msg+"\n");
        }
    }
    public static void main(String[] args) {
        Outer.StaticInner.display();
    }
}
```

Static Inner Class [4-4]



Following is the output of the code:

A screenshot of an IDE's output window. The title bar reads "Output - StaticInnerClass (run) x". The window contains the following text: "run:", "Message from static inner class: Hello", and "BUILD SUCCESSFUL (total time: 1 second)". On the left side of the window, there are three green arrow icons and a ">>" symbol. On the right side, there is a vertical scrollbar.

```
run:
Message from static inner class: Hello
BUILD SUCCESSFUL (total time: 1 second)
```

For Aptech Center Use Only



- ◆ Java supports advanced types, such as the enum type and immutable classes.





A Java enum:

is a special type that defines a fixed set of constants.

can contain constructors, variables, and methods.

is treated as an advance type of class holding a set of constants.

while being compiled, has some special methods added the compiler automatically.

cannot be declared as final.

defined using the `enum` keyword.

cannot extend from another class.



An
immutable
class:

- is a class whose object's state cannot be changed once instantiated.
- is inherently thread safe.
- state cannot change once instantiated.
- can be easily shared or cached without the need to copy or clone them.

For Aptech Center Use Only



The class should be declared final to disable method overriding.

Mutable objects that the instance fields refer to should not change.

Steps for creating an immutable class

The class should not contain setter methods to modify fields or objects referred to by fields.

All fields should be declared private and final.



- ◆ The code shows an immutable class.

Code Snippet

```
package com.classdesign.demo;

public final class ImmutableClassDemo {
    private final String fName;
    private final String lName;
    public ImmutableClassDemo(final String fName, final String lName) {
        this.fName = fName;
        this.lName = lName;
    }
    public String getFName() {
        return fName;
    }
    public String getLName() {
        return lName;
    }
}
```



- ◆ The code creates an immutable class through the following design:
 - ◆ The class is declared as `final`.
 - ◆ The instance variables are declared as `private` and `final`.
 - ◆ The constructor parameters are declared as `final` to ensure that its value does not change.
 - ◆ No setter methods, such as `setFName()` and `setLName()` methods are provided.

For Aptech Center Use Only



- ◆ An access modifier controls the access of class members and variables by other objects.
- ◆ Inheritance enables a class to inherit variables and methods from another class.
- ◆ Polymorphism is the ability of different object types to respond to the same message, each one in its own way.
- ◆ In Java, a package is used to group classes and interfaces logically and prevent name clashes between those with identical names.
- ◆ An abstract class contains one or more methods declared with the abstract keyword.
- ◆ An inner class without a class name is known as an anonymous inner class.