Chapter: 10

**Lambda Expression, Method referencing and Modules**

Lambda Expression, Method/Constructor referencing, Functional interfaces, Modules and Services

|  |  |
| --- | --- |
| **10.1 Introduction to LAMBDA Expressions ("LE")** | In much the same way that the addition of generics reshaped Java years ago, lambda expressions are reshaping Java today. |

Two important results of LAMBDA expression are—the default method (define default behavior for an interface method) and the method reference (refer to a method without executing it). For its usefulness lambda expressions have been added to C# and C++.

* LAMBDA expression: A lambda expression is an ***anonymous*** (i.e., ***unnamed***) method. However, this method is *not executed on its own*. Instead, it is ***used to implement*** a method defined by a functional interface. ***Lambda expressions*** commonly called closures.
* Thus, a *lambda expression* results *in a form of* *anonymous class*.
* A functional interface defines the ***target type*** of a lambda expression. A lambda expression can be used only in a context in which a ***target type*** is specified.
* FUNCTIONAL interface: A functional interface is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. i.e. a functional interface typically ***represents a single action***. A *functional* *interface* is sometimes referred to as a SAM type, where ***SAM*** stands for ***Single Abstract Method***.
* Example: the standard interface ***Runnable*** is a ***functional interface*** because it defines only one method: ***run()***. Therefore, ***run()*** defines the action of ***Runnable***.
* Implicit members of a functional interface: A ***functional interface*** may specify any public *method* defined by ***Object***, such as ***equals()***, without affecting its *“functional interface”* *status*. The public ***Object*** methods are considered implicit members of a ***functional interface*** because they are *automatically implemented by an* instance of a ***functional interface***.
* Lambda Operator and Lambda Expression: The operator **->** operates the *Lambda expression*, it is referred to as the ***lambda*** ***operator*** or the ***arrow*** ***operator***. It divides a lambda expression into two parts.
* The *left side* specifies any *parameters* required by the *lambda expression*.
* On the *right side* is the *lambda body*, which specifies the actions of the *lambda expression*. Java defines two types of *lambda* *bodies*. One type consists of a single expression, and the other type consists of a block of code.
* Of course, ***the*** method ***defined by a*** lambdaexpression ***does not have a*** name.
* A Lambda expression *may or may not have* parameters. When a *lambda* expression requires a *parameter*, it is specified in the *parameter list* on the *left side* of the " **->** ".
* Type of the parameter of a Lambda expression: You won’t need to *explicitly* specify the *type* of a LE because, in many cases, its type can be inferred. However, it is possible to *explicitly* specify the *type* of a parameter, Like a named method, a *lambda expression* can specify as many parameters as needed. Any *valid type* can be used as the *return type* of a *lambda expression*. For example,
* **(n) -> (n % 2)==0**  this *lambda expression* returns ***true*** if the value of parameter ***n*** is even and ***false*** otherwise.
* Thus (automatically), the return type of this lambda expression is ***Boolean***.
* Note: When a lambda expression has *only one parameter*, it is ***not necessary to surround the parameter name with parentheses*** when it is specified on the left side of the lambda operator. For example: ***n -> (n % 2)==0***
* **(n) -> 1.0 / n**  this *lambda expression* returns the reciprocal of the value of parameter ***n***.
* Example 1: Following lambda expressions have no parmeters:
* **() -> 98.6** This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value 98.6. The return type is inferred to be double. It is similar to this method: ***double myMeth() { return 98.6; }***
* **() -> Math.random() \* 100** This lambda expression obtains a pseudo-random value from ***Math.random()***, multiplies it by ***100***, and returns the result. It, too, does not require a parameter.

**10.2 Functional Interfaces ("FI")**

Recall from Java/C# 4.6, 5.12 and 5.13 that *not all interface methods are abstract*. Beginning with JDK 8, it is possible for an *interface* to have *one or more* default methods. ***Default methods*** are not ***abstract***. Neither are ***static interface methods***. Thus, *an interface method is abstract only if it is does not specify an implementation*.

* Therefore a functional interface can include default and/or static methods, but in all cases it must have *one and only one abstract method*. [Because non-default, non-static interface methods are implicitly abstract, there is no need to use the abstract modifier]

**10.2.1 Lambda Expression with no Parameters**

* Here is an example of a *functional interface :*  **interface** MyValue { **double** getValue(); }
* In this case, the method ***getValue()*** is *implicitly abstract*, and it is the only method defined by ***MyValue***. Thus, ***MyValue*** is a *functional interface*, and its function is defined by ***getValue( )***.
* As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the ***implementation of the abstract method*** defined by the ***functional interface*** that specifies its ***target type***.
* As a result, ***a*** lambda expression ***can be specified only in a*** context ***in which a*** target type ***is defined***. One of these contexts is created when a *lambda expression* is assigned to a *functional interface reference*.

[ Other ***target type contexts*** include ***variable initialization***, ***return statements***, and ***method arguments***, to name a few. ]

* Initializing an interface reference by lambda expression: Consider a *reference* to the *functional interface* ***MyValue*** is declared:

**MyValue** myVal; *// Create a reference to a MyValue instance.*

* Next, a *lambda expression* is assigned to that *interface reference*:

myVal = **()** **->** 98.6; *// Use a lambda in an assignment context.*

This lambda expression is compatible with getValue( ) because, like getValue( ), it has *no parameters* & returns a *double result*.

* In general, the type of the abstract method defined by the ***functional interface*** and the type of the ***lambda expression*** must be compatible. If they aren’t, a *compile-time error* will result.
* Initialization: The above two statements can be combined into a *single* statement: **MyValue** myVal = **()** **->** 98.6;

Here, ***myVal*** is *initialized* with the *lambda expression*.

* Lambda expression turn a code segment into an object: When a LE occurs in a *target type context*, an ***instance*** *of a class is automatically created* that implements the *functional interface, with the* LE defining the behavior of the *abstract**method* ***declared by the*** *functional**interface*. *When that* method *is called through the* target*, the* ***lambda expression*** *is* executed.
* In the preceding example, the LE becomes the implementation for the ***getValue()***. As a result, the following displays ***98.6***:

**System.out.println**("*A constant value:* " + ***myVal.getValue()***); *// Call getValue(), which is implemented by the previously assigned* ***LE****.*

* Because the LE assigned to ***myVal*** returns the value ***98.6***, that is the value obtained when ***getValue()*** is called.

**10.2.2 Parameterized Lambda Expression**

* If the LE takes one or more parameters, then the *abstract method* in the FI must also take the ***same number of parameters***. For example, here is a FI called ***MyParamValue***, which lets you pass a value to ***getValue()***:

**interface** MyParamValue{ **double** getValue(**double** v); }

* You can use this *interface* to implement the *reciprocal lambda* shown in the previous section:

**MyParamValue** myPval = (n) -> 1.0 / n;

**System.out.println**("*Reciprocal of 4 is* " + ***myPval.getValue(4.0)***); *// use myPval to display value*

* Here, ***getValue()*** is implemented by the LE referred to by ***myPval***, which returns the reciprocal of the argument.
* Notice that the type of ***n*** is not specified. Rather, its type is inferred from the context (i.e. method in the ***MyParamValue*** interface). In this case, its type is inferred from the parameter type of ***getValue()*** as defined by the ***MyParamValue*** interface, which is ***double***.
* It is also possible to explicitly specify the type of a parameter in a lambda expression : **(double** n**)** **->** 1.0 / n;
* Here, ***n*** is *explicitly specified* as ***double***. Usually it is not necessary to explicitly specify the *type*.
* For a LE to be used in a target type context, the type and number of the LE's *parameters* must be compatible with the FI*'*s ***abstract******method****’s* *parameters* and its *return* *type*. For example, if the ***abstract method*** specifies two **int** parameters, then the ***lambda*** must specify two parameters whose type either is *explicitly* ***int*** or can be *implicitly* *inferred* as ***int*** by the context.
* Example 2: The first example assembles the pieces shown in the foregoing section into a complete program.

|  |  |
| --- | --- |
| **interface** MyValue { **double** getValue(); } *//* ***FI*** *without parameter*  **interface** MyParamValue { **double** getValue(**double** v); } *// Parameterized* ***FI***  **class** LambdaDemo { **public static void main(String args[])** {  **MyValue** myVal; *// declare an interface reference*  myVal = **() ->** 98.6; | Output:  **A constant value: 98.6**  **Reciprocal of 4 is 0.25**  **Reciprocal of 8 is 0.125** |

**System.out.println**("A constant value: " + myVal.getValue()); *// Call getValue(), which is provided by the previously assigned LE.*

**MyParamValue** myPval = (n) -> 1.0 / n; *// parameterized LE and assign it to a MyParamValue reference.*

**System.out.println**("Reciprocal of 4 is " + myPval.getValue(4.0)); *// Call getValue(v) through the myPval reference.*

**System.out.println**("Reciprocal of 8 is " + myPval.getValue(8.0)); *// Call getValue(v) through the myPval reference.*

*// A lambda expression must be compatible with the method defined by the functional interface. Therefore, following won't work:*

*//* myVal = **() ->** "three"; *// Error! String not compatible with double!*

*//* myPval = **() -> Math.random**(); *// Error! Parameter required!*

}}

* Here, the LE ***myVal = () -> 98.6;*** is simply a *constant expression*. When it is assigned to ***myVal***, a class instance is constructed in which the lambda expression implements the ***getValue()*** method in ***MyValue***.
* The LE must be *compatible with the* ***abstract method*** *that it is intended to implement*. So, the commented-out lines are illegal.
* The first, because a value of type Stringis not compatible with double, which is the return type required by ***getValue()***.
* The second, because ***getValue(int)*** in ***MyParamValue*** requires a parameter, and one is not provided.
* Example 3: A FI can be used with any LE as long as it is compatible with the LE. Following use the same **FI** with 3 different **LE**s:
* It defines a FI called ***NumericTest*** that declares the abstract method ***test()***.***test()*** has two ***int*** parameters and returns a ***boolean*** result. Its purpose is to determine if the *two arguments* passed to ***test()*** satisfy *some condition*.

|  |  |  |
| --- | --- | --- |
| * In ***main()***, *three different tests* are created through the use of lambda expressions. | | |
| * One tests if the first argument can be evenly divided by the second; | * the second determines if the first argument is less than the second; | * the third returns true if the absolute values of the arguments are equal. |

* Notice that the LE that implement these tests have *two parameters* and return a ***boolean*** result. This is, of course, necessary since ***test()*** has *two parameters* and returns a ***boolean*** result.

**interface** NumericTest { **boolean** test(**int** n, **int** m); } *// A functional interface that takes two int parameters and returns a boolean result.*

**class** LambdaDemo2 { **public static void main(String args[])** { *// Following lambda expression determines if one number is a factor of another.*

**NumericTest** isFactor = (n, d) -> (n % d) == 0; *// defining first version of test() via the abstract method*

**if**(isFactor.test(10, 2)) **System.out.println**("2 is a factor of 10");

**if**(!isFactor.test(10, 3)) **System.out.println**("3 is not a factor of 10");

**System.out.println**();

|  |  |
| --- | --- |
| *// This lambda expression returns true if the first argument is less than the second.*  **NumericTest** lessThan = (n, m) -> (n < m); *// defining second version of test() via the abstract method*  **if**(lessThan.test(2, 10)) **System.out.println**("2 is less than 10");  **if**(!lessThan.test(10, 2)) **System.out.println**("10 is not less than 2");  **System.out.println**(); | **Use the same functional interface with three different lambda expressions** |

*// Following lambda expression returns true if the absolute values of the arguments are equal.*

**NumericTest** absEqual = (n, m) -> (n < 0 ? -n : n) == (m < 0 ? -m : m); *// defining third version of test() via the abstract method*

**if**(absEqual.test(4, -4)) **System.out.println**("Absolute values of 4 and -4 are equal.");

**if**(!lessThan.test(4, -5)) **System.out.println**("Absolute values of 4 and -5 are not equal.");

**System.out.println**(); }}

* Because all three LEs are compatible with ***test()***, all can be executed through a ***NumericTest*** reference.
* EMBEDDING LAMBDAS: In this Example there is no need to use three separate ***NumericTest*** reference variables because the ***same one could have been used for all three tests()***. Eg: Create the variable ***myTest*** and then use it to refer to each ***test()***, in turn, as :

|  |  |
| --- | --- |
| **NumericTest** myTest;  myTest = **(**n, d**)** **->** (n % d) == 0;  ***if****(myTest.test(10, 2))* ***System.out.println****("2 is a factor of 10");*  *// ...*  myTest = **(**n, m**)** **->** (n < m);  ***if****(myTest.test(2, 10))* ***System.out.println****("2 is less than 10");*  *//...*  myTest = **(**n, m**)** **->** (n < 0 ? -n : n) == (m < 0 ? -m : m);  ***if****(myTest.test(4, -4))* ***System.out.println****("Absolute values of 4 and -4 are equal.");*  *// ...* | * Of course, using different reference variables called ***isFactor***, ***lessThan***, and ***absEqual***, as the original program does, makes it very clear to which LE each variable refers. So ***do not embed lambdas wrongfully***. |

* Whenever ***more than one parameter*** is required, the ***parameters are specified, separated by commas***, in a *parenthesized list* on the left side of the lambda operator. Eg: factor test: ***(n, d) -> (n % d) == 0*** here ***n*** are ***d*** are separated by "***,***".
* We used *primitive* values as the ***parameter types*** and ***return type*** of the ***abstract method*** defined by a FI. However, we can use *object type* values also.
* Example 4: Following declares a FI called ***StringTest***. It has a method called ***test()*** that takes two ***String*** parameters and returns a ***boolean*** result.

**interface** StringTest{ **boolean** test(**String** aStr, **String** bStr); } *// A functional interface that tests two strings.*

**class** LambdaDemo3{ **public static void main(String args[])** {

**StringTest** isIn = **(**a, b**)** **->** a.**indexOf**(b) != -1; *// This lambda expression determines if one string is part of another.*

**String** str = "This is a test";

**System.out.println**("Testing string: " + str);

**if**(isIn.test(str, "is a")) **System.out.println**(" 'is a' found.");

/\* . . . \*/ }}

* Notice, LE uses the ***indexOf()*** method defined by the ***String*** class to determine ***if one string is part of another***. It works because the parameters ***a*** and ***b*** are determined by *type inference* to be of type ***String***. Thus, it is permissible to call a ***String*** method on ***a***.

|  |  |
| --- | --- |
| Note | In cases in which a LE requires ***two or more parameters***, and if we need to *explicitly declare the type of a parameter*, then ***all of the parameters in the list must have declared types (no type inference allowed for other parameters)***.   * For example, this is legal: ***(int n, int d) -> (n % d) == 0***.   But this is not legal: ***(int n, d) -> (n % d) == 0***.This is also illegal: ***(n, int d) -> (n % d) == 0*** |

**10.3 Block Lambda Expressions**

* Expression lambdas (single expression): The lambdas consist of a *single expression* are sometimes called expression lambdas. These types of *lambda bodies* are referred to as expression bodies. In an ***expression body***, the code on the *right side* of the ***lambda operator*** must consist of a *single expression*, which becomes the *lambda’s value*.
* Block lambdas (expression block): The type of *lambda expression* in which the code on the *right side* of the ***lambda operator*** consists of a *block of code* that can contain *more than one statement*. This type of *lambda body* is called a block body. Lambdas that have *block bodies* are sometimes referred to as block lambdas.
* In a block lambda you can declare variables, use loops, specify if and switch statements, create nested blocks, and so on.
* A block lambda is easy to create. Simply *enclose the body* within *braces* as you would any other block of statements.
* You must explicitly use a ***return*** statement to return a value. This is necessary because a block lambda body does not represent a single expression.
* Example 5: Following uses a block lambda to find the ***smallest positive factor*** of an **int** value. It uses an interface called NumericFunc that has a method ***func()***, which takes one int argument and returns an int result.

**interface** NumericFunc { **int** func(**int** n); }

**class** BlockLambdaDemo { **public static void main(String args[])** {

*// Following block lambda returns the smallest positive factor of a value.*

**NumericFunc** smallestF = **(**n**)** **->**  { **int** result = 1;

n = n < 0 ? -n : n; *// Get absolute value of n. Ternary operator ? used*

**for**(**int** i=2; i <= n/i; i++) **if**((n % i) == 0) { result = i; **break**; }

**return** result; }; *// Block lambda expression ends*

**System.out.println**("Smallest factor of 12 is " + smallestF.func(12));

**System.out.println**("Smallest factor of 11 is " + smallestF.func(11)); }}

* In the program, notice that the *block lambda* declares a variable called ***result***, uses a ***for*** loop, and has a ***return*** statement.
* When a return statement occurs within a lambda expression, it simply causes a return from the lambda. It does not cause an enclosing method to return. The *block body* of a lambda is *similar to a method body*.

**10.4 Generic Functional Interfaces**

A LE, itself, cannot specify type parameters, hence cannot be generic. *(Of course, because of type inference, all lambda expressions exhibit some “generic-like” qualities.)* However, the FI associated with a LE can be *generic*. The target type of the LE is determined, in part, by the *type argument(s)* specified when a FI reference is declared.

* Example 6: To understand the value of generic FI, now we combine ***NumericTest*** (*Example 3*) and ***StringTest*** (*Example 4*) together using type parameter in the generic FI ***SomeTest<T>***.

**interface** SomeTest<T> { **boolean** test(**T** n, **T** m); } *// A generic FI with two parameters that returns a boolean result.*

**class** GenericFunctionalInterfaceDemo { **public static void main(String args[])** {

**SomeTest<Integer>** isFactor = **(**n, d**)** **->** (n % d) == 0; *// This LE determines if one integer is a factor of another.*

***/\* . . . \*/***

**SomeTest<Double>** isFactorD = **(**n, d**) ->** (n % d) == 0; *// This LE determines if one Double is a factor of another.*

***/\* . . . \*/***

**SomeTest<String>** isIn = **(**a, b**) ->** a.indexOf(b) != -1; *// This LE determines if one string is part of another.*

***/\* . . . \*/*** }}

* The generic FI ***SomeTest*** is declared as: ***interface SomeTest<T> { boolean test(T n, T m); }***
* Here, ***T*** specifies the type of both parameters for ***test()***. i.e. It is compatible with any LE with that takes *two parameters* of the *same type* and returns a *boolean result*.
* The ***SomeTest*** interface is used to provide a reference to ***three different types of lambdas***. The first uses type ***Integer***, the second uses type ***Double***, and the third uses type ***String***.

**10.5 Pass an LE as an Argument**

A lambda expression can be used in *any context* that provides a *target type*. The *target contexts* used by the preceding examples are assignment and initialization. Another one is when a lambda expression is passed as an argument. It gives you a way to pass executable code as an argument to a method.

* Example 7: To illustrate the process of passing an LE as an argument, we create three *string functions* that perform the operations:

|  |  |
| --- | --- |
| 1. reverse a string, 2. reverse the case of letters within a string, 3. replace spaces with hyphens. | * These functions are implemented as LEs of the FI ***StringFunc***. * They are then passed as the first argument to a method called changeStr(). This method applies the string function to the string passed as the second argument to changeStr() and returns the result. Thus, changeStr( ) can be used to apply a variety of different string functions. |

|  |  |
| --- | --- |
| **interface** StringFunc { **String** func(**String** str); } *// Functional Interface*  **class** LambdaArgumentDemo {  **static** **String** changeStr(**StringFunc** sf, **String** s) { **return** sf.func(s); }  **public static void main(String args[]) { String** inStr = "***LE Expand Java***";  **String** outStr;  **System.out.println**("***Here is input string:*** " + inStr);  *// Folowing LE reverses the contents of a string and assign it to a StringFunc reference variable.*  **StringFunc** reverse = (str) -> { **String** result = "";  **for**(int i = str.**length**()-1; i >= 0; i--) result += str.**charAt**(i);  **return** result; };  *// Pass reverse lambda to the first argument to changeStr(). and the input string as the 2nd argument.*  outStr = changeStr(reverse, inStr);  **System.out.println**("***The string reversed:*** " + outStr); | */\* Following* ***LE*** *replaces spaces with hyphens. And LE directly passed as an argument to changeStr(). \*/*  outStr = changeStr((str) -> str.**replace**(' ', '-'), inStr);  **System.out.println**("*Spaces replaced string:* " + outStr);  */\* This* ***block lambda*** *inverts the case of the string characters. This bock is also passed as an argument directly to changeStr(). \*/*  outStr = changeStr (  (str) -> {  **String** result = ""; char ch;  **for**(**int** i = 0; i < str.**length**(); i++ ) {  ch = str.**charAt**(i);  **if**(**Character**.**isUpperCase**(ch))  result += **Character.toLowerCase**(ch);  **else** result += **Character.toUpperCase**(ch); }  **return** result; } , inStr );  **System.out.println**("*Reversed case string:* " + outStr); } } |

* The FI - ***StringFunc*** : ***interface StringFunc { String func(String str); }*** defines the method ***func()***, which takes a ***String*** argument and returns a ***String***. Thus, ***func()*** can act on a string and return the result.
* The ***LambdaArgumentDemo*** class, defines the ***changeStr()*** method. ***changeStr()*** has two parameters: The first is FI's ***StringFunc*** type. i.e. it can be passed a reference to any ***StringFunc*** instance including an instance created by a LE. The second is **s**, ***string*** to be acted on is passed to **s**. The resulting ***string*** is returned.

**class** LambdaArgumentDemo{

**static String** changeStr(**StringFunc** sf, **String** s) { **return** sf.func(s); }

* Inside ***main()***, define a block LE that *reverses the characters* and ***assign*** it to the ***StringFunc*** ***reference reverse***.

**StringFunc** reverse = (str) -> { **String** result = "";

**for**(**int** i = str.**length**()-1; i >= 0; i--) result += str.**charAt**(i);

**return** result; };

* Call ***changeStr()***, passing in the ***reverse*** lambda and ***inStr***. Assign the result to ***outStr***, and display the result.

***outStr = changeStr(reverse, inStr);***

* Because the first parameter to ***changeStr()*** is of type ***StringFunc***, the ***reverse*** *lambda* can be passed to it.
* Following lambdas *replace spaces with hyphens* and *invert the case of the letters*. Notice that both of these lambdas are embedded in the call to ***changeStr()***, itself, rather than using a separate ***StringFunc*** variable.

*// This lambda expression replaces spaces with hyphens. And LE directly passed as an argument to changeStr()*

outStr = changeStr( (str) -> str.**replace**(' ', '-'), inStr );

*// This block lambda inverts the case of the characters in the string. This bock is also passed as an argument directly in the call to changeStr().*

outStr = changeStr( (str) -> { **String** result = ""; **char** ch;

**for**(**int** i = 0; i < str.**length**(); i++ ) { ch = str.**charAt**(i);

**if**(**Character**.**isUpperCase**(ch)) result += **Character.toLowerCase**(ch);

**else** result += **Character.toUpperCase**(ch); }

**return** result;} , inStr );

* Embedding the *space-replacing-lambda* in the call to ***changeStr()*** is convenient: because it is a short, expression lambda that simply calls ***replace()*** to replace *spaces* with "**-**". [The ***replace()*** method is defined by the ***String*** class. ]
* For the sake of illustration, the *case-inverting-lambda* block is also embedded in the call to ***changeStr()***. Though, it is technically ok to pass *whole-lambda-block* as a parameter *(but a bad practice)*, it is better to *assign* such a lambda to a *separate* *reference* *variable* (as string-reversing lambda), and then *pass* that *variable* to the *method*.

Note

1. isUpperCase( ), toUpperCase( ), and toLowerCase( ) : The static methods ***isUpperCase()***, ***toUpperCase()***, and ***toLowerCase()*** defined by ***Character***. Recall that ***Character*** is a *wrapper* class for ***char***.

* The ***isUpperCase()*** method returns ***true*** if its argument is an uppercase letter and false otherwise.
* The ***toUpperCase()*** and ***toLowerCase()*** perform the indicated action and return the result.

1. In addition to variable initialization, assignment, and argument passing, some other places constitute a target type context for a lambda expression. They are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. Casts, | 1. the ? operator, | 1. array initializers, | 1. return statements, | 1. LEs, themselves |

**10.6 Lambda Expressions and Variable Capture**

A LE can obtain or set the value of an instance variable or static variable and call a method defined by its *enclosing class* (enclosing scope). A LE also has access to ***this*** (both explicitly and implicitly), which refers ***to the invoking instance*** of the LE's enclosing class.

* Effectively final variable: An *effectively final* variable is one whose value ***does not change after it is first assigned***. There is no need to *explicitly declare* such a variable as ***final***, although doing so would not be an error. (The ***this*** parameter of an ***enclosing*** ***scope*** is automatically effectively ***final***, and LEs do not have a ***this*** of their own.)
* Variable capture: when a LE uses a *local variable* from its enclosing scope (or class), a special situation is created that is referred to as a variable capture. In this case, a LE may only use *local variables* that are effectively ***final***. (LE cannot modify that variable.)
* It is important to understand that a *local variable* of the ***enclosing scope*** cannot be modified by the LE. Doing so would remove its *effectively final status*, thus rendering it illegal for capture.
* Example 8: The following program illustrates the difference between *effectively final* and *mutable local variables*: Capturing a local variable from the *enclosing scope*.

**interface** MyFunc { **int** func(**int** n); }

**class** VarCapture { **public static void main(String args[])** {

**int** num = 10; // A local variable that can be captured.

**MyFunc** myLambda = **(**n**)** **->** { **int** v = num + n; *// This use of num is OK. It does not modify num.*

*/\* modifying num* ***inside lambda*** *\*/* *//* num++; *// illegal because it attempts to modify the value of num.*

**return** v; };

**System.out.println**(myLambda.func(8)); *// Use the lambda. This will display 18.*

*//* num = 9; *// modifying num* ***outside lambda*** *also illegal; also cause an error, removing the effectively final status from num.*

}}

* Here, ***num*** is *effectively final* and can be used inside ***myLambda***. Hence, ***println()*** outputs ***18***. When ***func()*** is called with the argument ***8***, the value of ***v*** inside the lambda is set by adding ***num*** (which is ***10***) to the value passed to ***n*** (which is ***8***). Thus, ***func()*** returns ***18.***
* This works because ***num*** is not modified after it is initialized at the start. However, if ***num*** were to be modified, either inside the lambda or outside of lambda, ***num*** would lose its *effectively final status*. This would cause an error, and the program would not compile.

**10.7 Exception and LE**

A LE can throw an exception. If it throws a *checked exception*, however, then that ***exception must be compatible*** with the exception(s) listed in the ***throws*** clause of ***theabstract*** method in the FI. For example, if a LE throws an ***IOException***, then the *abstract method* in the FI must list ***IOException*** in a ***throws*** clause.

* Example 9: LE throwing an exception is demonstrated by the following program:

***import java.io.\*;***

**interface** MyIOAction { **boolean** ioAction(**Reader** rdr) **throws IOException**; } *//****IOException*** *is specified in* ***throws*** *of* ***ioAction()*** *in* ***MyIOAction****.*

**class** LambdaExceptionDemo { **public static void main(String args[])** { **double[ ]** values = { 1.0, 2.0, 3.0, 4.0 };

*// This block lambda could throw an IOException.* **MyIOAction** myIO = (rdr) -> { **int** ch = rdr.**read**(); *// now* ***read()*** *can throw* ***IOException***

**return** **true**; }; }}

* Because a call to ***read()*** could result in an ***IOException***, the ***ioAction()*** method of the FI ***MyIOAction*** must include ***IOException*** in a throws clause. Without it, the program will not compile because the LE will no longer be compatible with ***ioAction()***.

**10.8 Use an array parameter in LE**

Since, the type of a LE *parameter* will be inferred from the target context. Thus, if the *target context* requires an *array*, then the *parameter*’s type will ***automatically be inferred as an array***. However, when the ***type*** *of the parameter is* ***inferred***, the parameter to the LE is not specified using the normal array syntax. Rather, the parameter is specified as a simple name, such as ***n***, not as ***n[]***.

* Example 10: Here is a generic FI called ***MyTransform***, which can be used to apply some transform to the elements of an array:

**interface** MyTransform<T> { **void** transform(**T**[] a); } *// A functional interface.*

MyTransform<**Double**> sqrts = (v) -> { **for**(**int** i=0; i < v.**length**; i++) v[i] = Math.**sqrt**(v[i]); }; *// LE block that use array type defined in FI*

* Here, the type of ***a*** in ***transform()*** is ***Double[]***, because ***Double*** is specified as the type parameter for ***MyTransform LE*** when ***sqrts*** is declared. Therefore, the type of ***v*** in the LE is *inferred* as ***Double[]***. It is ***not necessary (or legal)*** to specify it as ***v[]***.
* It is legal to declare the lambda parameter as ***<Double[] v>***, because doing so *explicitly declares* the *type of the parameter*.

**10.9Method References (MRf) and Constructor References (CRf) & *Introducing separator*** '::'

A method reference (MRf) is a way to ***refer to a method without executing it***. It relates to LE because it, too, requires a *target type* *context* that consists of a compatible FI. When evaluated, a MRf also creates an instance of a FI. Methods participating MRf are called predicates.

* MRf to static Methods: A MRf to a static method is created by specifying the ***method name*** preceded by its ***class name***, as follows:

***ClassName :: methodName***

* Notice that the ***class name*** is separated from the ***method*** ***name*** by a double colon ' **::** ' seperator.
* This MRf can be used anywhere in which it is compatible with its *target type*.
* Example 11: The following program demonstrates the static MRf.

|  |  |
| --- | --- |
| **interface** IntPredicate { **boolean** test(**int** n); }  **class** MyIntPredicates {  *// returns true if a number is prime.*  **static** **boolean** isPrime(**int** n) {  **if**(n < 2) **return** false;  **for**(**int** i=2; i <= n/i; i++) { **if**((n % i) == 0) **return** **false**; }  **return** **true**; }  *// returns true if a number is even.*  **static** **boolean** isEven(**int** n) { **return** (n % 2) == 0; }  *// returns true if a number is positive.*  **static** **boolean** isPositive(**int** n) { **return** n > 0; }  } | **class** MethodRefDemo {  */\* can be passed a reference to any instance of* ***IntPredicate FI*** *to following* ***numTest*** *methods first parameter, including one created by a method reference.\*/*  **static** **boolean** numTest(**IntPredicate** p, **int** v) { **return** p.test(v); }  **public static void main(String args[])** { **boolean** result;  *// Here, a* ***MRf*** *to* ***isPrime*** *is passed to* ***numTest****().*  result = numTest(***MyIntPredicates :: isPrime***, 17);  **if**(result) **System.out.println**("17 is prime.");  *// Next, a* ***MRf*** *to* ***isEven*** *is used.*  result = numTest(***MyIntPredicates :: isEven***, 12);  **if**(result) **System.out.println**("12 is even.");  *// Now, a* ***MRf*** *to* ***isPositive*** *is passed.*  result = numTest(***MyIntPredicates :: isPositive***, 11);  **if**(result) **System.out.println**("11 is positive."); }} |

* A FI called ***IntPredicate*** is created first it has a method called ***test()*** with ***int*** parameter and it returns ***boolean***. Thus, it can be used to ***test an integer value against some condition***.
* Inside ***MyIntPredicates***, three ***static*** methods are defined, called ***isPrime()***, ***isEven()***, and ***isPositive()***.
* Inside ***MethodRefDemo***, a method called ***numTest()*** is created it takes a *reference* to ***IntPredicate*** as its *first parameter*,. Its *second parameter* specifies the integer being tested.
* Inside ***main()***, three different tests are performed by calling ***numTest()***, passing in a MRf to the ***test*** to perform.
* First, a *reference* to the ***static*** method ***isPrime()*** is passed as the first argument to ***numTest()***. This works because ***isPrime*** is compatible with the ***IntPredicate*** FI. Thus, the "expression" ***MyIntPredicates :: isPrime*** evaluates *to a reference to an object* in which ***isPrime()*** ***provides the implementation*** of ***test()*** in ***IntPredicate***. The other two calls to ***numTest()*** work in the same way.
* Also notice, all 3 ***static*** methods ***boolean* isPrime*(int n)***, ***boolean* isEven*(int n)***, ***boolean* isPositive*(int n)*** are in same form as FI method ***boolean*** *test****(int n)***;
* MRf to Instance Methods: The syntax to create ***a reference to an instance method on a specific object***, ***objRef::methodName***
* i.e. the syntax is similar to that used for a *static method*, except that an object reference is used instead of a class name.
* Thus, the method referred to by the MRf operates relative to ***objRef***.
* Example 12: Following illustrates this point. It uses the same ***IntPredicate*** interface and ***test()*** method as Example 11.

|  |  |
| --- | --- |
| *// A* ***FI*** *for numeric predicates that operate on integer values.*  **interface** IntPredicate { **boolean** test(**int** n); }  */\* MyIntNum class stores an int value and defines the instance method isFactor(), which returns true if its argument is a factor of the stored value. \*/*  **class** MyIntNum { **private** **int** v;  MyIntNum(**int** x) { v = x; }  **int** getNum() { **return** v; }  *// instance method: Return true if n is a factor of v.*  **boolean** isFactor(**int** n) {  **return** (v % n) == 0; } } | **class** MethodRefDemo2 { **public static void main(String args[])** {  **boolean** result;  */\* A* ***MRf to an instance method*** *\*/* **MyIntNum** myNum = **new** MyIntNum(12);  **MyIntNum** myNum2 = **new** MyIntNum(16);  **IntPredicate** *ip* = myNum :: isFactor; *// a* ***MRf*** *to* ***isFactor*** *on* ***myNum*** *is created.*  result = ip.test(3); *//* ***MRf ip*** *is used to call* ***isFactor()*** *via* ***test()****.*  **if**(result) **System.out.println**("*3 is a factor of* " + myNum.getNum());    *ip* = myNum2 :: isFactor; *// a* ***MRf*** *to* ***isFactor*** *on* ***myNum2*** *is created.*  result = ip.test(3); *//* ***MRf ip*** *is used to call* ***isFactor()*** *via* ***test()****.*  **if**(!result) **System.out.println**("*3 is not a factor of* " + myNum2.getNum()); }} |

* MyIntNumstores an ***int*** and defines ***isFactor()*** - tests passed value is a *factor* of stored value by MyIntNuminstance.
* Inside ***main()*** two ***MyIntNum*** instances are created. Then ***numTest()*** called, passing in a MRf to the ***isFactor()*** method and the value to be checked. In each case, the MRf operates relative to the specific object.
* Pay special attention to the line: ***IntPredicate ip = myNum::isFactor;***
* The MRf assigned to ip refers to an *instance method* ***isFactor()*** on ***myNum***. Thus, when ***test()*** is called through that *reference*, as: ***result = ip.test(3);*** the method will call ***isFactor()*** on ***myNum***, which is the *object specified* when the MRf was created.
* The same situation occurs with the MRf ***myNum2::isFactor***, except that ***isFactor()*** will be called on ***myNum2***.
* Use an instance method reference to refer to any instance: To Specify an *instance method* that can be used with *any object* of a given class—not just a specified object: Create a MRf as: **ClassName::instanceMethodName**
* Here, the *name of the class* is used instead of a *specific object*, even though an *instance method* is specified.
* With this form, the ***first parameter*** of the FI matches the *invoking object* and the ***second parameter*** matches the *parameter (if any) specified by the method*.
* Example 13: It reworks the previous Example 12. First, it replaces ***IntPredicate*** FI with the ***MyIntNumPredicate*** FI.

|  |  |
| --- | --- |
| */\* A FI for numeric predicates that operate on an object of type MyIntNum and an integer value. \*/*  **interface** MyIntNumPredicate {  **boolean** test(**MyIntNum** mv, **int** n); }  */\* MyIntNum class stores an int value and defines the instance method isFactor(), which returns true if its argument is a factor of the stored value. \*/*  **class** MyIntNum { **private** **int** v;  MyIntNum(**int** x) { v = x; }  **int** getNum() { **return** v; }  *// Return true if n is a factor of v.*  **boolean** isFactor(**int** n){ **return** (v % n) == 0; } } | **class** MethodRefDemo3 { **public static void main(String args[])** {  **boolean** result;  **MyIntNum** myNum = **new** MyIntNum(12);  **MyIntNum** myNum2 = **new** MyIntNum(16);  *// This makes inp refer to the instance method isFactor().*  ***/\* MRf to any object of type MyIntNum \*/***  **MyIntNumPredicate** inp = MyIntNum::isFactor;    result = inp.test(myNum, 3); *// calls* ***isFactor****() on* ***myNum****.*  **if**(result) **System.out.println**("*3 is a factor of* " + myNum.getNum());  result = inp.test(myNum2, 3); *// calls* ***isFactor****() on* ***myNum2****.*  **if**(!result) **System.out.println**("*3 is a not a factor of* " + myNum2.getNum());  }} |

* The first parameter to ***test()*** is of type ***MyIntNum***. It will be used to *receive the object being operated upon*. This allows the program to create a MRf to the *instance method* ***isFactor()*** that can be used with any ***MyIntNum*** *object*.
* Pay special attention to this line: ***MyIntNumPredicate*** *inp = MyIntNum::isFactor;*
* It creates a MRf to the instance method ***isFactor()*** that will work with any object of type ***MyIntNum***. For example, when ***test()*** is called through the ***inp***, as shown here: ***result = inp.test(myNum, 3);*** it results in a call to ***myNum.isFactor(3)***. [In other words, ***myNum*** becomes the object on which ***isFactor(3)*** is called.]
* **Specify a** MRf **to a** generic method: Because of *type inference* most often we don’t need to *explicitly specify* a type argument to a generic method when obtaining its MRf. However Java does include a syntax to handle those cases.

|  |  |
| --- | --- |
| Generally: | * When a genericmethod is specified as a MRf, its *type argument* comes after ***::*** and before the ***method name***. * In cases in which a genericclass is specified, the *type argument* follows the ***class name*** and precedes the ***::***. |

|  |  |
| --- | --- |
| **interface** SomeTest<T> { **boolean** test(T n, T m); } | **class** MyClass { **static <T> boolean** myGenMeth**(T x, T y)** { **boolean** result = **false**;  **return** result; }} |

Then this statement is valid: **SomeTest<Integer>** mRef = **MyClass::<Integer>myGenMeth**;

* Here, *type argument* for the *generic method* ***myGenMeth*** is explicitly specified. Notice that the type argument occurs after **::** .
* Constructor References (CRf): CRf are created in similar way of MRf. The general form of the syntax is: ***classname::new***
* This reference can be assigned to any FI *reference* that defines a ***method*** compatible with the ***constructor***.
* Example 14: Here is a simple example to Demonstrate a Constructor reference.

|  |  |
| --- | --- |
| *// MyFunc is a FI whose method returns a MyClass reference.*  **interface** MyFunc { **MyClass** func(**String** s); }  **class** MyClass { **private** **String** str;  MyClass(**String** s) {str = s;} *// parameterized constructor*  MyClass() { str = ""; } *// default constructor.*  **String** getStr() { **return** str; } } | **class** ConstructorRefDemo **public static void main(String args[])** {  */\*CRf\*/* **MyFunc** myClassCons = **MyClass**::**new**; *//new refers to parameterized constructor*  **MyClass** mc = myClassCons.func("Testing"); *// instance of MyClass via that CRf.*  *// Use the instance of MyClass just created.*  **System.out.println**("str in mc is " + mc.getStr( )); }} |

* Here, notice that the ***func()*** method of ***MyFunc*** returns a reference of type ***MyClass*** and has a ***String*** parameter.
* Notice, ***MyClass*** defines two constructors. First is parameterized of type ***String***. Second is the default, parameterless.
* Now, examine the line: **MyFunc** myClassCons = **MyClass::new**; Here, the expression ***MyClass::new*** creates a *reference* to the ***MyClass*** *constructor*. In this case, because ***func()*** in ***MyFunc*** FI takes a ***String*** *parameter*, ***new*** refers to the *matching* parameterized constructor ***MyClass(String s)***, not the *default* constructor.
* The CRf is assigned to a ***MyFunc*** reference called ***myClassCons***. Now ***myClassCons*** has become another way to call ***MyClass(String s)***, as this line shows: ***MyClass*** *mc = myClassCons.func("Testing");*
* If you want ***MyClass::new*** to use ***MyClass***’s *default constructor*, then use a FI that defines a *parameterless method*. For example, if you define ***MyFunc2***, as shown here: **interface** MyFunc2 { **MyClass** func(); }
* A CRf then the following line will assign to MyClassCons a reference to MyClass’s *default (i.e., parameterless) constructor*:

**MyFunc2** myClassCons = **MyClass::new**;

* In general, the ***constructor*** *that will be used* when ***::new*** is specified is the one whose ***parameters*** *match* those specified by the FI.
* CRf *for a* generic class*:* In the case of creating a CRf for a *generic class*, you can specify the *type parameter* in the ***normal way, after the class name***. For example, if ***MyGenClass*** is declared like this: ***MyGenClass<T> { // ...***

Then **MyGenClass<Integer>::new;** creates a CRf with a *type argument* of ***Integer***:

* Because of *type inference*, you ***won’t always need*** to specify the type argument, but you can ***when necessary***.
* CRf *for an* array: To create a CRf for an array use this form: **type[]::new** Here, type specifies the *type of object* being created.
* Example 15: Assuming the form of ***MyClass*** shown in the preceding example and given the ***MyClassArrayCreator*** FI shown:

**interface** MyClassArrayCreator { **MyClass[]** func(**int** n); }

Following creates an array of MyClass objects and gives each element an initial value:

**MyClassArrayCreator** mcArrayCons = **MyClass[]::new**;

**MyClass[]** a = mcArrayCons.func(3);

**for**(**int** i=0; i < 3; i++) a[i] = **new** MyClass(i);

* Here, the call to ***func(3)*** causes a *three-element array* to be created.
* Generally, Any FI that will be used *to create an array* must *contain a method that takes a single* int *parameter* and returns a ***reference*** to the array of the specified size.
* To create a generic FI that can be used with *other types of classes*, as: ***interface*** *MyArrayCreator<T> {* ***T[]*** *func(****int*** *n); }*
* For example, you could create an array of five Thread objects like this:

**MyArrayCreator<Thread>** mcArrayCons = **Thread[]::new**;

**Thread[]** thrds = mcArrayCons.func(5);

**10.10 Predefined FIs (PREDICATE interfaces) and LEs with API Library**

In many cases you won’t need to define your own FIs because JDK 8 adds a new package called java.util.function that provides several predefined FIs. Some given in the following table:

|  |  |  |
| --- | --- | --- |
| Interface | | Purpose |
| **UnaryOperator<T>** | | Apply a unary operation to an object of type T and return the result, which is also of type T. Its method is called apply( ). |
| **BinaryOperator<T>** | | Apply an operation to two objects of type T and return the result, which is also of type T. Its method is called apply( ). |
| **Consumer<T>** | | Apply an operation on an object of type T. Its method is called accept( ). |
| **Supplier<T>** | | Return an object of type T. Its method is called get( ). |
| **Function<T, R>** | | Apply an operation to an object of type T and return the result as an object of type R. Its method is called apply( ). |
| **Predicate<T>** | Determine if an object of type T fulfills some constraint. Returns a boolean value that indicates the outcome. Its method is called test( ). | |

* Example 14: The following program shows the *Predicate interface* in action. It uses *Predicate* as the FI for a LE that determines if a number is even. ***Predicate’s abstract method*** is called ***test()***, and it is: ***boolean test(T val)*** It must return ***true*** if ***val*** *satisfies some constraint or condition*. As it is used following, it will return ***true*** if ***val*** is even.

***import java.util.function.Predicate;*** *// Import the Predicate interface.*

**class** UsePredicateInterface { **public static void main(String args[])** {

**Predicate<Integer>** isEven = **(**n**)** **->** (n %2) == 0; *// This lambda uses the built-in Predicate interface Predicate<Integer> to determine if n is even.*

**if**(isEven.test(4)) **System.out.println**("4 is even"); **if**(!isEven.test(5)) **System.out.println**("5 is odd"); }}

* Lambda expressions resulted in new capabilities being incorporated into the API library: the new stream package java.util.stream is added to the *Java API library*. This package defines several stream classes, the most general of which is Stream.
* A stream can represents a *sequence of objects*. A stream supports many types of operations that let you create a *pipeline* that performs a *series of actions on the data*. Often, these actions are represented by LEs.
* *Eg:* using the stream API, you can construct sequences of actions that resemble, in concept, the type of database queries for which you might use SQL. Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved.

|  |  |
| --- | --- |
| Note: | Although the streams supported by the new stream API have *some similarities with the* ***I/O streams*** described in Chapter 6, they are not the same. |

**10.11 MODULE Intro**

A Module is set of packages designed for re-use. Modules let you control which ***parts of a module*** are accessible to ***other modules*** and which are not. Through the use of ***modules*** you can create more ***reliable***, ***scalable*** programs.

* As a general rule, ***modules are most helpful*** to large applications because they help reduce the management complexity often associated with a large software system. [Small programs also benefit from modules because the Java API library has now been organized into modules. Thus, it is now possible to specify which parts of the API are required by your program and which are not. This makes it possible to deploy programs with a smaller runtime footprint, which is especially important when creating code for small devices, such as those intended to be part of the Internet of Things (IoT).]
* Module: A ***module is a grouping of packages*** and ***resources*** that can be *collectively referred* to by the module’s name. A ***module*** ***declaration*** specifies the ***module's name*** and defines the relationship a ***module and its packages*** have to other ***modules***.
* Actually module determines the ***structure of a program***. It helps to easily control and manage what part of the code ***can be*** ***reused*** and the parts ***can't be reused***. It is the largest building block of codes in Java.
* Module declarations are *program statements* in a ***Java source*** ***file*** and are supported by several ***module-related keywords*** added to Java by JDK 9. They are:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***exports*** | ***module*** | ***open*** | ***opens*** | ***provides*** | ***requires*** | ***to*** | ***transitive*** | ***uses*** | ***with*** |

* These keywords are recognized as keywords only in the context of a module declaration. Otherwise, they are interpreted as identifiers in other situations [not now recommended]. So there is *no chance of conflict* with pre-JDK 9 code. Because they are context-sensitive, the ***module-related*** keywords are formally called restricted keywords.

**10.12 MODULE: Declaration and Use**

* ***moduleinfo.java*** and module descriptor: A ***module declaration*** is contained in a file called ***moduleinfo.java***, i.e. a ***module*** is defined in a ***Java source file***. This file is then compiled by ***javac*** into a class file and is known as a ***module descriptor***.
* The moduleinfo.java file must contain *only a module definition*. It is not a general purpose file.
* A module declaration begins with the keyword ***module***. General form: ***module moduleName {*** */\* module definition \*/* ***}***
* The name of the module is specified by ***moduleName***, which must be a valid ***Java identifier*** or a ***sequence of identifiers*** separated by ***periods***.
* The *module definition* is specified within the braces. Although a module definition may be empty (which results in a declaration that simply names the module), typically it specifies *one or more clauses* that define the *characteristics* of the *module*.
* Accessibility Mechanism comparison: Let's compare the ***Accessibility*** between parts of source file, before and after JDK 9.

|  |  |  |  |
| --- | --- | --- | --- |
| Before JDK 9 | | After Module introduced | |
| * Public * Protected | * Package * Private | * Public to all * Public to friend module * Public to only current module | * Protected * Package * Private |

* Before JDK 9, reuse of codes done by, classes – using ***inheritance***, Interfaces­­­­­­­­­­­­­ – by ***abstraction***, Packages – by ***arranging classes***.
* **requires** and **exports**: A module is able to specify that it requires another module. A *dependence relationship* is specified by use of a ***requires*** statement. By default, the ***presence of the required module*** is checked at both ***compile time*** and ***run time***.
* A module is able to control which, if any, of its packages are accessible by another module. To do this, use the ***exports*** keyword. ***Public,*** ***protected*** types in a package are accessible to other modules only if they are explicitly ***exported***.
* Giving module names: The ***name of a*** module (such as ***appfuncs***) is the prefix ***of the name of a*** package (such as ***appfuncs.simplefuncs***) that it contains. This is not required, but when creating modules suitable for ***distribution***, you must be careful with the names you choose because you will want those names to be unique.
* Reverse domain naming: In this method, the ***reverse domain name of the*** domain that “owns” the project is used as a prefix for the module. Eg: a project associated with ***xyz.com*** would use ***com.xyz*** as the ***module prefix***. (Same for *package names*.)
* Check the Java documentation for current recommendations it may change over time to time.

|  |  |
| --- | --- |
| NOTE : | We use the commandline tools to create, compile, and run ***modulebased*** code. This approach shows the fundamentals of the module system, including how it utilizes directories. You will need to manually create a number of directories and ensure that each file is placed in its proper directory. |

* Example 15: Following creates a *modular application* that demonstrates some simple mathematical functions. It illustrates the core concepts and procedures required to create, compile, and run module-based code. The application defines two modules:

1. The first module is called ***appstart***. It contains a package called ***appstart.mymodappdemo*** that defines the application’s entry point in a class called ***MyModAppDemo*** [ i.e. this class contains ***main()*** ].
2. The second module is called ***appfuncs***. It contains a package called ***appfuncs.simplefuncs*** that includes the class ***SimpleMathFuncs***. This class defines three *static methods* that implement some simple *mathematical functions*.

* Create directory tree: Create a directory called ***mymodapp***. This is the *top-level directory* for the entire *application*.
* Under ***mymodapp***, create a ***subdirectory*** called ***appsrc***. This is the ***top-level*** directory for the application’s ***source code***.
* Under ***appsrc***, create the ***subdirectory*** ***appstart***. Under this directory, create a ***subdirectory*** also called ***appstart***. Under this directory, create the directory ***mymodappdemo***. Thus, beginning with ***appsrc***, you will have created this tree:

***mymodapp\appsrc\appstart\appstart\mymodappdemo***

* Also under ***appsrc***, create the ***subdirectory*** ***appfuncs***. Under this directory, create a ***subdirectory*** also called ***appfuncs***. Under this directory, create the directory called ***simplefuncs***. Thus, beginning with ***appsrc***, you will have created this tree:

***mymodapp\appsrc\appfuncs\appfuncs\simplefuncs***

* After setting up these directories, we can create the application’s source files. This example will use four source files. Two are the source files that define the application: ***SimpleMathFuncs.java*** and ***MyModAppDemo.java***. Then we use two ***moduleinfo.java*** source files one for ***appstart*** directory and another for ***appfuncs*** directory.
* SimpleMathFuncs.java: Notice that ***SimpleMathFuncs*** is packaged in ***appfuncs.simplefuncs***. ***SimpleMathFuncs*** defines three simple ***static math functions***. [The first, isFactor(), returns true if a is a factor of b. The lcf( ) method returns the least common factor of a and b. The gcf( ) method returns the greatest common factor of a and b. In both cases, 1 is returned if no common factors are found. ]
* ***This file must be put in:*** **appsrc\appfuncs\appfuncs\simplefuncs** It is appfuncs.simplefuncs ***package directory.***

**package** appfuncs.simplefuncs; *// notice package declaration*

**public class** SimpleMathFuncs {

**public static** **boolean** isFactor(**int** a, **int** b) { **if**((b%a) == 0) **return** **true**; **return** **false**;}

**public static int** lcf(**int** a, **int** b) { a = **Math.abs**(a); b = **Math.abs**(b); **int** min = a < b ? a : b;

**for**(**int** i = 2; i <=min/2; i++) { **if**(isFactor(i, a) && isFactor(i, b)) **return** i; }

**return** 1; }

**public static int** gcf(**int** a, **int** b) { a = **Math.abs**(a) ; b = **Math.abs**(b); **int** min = a < b ? a : b;

**for**(**int** i = min/2; i >= 2; i--) { **if** (isFactor(i, a) && isFactor(i, b)) **return** i; }

return 1; } }

* MyModAppDemo.java: Uses the methods in ***SimpleMathFuncs***. Notice that it is packaged in ***appstart.mymodappdemo***. Also note that it imports the ***SimpleMathFuncs*** *class* because it depends on ***SimpleMathFuncs*** for its operation.
* ***This file must be put in:*** **appsrc\appstart\appstart\mymodappdemo** It is appstart.mymodappdemo ***package directory.***

**package** appstart.mymodappdemo; *//Notice the package declaration*

**import** appfuncs.simplefuncs.SimpleMathFuncs; *//import statement.*

**public** **class** MyModAppDemo { **public static void main(String[] args)** {

**if**(SimpleMathFuncs.isFactor(2, 10)) **System.out.println**("2 is a factor of 10");

**System.out.println**("Smallest factor common to both 35 and 105 is " + SimpleMathFuncs.lcf(35, 105));

**System.out.println**("Largest factor common to both 35 and 105 is " + SimpleMathFuncs.gcf(35, 105)); }}

* Now we need to add ***module-info.java*** files *for each module*. These files contain the ***module definitions***. First, add following, which defines the ***appfuncs*** ***module***:

*// Module definition for the functions module : Define a module for appfuncs.*

**module** appfuncs { **exports** appfuncs.simplefuncs; */\* Exports the package appfuncs.simplefuncs. \*/* }

* Notice that appfuncs exports the package ***appfuncs.simplefuncs***, which makes it *accessible* to other *modules*.
* This file must be put into this directory: ***appsrc\appfuncs*** i.e, it goes in the *appfuncs module* directory, which is above the package directories.
* To add the ***moduleinfo.java*** file for the appstart module:

*// Module definition for the main application module : Define a module for appstart.*

**module** appstart { **requires** appfuncs; */\* Requires the module appfuncs. \*/* }

* Notice that appstart requires the module appfuncs. This file must be put into its module directory: ***appsrc\appstart***
* Compile and Run:

*Didn't work ----* First create the following directory (folder): in ***mymodapp***

*Didn't work ----* **mymodapp\appmodules\appfuncs**

*Didn't work ----* javac must run inside ***mymodapp: C:\Users\User\mymodapp> javac . . . .***

*Didn't work ----* Power shell command : PS C:\Users\User\mymodapp> javac -d appmodules\appfuncs appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java

* Like all other Java programs, module-based programs are compiled using javac. In this case we need to ***explicitly specify*** a module path. A module path tells the compiler where the compiled files will be located.
* Execute the ***javac*** commands from the ***mymodapp*** directory in order for the paths to be correct.
* Compile the ***SimpleMathFuncs.java*** file, using followng command:

**javac -d appmodules\appfuncs appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java**

* This command must be executed from the mymodapp directory. Notice the use of the –d option. This tells javac where to put the output .class file. For the examples in this chapter, the top of the directory tree for compiled code is appmodules.
* This command will automatically create the output package directories for appfuncs.simplefuncs under appmodules\appfuncs as needed.
* Next, here is the ***javac*** command that compiles the ***moduleinfo.java*** file for the ***appfuncs*** module:

***javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java***

* This puts the moduleinfo.class file into the appmodules\appfuncs directory.
* However, we can combine above two command lines into a single one. It is usually easier to compile a module’s moduleinfo.java file and its source files in one command line.

***javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java***

* Now, compile the ***moduleinfo.java*** and ***MyModAppDemo.java*** files for the ***appstart*** module, using following command:

**javac --module-path appmodules -d appmodules\appstart appsrc\appstart\module-info.java**

**appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java**

* Notice the ***–module-path*** option. It specifies the module path, which is the path on which the compiler will look for the user-defined modules required by the ***moduleinfo.java*** file. In this case, it will look for the ***appfuncs*** module because it is needed by the ***appstart*** module.
* It also specifies the output directory as ***appmodules\appstart***. This means that the ***moduleinfo.class*** file will be in the ***appmodules\appstart*** module directory and ***MyModAppDemo.class*** will be in the ***appmodules\appstart\appstart\mymodappdemo*** package directory.
* Once you have completed the compilation, you can run the application with this java command:

***java --module-path appmodules -m appstart/appstart.mymodappdemo.MyModAppDemo***

* Here, the --module-path option specifies the path to the application’s modules. [***appmodules*** is the directory at the top of the compiled modules tree].
* The –m option specifies the class that contains the entry point of the application and, in this case, the name of the class that contains the ***main()*** method.

**10.13 requires and exports : Details**

The two foundational features of the module system: the ability to ***specify a dependence*** and the ability to ***satisfy that dependence***. These capabilities are specified through the use of the ***requires*** and ***exports*** statements within a module declaration.

* *General form of the requires statement:* ***requires moduleName;*** Here, moduleName specifies the *name of a module* that is required by the module in which the requires statement occurs. This means that the required module must be present in order for the current module to compile.
* In the language of modules, the current module is said to read the module specified in the requires statement.
* *General form of the exports statement:* ***exports packageName;*** Here, packageName specifies the ***name of the package*** that is exported by the module in which this ***statement occurs***. When a module exports a package, it makes all of the public and protected types (including members) in the package accessible to ***other modules***.
* If a package within a module is not exported, then it is private to that module, including all of its public types.
* public & protected types of a package, whether exported or not, are always accessible within that package’s module. exports makes them accessible to outside modules.
* requires and exports work together. If one module depends on another, then it must specify that dependence with requires. The module on which another depends must explicitly export (i.e., make accessible) the packages that the dependent module needs. If either side is missing, the dependent module will not compile.
* requires and exports must occur only within a module statement. Furthermore, a module statement must occur by itself in a file called moduleinfo.java.

**10.14 *java.base* And PLATFORM modules**

Java API packages have been incorporated into modules. The API modules are referred to as platform modules, and their names all begin with the prefix ***java***. Here are some examples: ***java.base***, ***java.desktop***, and ***java.xml***. By modularizing API, it becomes possible to deploy an application with only the packages that it requires, rather than the entire Java Runtime Environment (JRE).

* java.base: Of the platform modules, the most important is ***java.base***. It ***includes*** and ***exports*** those packages *fundamental* to Java, such as ***java.lang***, ***java.io***, and ***java.util***, among many others.
* ***java.base*** is automatically accessible and automatically required by to all modules. There is no need ***requires java.base*** statement in a module declaration.
* java.lang is also automatically available to all programs without import statement, since java.base module is automatically accessible to all module-based programs without explicitly requesting it.
* That’s why ***System.out.println()*** worked without specifying a requires statement for the module that contains the System class.
* Since java.base contains the java.lang package, and java.lang contains the System class, ***MyModAppDemo*** in the preceding example can automatically use ***System.out.println()*** without a requires statement. Same applies to Math class in SimpleMathFuncs, because the Math class is also in java.lang.
* Many of the API classes you will commonly need are in the packages included in java.base. Thus, the automatic inclusion of java.base simplifies the creation of module-based code because Java’s core packages are automatically accessible.
* Beginning with JDK 9, the documentation for the Java API now tells you the name of the module in which a package is contained. If the module is java.base, then you can use the contents of that package directly. Otherwise, your module declaration must include a requires clause for the desired module.
* Compact profiles: *Compact profiles* are a feature that, in some situations, let you specify a subset of the API library. They are not part of the module system.

**10.15 LEGACY code and the UNNAMED module**

Java ensures backward compatibility with preexisting code. Support for legacy code is provided by two key features.

* The first is the unnamed module. When you use code that is not part of a named module, it automatically becomes part of the unnamed module. The unnamed module has two important attributes.
* First, all of the packages in the unnamed module are automatically exported.
* Second, the unnamed module can access any and all other modules. Thus, when a program does not use modules, all API modules in the Java platform are automatically accessible through the unnamed module.
* The second key feature that supports legacy code is the automatic use of the class path, rather than the module path. When you compile a program that does not use modules, the class path ***mechanism is employed***, just as it has been since Java’s original release.
* Modularizing simple programs would simply add clutter and complicate them for no reason or benefit. Modules are often of the greatest benefit when creating commercial programs.

**10.16** The ***to*** *clause*and ***requires*** *transitive, multi-module compilation*

* to clause: In some specialized development situations, it can be desirable to make a package accessible to only a specific set of modules, not all other modules. In an exports *statement*, the to clause specifies a list of one or more modules that have *access to the exported package*. Furthermore, only those modules named in the to clause will have access. In the language of modules, the to clause creates what is known as a qualified export. The form of exports that includes to is shown here:

***exports packageName to moduleNames;***

* Here, moduleNames is a *comma-separated list* of modules to which the exporting module grants access.
* You can try the to clause by changing the moduleinfo.java file for the appfuncs module, as shown here:

***module*** appfuncs { ***exports*** appfuncs . simplefuncs ***to*** appstart; */\* A qualified export: Exports the package appfuncs.simplefuncs to appstart. \*/* }

Now, simplefuncs is exported only to appstart and to no other modules. After making this change, you can recompile the application by using this javac command:

***javac -d appmodules --module-source-path appsrc appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java***

* multi-module compilation mode: Notice the command line, it specifies the *module-source-path* option. The *module source path* specifies the top of the module source tree. The module-source-path option automatically compiles the files in the tree under the specified directory, which is appsrc in this example. The module-source-path option must be used with the –d option to ensure that the compiled modules are stored in their proper directories under appmodules. This form of javac is called multi-module mode because it enables more than one module to be compiled at a time.
* The multi-module compilation mode is especially helpful here because the to clause refers to a ***specific module***, and the ***requiring module*** must have access to the ***exported package***. Thus, in this case, both appstart and appfuncs are needed to avoid warnings and/or errors during compilation. Multi-module mode avoids this problem because both modules are being compiled at the same time.
* The multi-module mode also automatically finds and compiles all source files for the application, ***creating the necessary output directories***.
* ***requires*** transitive: Consider a situation in which there are three modules, A, B, and C, that have the dependences: A requires B. B requires C. Since A depends on B and B depends on C, A has an indirect dependence on C. As long as A does not directly use any of the contents of C, then you can simply have A require B in its module-info file, and have B export the packages required by A in its module-info file, as shown here:

*/\* A's module-info file \*/***module** A { **requires** B; }

*/\* B's module-info file \*/***module** B { **exports** somepack; **requires** C; }

* Here, ***somepack*** is a *placeholder* for the package exported by B and used by A.
* Although this works as long as A does not need to use anything defined in C, a problem occurs if A does want to access a type in C. There are two solutions:
* The first solution is to simply add a requires C statement to A’s file, as: **module** A { **requires** B; **requires** C; */\* also require C \*/* }
* This solution works, but if B will be used by many modules, you must add requires C to ***all module definitions*** that require B. This tedious & error prone.
* A better solution using requires transitive: You can create an implied dependence on C. ***Implied dependence*** is also referred to as implied readability. To create an implied dependence, add the ***transitive*** keyword after ***requires*** in the clause that requires the module upon which an ***implied readability*** is needed. In the case of this example, you would change B’s module-info file as: **module** B{ **exports** somepack; **requires** **transitive** C;}
* Here, C is now required as transitive. Now, any module that depends on B will also automatically depend on C. Thus, A would automatically have access to C.
* In a requires statement, if transitive is immediately followed by a separator (such as a semicolon), it is interpreted as an identifier rather than a keyword.

**10.17 SERVICES**

In programming, it is often useful to separate *what must be done* from *how it is done*. One way this is accomplished in Java is through the use of *interfaces*. The interface specifies the what, and the *implementing class* specifies the how.

* This concept can be expanded so that the implementing class is provided by code that is outside your program, through the use of a plugin. Using such an approach, the capabilities of an application can be enhanced, upgraded, or altered by simply changing the plugin. The core of the application itself remains unchanged. Java supports a pluggable application architecture through the use of services and service providers. In large, commercial applications, Java’s module system provides support for them.
* Service and Service Provider: In Java, a service is a program unit whose functionality is defined by an interface or an abstract class. Thus, a service specifies in a general way some form of program activity. A ***concrete implementation of a service*** is supplied by a service provider. In other words, a service defines the form of some action, and the service provider supplies that action.
* As mentioned, services are often used to support a pluggable architecture. For example, a service might be used to support the translation of one language into another. In this case, the service supports translation in general. The service provider supplies a specific translation, such as German to English or French to Chinese. Because all service providers implement the same interface, different translators can be used to translate different languages without having to change the core of the application. You can simply change the service provider.
* *Service providers* are supported by the ***ServiceLoader*** class. ***ServiceLoader*** is a *generic class* packaged in ***java.util***. It is declared like: ***class ServiceLoader<S>*** Here, ***S*** specifies the ***service type***.
* Service providers are loaded by the ***load()*** method. It has several forms; the one we will use is:

***public static <S> ServiceLoader<S> load(Class <S> serviceType)***

* Here, serviceType specifies the Class *object* for the desired service type. Recall from Java/C# Chapter 9 : Generics that ***Class*** is a class that *encapsulates information* about a class. There are a variety of ways to obtain a Class instance. The way we will use here is called a class literal. A class literal has general form: ***className.class*** Here, ***className*** specifies the *name of the class*.
* When ***load()*** is called, it returns a ***ServiceLoader*** *instance* for the application. This object supports iteration and can be cycled through by use of a for-each for loop. Therefore, to find a specific provider, simply search for it using a loop.
* Service-Based Keywords: Modules support services through the use of the keywords provides, uses, and with. When used together, they enable you to specify a module that ***provides a service***, a module that ***needs that service***, and the ***specific implementation of that service***. Furthermore, the module system ensures that the service and service providers are ***available and will be found***.
* A module specifies that it provides a service with a provides statement. To provide a service, the module indicates both the name of the service and its implementation. General form of provides: ***provides serviceType with implementationTypes;***
* Here, serviceType specifies the ***type of the service***, which is often an interface, although abstract classes are also used.
* A ***comma-separated list*** of the implementation types is specified by implementationTypes.
* A module indicates that it requires a service with a uses statement. Form: ***uses serviceType;*** serviceType specifies the type of the service required.
* The ***specific type of service provider*** is declared by with.

**10.18 Example of A Module-Based Service (*Example 16*)**

|  |  |
| --- | --- |
| * Consider Example 15, to it we will add two new modules: | * The first is called ***userfuncs***. It will define interfaces that support functions that perform binary operations in which each argument is an int and the result is an int. * The second module is called ***userfuncsimp***, and it contains concrete implementations of the interfaces. |

* This example expands the original version of the application by providing support for functions beyond those built into the application. Recall that the ***SimpleMathFuncs*** class supplies three builtin functions: ***isFactor()***, ***lcf()***, and ***gcf()***. Although it would be possible to add more functions to this class, doing so requires modifying and recompiling the application. By implementing services, it becomes possible to “plug in” ***new functions at run time without modifying the application***, and that is what this example will do.
* Create two directory tree: ***appsrc\userfuncs\userfuncs\binaryfuncs*** and ***appsrc\userfuncsimp\userfuncsimp\binaryfuncsimp***
* Here, the service supplies functions that take two int arguments and return an int result. Of course, other types of functions can be supported if additional interfaces are provided, but support for binary integer functions is sufficient for our purposes and keeps the source code size of the example manageable.
* The Service Interfaces: Two *service-related interfaces* are needed. One specifies the *form of an action*, and the other specifies the *form of the provider of that action*. Both go in the ***binaryfuncs*** directory, and are in the ***userfuncs.binaryfuncs*** package.
* BinaryFunc: Following is the first interface, called ***BinaryFunc***, declares the ***form of a binary function***. It defines a function that takes two ***int*** arguments and returns an ***int*** result. Thus, it can ***describe any binary operation*** on two ints that returns an int

**package** userfuncs.binaryfuncs;

**public interface** BinaryFunc { **public** **String** getNameO; *// Obtain the name of the function*

**public** **int** func(**int** a, **int** b); */\* This is the function to perform. It will be provided by specific implementations \*/*  }

***BinaryFunc*** declares the form of an object that can implement a binary integer function. This is specified by ***func()***. The name of the function is obtainable from ***getName()***. Name will be used to determine what type of function is implemented. This interface is implemented by a class that supplies a binary function.

* BinFuncProvider: Following is the second interface, which declares the form of the service provider. It is called ***BinFuncProvider***. This interface obtains BinaryFunc instances

**package** userfuncs.binaryfuncs;

**import** userfuncs.binaryfuncs.BinaryFunc;

**public interface** BinFuncProvider { **public** BinaryFunc get(); *// Obtain a BinaryFunc* }

BinFuncProvider declares only one method, ***get()***, which is used to obtain an instance of BinaryFunc. This interface must be implemented by a class that wants to provide instances of BinaryFunc.

* The Implementation Classes: In this example, two ***concrete implementations*** of BinaryFunc are supported. The first is AbsPlus, which returns the sum of the absolute values of its arguments. The second is AbsMinus, which returns the result of subtracting the absolute value of the second argument from the absolute value of the first argument. These are provided by the classes AbsPlusProvider and AbsMinusProvider.
* AbsPlus: The source code for these classes must be stored in the binaryfuncsimp directory, and they are all part of the userfuncsimp.binaryfuncsimp package. AbsPlus provides a concrete implementation of BinaryFunc. It returns the result of ***abs(a) + abs(b)***. The code for AbsPlus is shown here:

***package*** userfuncsimp.binaryfuncsimp;

***import*** userfuncs.binaryfuncs.BinaryFunc;

***public class*** AbsPlus ***implements*** BinaryFunc { ***public*** ***String*** getName() { ***return*** "absPlus"; */\* Return name of this function\*/* }

*// Implement the AbsPlus function : Implement func() for absolute-value addition.*

***public*** ***int*** func(***int*** a, ***int*** b) { ***return*** Math.abs(a) + Math.abs(b); } }

AbsPlus implements ***func()*** such that it returns the result of adding the absolute values of ***a*** and ***b***. Notice that ***getName()*** returns the "absPlus" string. It identifies this function.

* AbsMinus: AbsMinus provides a *concrete implementation* of BinaryFunc. It returns the result of ***abs(a) - abs(b)***.

***package*** userfuncsimp.binaryfuncsimp;

***import*** userfuncs.binaryfuncs.BinaryFunc;

***public*** ***class*** AbsMinus ***implements*** BinaryFunc { ***public String*** getName() { ***return*** "absMinus"; /\* Return name of this function \*/ }

*// Implement the AbsMinus function : Implement func() for absolute-value subtraction.*

***public*** ***int*** func(***int*** a, ***int*** b) { ***return*** Math.abs(a) - Math.abs(b) } }

Here, ***func()*** is implemented to return the difference between the absolute values of a and b, and the string "***absMinus***" is returned by ***getName()***.

* Implementing providers: To obtain an instance of AbsPlus, the AbsPlusProvider is used. It implements BinFuncProvider and is shown here:

**package** userfuncsimp.binaryfuncsimp;

**import** userfuncs.binaryfuncs.\*;

**public** **class** AbsPlusProvider **implements** BinFuncProvider {

**public** **BinaryFunc** get() { **return** **new** AbsPlus (); } */\* Returns an AbsPlus object. \*/* }

***get()*** simply returns a new ***AbsPlus()*** object. [Though above is simple, it is important to point out that some service providers will be much more complex.]

* The provider for AbsMinus is called AbsMinusProvider and is:

**package** userfuncsimp.binaryfuncsimp;

**import** userfuncs.binaryfuncs;

**public** **class** AbsMinusProvider **implements** BinFuncProvider {

**public** **BinaryFunc** get() { **return new** AbsMinus(); } */\* Returns an AbsMinus object. \*/* }

Its ***get()*** method returns an object of ***AbsMinus***.

* The Module Definition Files: Next, two module definition files are needed. The first is for the userfuncs module. It is shown here:

***module userfuncs { exports userfuncs.binaryfuncs; }***

* This code must be contained in a module-info.java file that is in the userfuncs *module directory*. Notice that it exports the userfuncs.binaryfuncs package. This is the package that defines the BinaryFunc and BinFuncProvider *interfaces*.
* The second module-info.java file is shown next. It defines the module that contains the implementations. It goes in the userfuncsimp module directory.

**module** userfuncsimp { **requires** userfuncs;

**provides** userfuncs.binaryfuncs.BinFuncProvider **with**

userfuncsimp.binaryfuncsimp.AbsPlusProvider, userfuncsimp.binaryfuncsimp.AbsMinusProvider; }

This module requires userfuncs because that is where BinaryFunc and BinFuncProvider are contained, and those interfaces are needed by the implementations. The module provides BinFuncProvider implementations with the classes AbsPlusProvider and AbsMinusProvider.

* ***Demonstrate the*** Service Providers in MyModAppDemo ***inside*** ***main()***: For the use of the services and service providers, the ***main()*** method of MyModAppDemo is expanded to use AbsPlus and AbsMinus. It does so by loading them at run time by use of ***ServiceLoader.load()***. Here is the updated code:

**package** appstart.mymodappdemo;

**import** java.util.ServiceLoader; **import** appfuncs.simplefuncs.SimpleMathFuncs; **import** userfuncs.binaryfuncs.\*;

**public class** MyModAppDemo { **public static void main(String[] args)** {

*// First, use built-in functions as before*

**if**(SimpleMathFuncs.isFactor(2, 10)) **System.out.println**("2 is a factor of 10");

**System.out.println**("Smallest factor common to both 35 and 105 is " + SimpleMathFuncs.lcf(35, 105));

**System.out.printIn**("Largest factor common to both 35 and 105 is " + SimpleMathFuncs.gcf(35, 105));

*// Now, use service-based, user-defined operations.*

**ServiceLoader<BinFuncProvider>** ldr = ServiceLoader.load(BinFuncProvider.class); *// Get a service loader for binary functions.*

**BinaryFunc** binOp = null;

*// Find the provider for absPlus and obtain the function using for-each-for-loop*

**for**(**BinFuncProvider** bfp : ldr) { **if**(bfp.**get**().**getName**().**equals**("absPlus")) { binOp = bfp.get(); **break** ; }}

**if**(binOp != null**)** **System.out.printIn**("Result of absPlus function: " + binOp.func(12, -4));

**else** **System.out.printIn**("absPlus function not found");

binOp = **null**;

*// Now, find the provider for absMinus and obtain the function,*

**for** (**BinFuncProvider** bfp : ldr) { **if** (bfp.**get**().**getName**().**equals**("absMinus") ) { binOp = bfp.get(); **break**; } }

**if**(binOp != **null**) **System.out.printIn**("Result of absMinus function: " + binOp.func(12, -4));

**else** **System.out.printIn**("absMinus function not found"); }}

* A service loader for services of type ***BinFuncProvider*** is created with this statement:

***ServiceLoader<BinFuncProvider> ldr = ServiceLoader.load(BinFuncProvider.class);***

Notice that the type parameter to ServiceLoader is BinFuncProvider. This is also the type used in the call to load( ). This means that ***providers that implement*** this interface will be found. Thus, after this statement executes, BinFuncProvider classes in the module will be available through ldr. In this case, both AbsPlusProvider and AbsMinusProvider will be available.

* Next, a reference of type BinaryFunc called binOp is declared and initialized to null. It will be used to ***refer to an implementation*** that supplies a ***specific type*** of *binary function*.
* Next, the following loop searches ldr for one that has the "absPlus" name.

**for**(**BinFuncProvider** bfp : ldr) { **if**(bfp.**get**().**getName**().**equals**("absPlus")) { binOp = bfp.get(); **break** ; }}

Here, a for-each loop iterates through ldr. Inside the loop, the name of the function supplied by the provider is checked. If it matches "absPlus", that function is assigned to binOp by calling the provider’s ***get()*** method.

* If the function is found, it is executed by this statement: ***if****(binOp != null****)******System.out.printIn****("Result of absPlus function: " + binOp.func(12, -4));*

Because binOp refers to an instance of AbsPlus, the call to ***func()*** performs an absolute value addition. A similar sequence is used to find and execute AbsMinus.

* Module definition for the main application module: Because MyModAppDemo now uses BinFuncProvider, its module definition file must include a uses statement that specifies this fact. Recall that MyModAppDemo is in the appstart module. Therefore, you must change the ***module-info.java*** file for appstart as:

**module** appstart { **requires** appfuncs; **requires** userfuncs;

**uses** userfuncs.binaryfuncs.BinFuncProvider; */\* appstart now uses BinFuncProvider \*/* }

* Compile and Run: execute the commands: javac -d appmodules --module-source-path appsrc appsrc\userfuncsimp\module-info.java

appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java

java --module-path appmodules -m appstart/appstart.mymodappdemo.MyModAppDemo

* If either the provides statement in the ***userfuncsimp module*** or the uses statement in the ***appstart module*** were missing, the application would fail.

**10. 19 Runtime MODULE FEATURES**

There are three more features: open module, the opens statement, and the use of requires static. Each of these features is designed to handle a specialized situation.

* Open Modules: By default, the types in a module’s packages are accessible only if they are explicitly exported via an exports statement. But there can be circumstances in which it is useful to enable runtime access to all packages in the module, whether a package is exported or not. To allow this, you can create an open module. An open module is declared by preceding the module keyword with the open modifier, as shown here:

***open module moduleName {*** */\* module definition \*/* ***}***

* In an open module, types in all packages are accessible at run time. Understand, however, that only those packages that are explicitly exported are available at compile time. Thus, the open modifier affects only runtime accessibility.
* The primary reason for an open module is to enable the packages in the module to be accessed through ***reflection***. ***Reflection*** is the feature that lets a program analyze code at run time. Reflection can be quite important to certain types of programs that require runtime access to a thirdparty library.
* The opens Statement: It is possible for a module to open a specific package for runtime access by other modules and for reflective access rather than opening an entire module. To do so, use the opens statement, shown here: ***opens packageName;***
* Here, packageName specifies the package to open. It is also possible to include a to clause, which names those modules for which the package is opened.
* It is important to understand that opens does not grant compile-time access. It is used only to open a package for runtime and reflective access.
* An opens statement cannot be used in an open module. Remember, all ***packages*** in an ***open module*** are already ***open***.
* requires static: As you know, ***requires*** specifies a dependence that, by default, is enforced both during compilation and at run time. However, it is possible to relax this requirement in such a way that a module is not required at run time. This is accomplished by use of the static modifier in a ***requires*** statement. For example, this specifies that mymod is required for compilation, but not at run time: ***requires static mymod;***
* Here, static makes mymod optional at run time. This can be helpful in a situation in which a program can utilize functionality if it is present, but not require it.

**10.20 Module graph**

During compilation, the compiler resolves the dependence relationships between modules by creating a module graph that represents the dependences. The process ensures that all *dependences* are *resolved*, including those that occur indirectly. For example, if module A requires module B and B requires module C, then the module graph will contain module C even if A does not use it directly.

* Module graphs can be depicted visually in a drawing to illustrate the relationship between modules. Here is a simple example. It is the graph for the Example 15, (Because java.base is automatically included, it is not shown in the diagram.):
* In Java, the arrows point from the dependent module to the required module. Thus, a drawing of a module graph depicts what modules have access to what other modules. Frankly, only the smallest applications can have their module graphs visually represented because of the complexity typically involved in many commercial applications.

**CONTINUING YOUR STUDY OF MODULES**

Beginning with JDK 9, the JDK includes the jlink tool that assembles a modular application into a runtime image that has only those modules related to the application. This saves both space and download time. A modular application can be packaged into a JAR file. (JAR stands for Java ARchive. It is a file format typically used for application deployment.) As a result, the jar tool now has options that support modules. For example, it can now recognize a module path. A JAR file that contains a module-info.class file is called a modular JAR file. For specialized advanced work with modules, you will want to learn about layers of modules, automatic modules, and the technique by which modules can be added during compilation or execution.