Fundamentals of Java

Session: 7

Methods and Access Specifiers

Objectives

- Describe methods
- Explain the process of creation and invocation of methods
- Explain passing and returning values from methods
- Explain variable argument methods
- Describe the use of Javadoc to lookup methods
- Describe access specifiers and the types of access specifiers
- Explain the use of access specifiers with methods
- Explain the concept of method overloading
- Explain the use of this keyword

Introduction

- Methods in Java are such a feature that allows grouping of statements and execution of a specific set of statements instead of executing the entire program.
- Java provides a set of access specifiers that can help the user to restrict access to certain methods.

Methods 1-3

A Java method can be defined as a set of statements grouped together for performing a specific task.

For example, a call to the `main()` method which is the point of entry of any Java program, will execute all the statements written within the scope of the `main()` method.

- The syntax for declaring a method is as follows:

Syntax

```
modifier return_type method_name([list_of_parameters]) {
// Body of the method
}
```

where,

`modifier`: Specifies the visibility of the method. Visibility indicates which object can access the method. The values can be `public`, `private`, or `protected`.

`return_type`: Specifies the data type of the value returned by the method.

`method_name`: Specifies the name of the method.

`list_of_parameters`: Specifies the comma-delimited list of values passed to the method.

Methods 2-3

- Generally, a method declaration has the following six components, in order:

1. Modifiers such as `public`, `private`, and `protected`.
2. A return type that indicates the data type of the value returned by the method.
   - The return type is set to `void` if the method does not return a value.
3. The method name that is specified based on certain rules. A method name:
   - cannot be a Java keyword
   - cannot have spaces
   - cannot begin with a digit
   - cannot begin with any symbol other than a `$` or _
   - can be a verb in lowercase
   - can be a multi-word name that begins with a verb in lowercase, followed by adjectives or nouns
   - can be a multi-word name with the first letter of the second word and each of the following words capitalized
   - should be descriptive and meaningful

Methods 3-3

- Some valid method names are **`add`**, **`_view`**, **`$calc`**, **`add_num`**, **`setFirstName`**, **`compareTo`**, **`isValid`**, and so on.

4
- Parameter list in parenthesis is separated with a comma delimiter.
- Each parameter is preceded by its data type.
- If there are no parameters, an empty parenthesis is used.

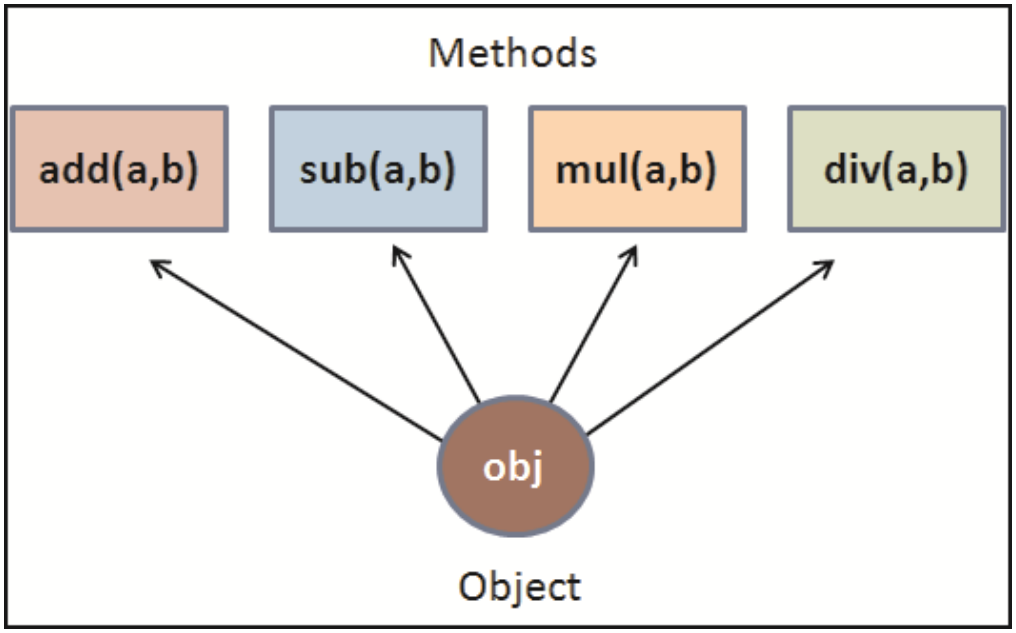
5
- An exception list that specifies the names of exceptions that can be thrown by the method.
- An exception is an event encountered during the execution of the program, disrupting the flow of program execution.

6
- Method body consists of a set of statements enclosed between curly braces '{}'.
- Method body can have variables, method calls, and even classes.

- The two components of a method declaration namely, the method name and the parameter types comprise the method signature.

Creating and Invoking Methods 1-7

- Methods help to segregate tasks to provide modularity to the program.
- A program is modular when different tasks in a program are grouped together into modules or sections.
- For example, to perform different types of mathematical operations such as addition, subtraction, multiplication, and so on, a user can create individual methods as shown in the following figure:



- The figure shows an object named **`obj`** accessing four different methods namely, **`add(a,b)`**, **`sub(a,b)`**, **`mul(a,b)`**, and **`div(a,b)`** for performing the respective operations on two numbers.

Creating and Invoking Methods 2-7

- To create a method that adds two numbers, the user can write a method as depicted in the following code snippet:

```
public void add(int num1, int num2){
   int num3; // Declare a variable
   num3 = num1 + num2; // Perform the addition of numbers
   System.out.println("Addition is " + num3); // Print the result
}
```

- Defines a method named **`add()`** that accepts two integer parameters **`num1`** and **`num2`**.
- Has declared the method with the `public` access specifier which means that it can be accessed by all objects.
- Has set the return type to `void`, indicating that the method does not return anything.
- Statement '**`int num3;`**' is a declaration of an integer variable named **`num3`**.
- Statement '**`num3 = num1 + num2;`**' is an addition operation performed on parameters **`num1`** and **`num2`** using the arithmetic operator '**+**'.
- Result is stored in a third variable **`num3`** by using the assignment operator '**=**'.
- Finally, '**`System.out.println("Addition is "+ num3);`**' is used to print the value of variable **`num3`**.
- Method signature is '**`add(int, int)`**'.

Creating and Invoking Methods 3-7

- To use a method, it must be called or invoked. When a program calls a method, the control is transferred to the called method.
- The called method executes and returns control to the caller.
- The call is returned back after the return statement of a method is executed or when the closing brace is reached.
- A method can be invoked in one of the following ways:

If the method returns a value, then, a call to the method results in return of some value from the method to the caller. For example,

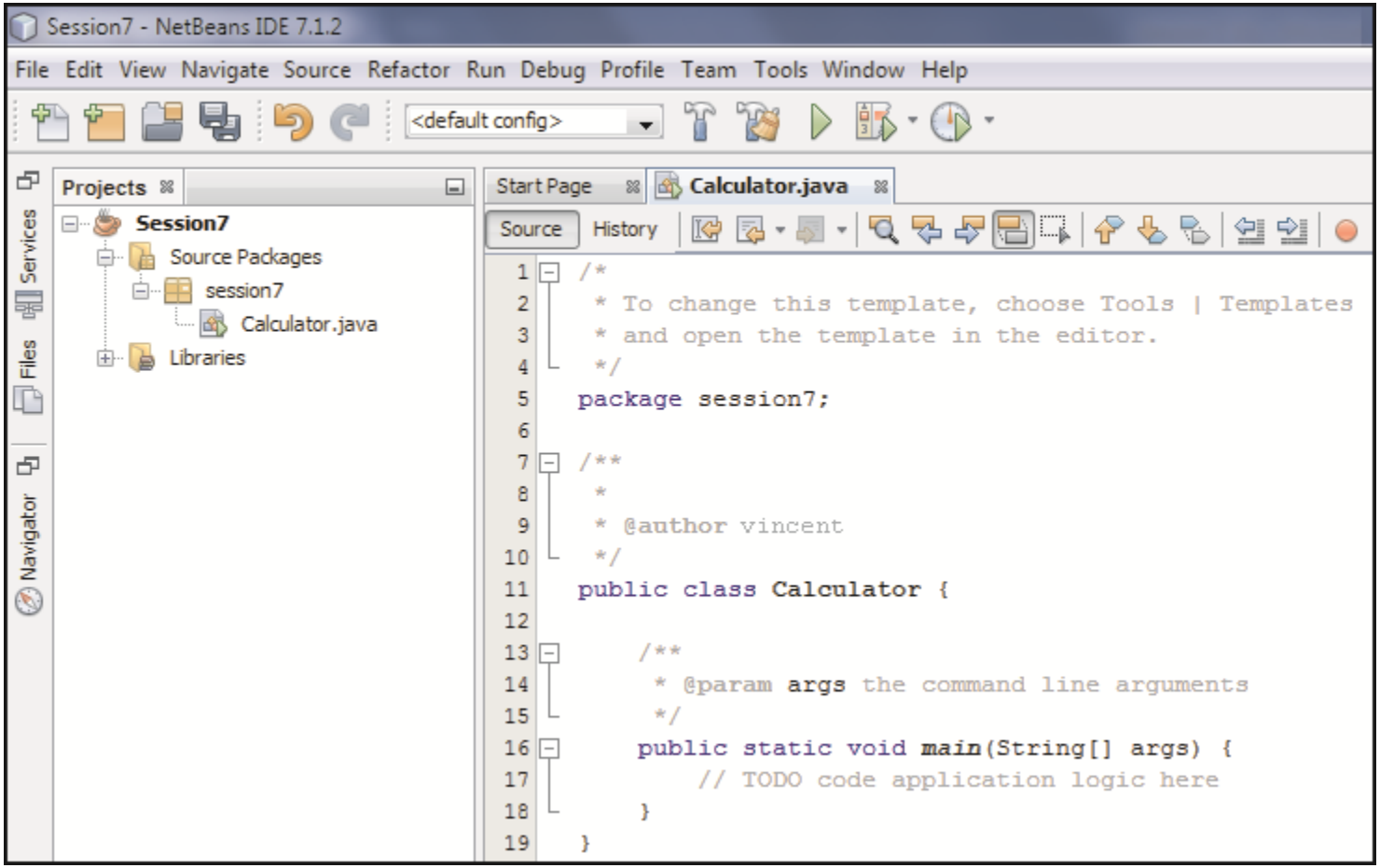
```
int result = obj.add(20, 30);
```

If the method's return type is set to `void`, then, a call to the method results in execution of the statements within the method without returning any value to the caller.

For example, a call to the method would be `obj.add(23,30)` without anything returned to the caller.

Creating and Invoking Methods 4-7

- Consider the project **`Session7`** created in the NetBeans IDE as shown in the following figure:



- The project consists of a package named **`session7`** with the **`Calculator`** class that has the **`main()`** method.
- Several methods for mathematical operations can be added to the class.

Creating and Invoking Methods 5-7

- Following code snippet demonstrates an example of creation and invocation of methods:

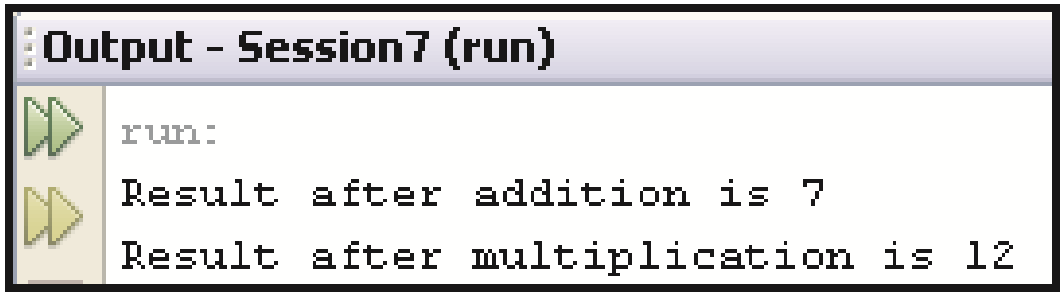
```
package session7;
  public class Calculator {
  // Method to add two integers
  public void add(int num1, int num2) {
    int num3;
    num3 = num1 + num2;
  System.out.println("Result after addition is " + num3);
  }
  // Method to subtract two integers
  public void sub(int num1, int num2) {
    int num3;
    num3 = num1 - num2;
    System.out.println("Result after subtraction is " + num3);
  }
  // Method to multiply two integers
  public void mul(int num1, int num2) {
    int num3;
    num3 = num1 * num2;
    System.out.println("Result after multiplication is " + num3);
  }
```

Creating and Invoking Methods 6-7

```
// Method to divide two integers
public void div(int num1, int num2) {
   int num3;
   num3 = num1 / num2;
   System.out.println("Result after division is " + num3);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
   // Instantiate the Calculator class
   Calculator objCalc = new Calculator();
   // Invoke the methods with appropriate arguments
   objCalc.add(3, 4);
   objCalc.mul(3, 4);
 }
}
```

Creating and Invoking Methods 7-7

- Class **Calculator** consists of methods such as **add()**, **sub()**, **mul()**, and **div()** that are used to perform the respective operations.
- Each method accepts two integers as parameters.
- The **main()** method creates an object, **objCalc** of class **Calculator**.
- The object **objCalc** uses the dot '**.**' operator to invoke the **add()** and **mul()** methods.
- Following figure shows the output of the program:

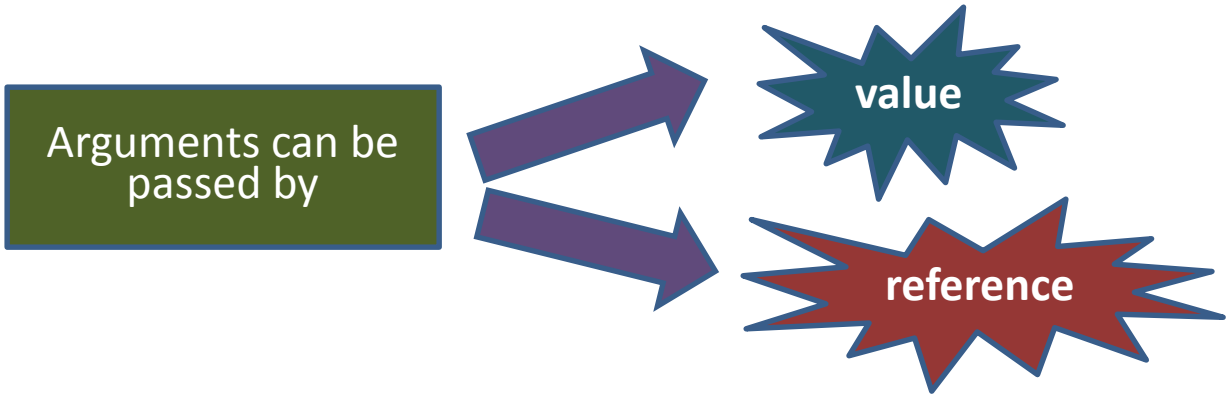


Passing and Returning Values from Methods

Parameters	Arguments
Parameters are the list of variables specified in a method declaration.	Arguments are the actual values that are passed to the method when it is invoked.

When a method is invoked, the type and order of arguments that are passed must match the type and order of parameters declared in the method.

A method can accept primitive data types such as `int`, `float`, `double`, and so on as well as reference data types such as arrays and objects as a parameter.



Passing Arguments by Value 1-2

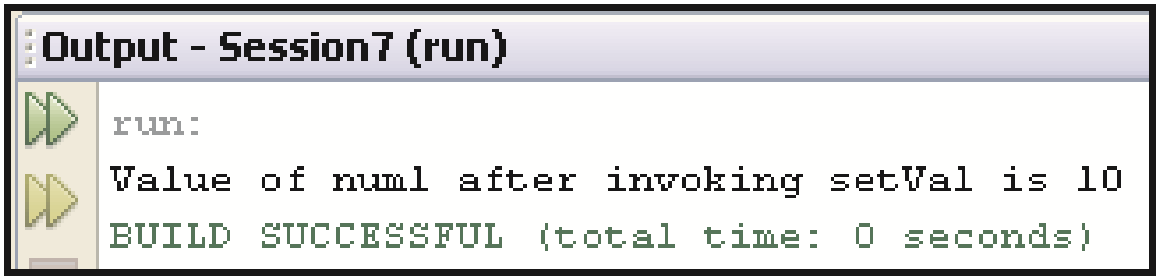
- When arguments are passed by value it is known as call-by-value and it means that:
  - A copy of the argument is passed from the calling method to the called method.
  - Changes made to the argument passed in the called method will not modify the value in the calling method.
  - Variables of primitive data types such as `int` and `float` are passed by value.
- Following code snippet demonstrates an example of passing arguments by value:

```
package session7;
public class PassByValue {
   // method accepting the argument by value
   public void setVal(int num1) {
   num1 = num1 + 10;
}
```

Passing Arguments by Value 2-2

```
  public static void main(String[] args) {
    // Declare and initialize a local variable
    int num1 = 10;
    // Instantiate the PassByValue class
    PassByValue obj = new PassByValue();
    // Invoke the setVal() method with num1 as parameter
    obj.setVal(num1);
    // Print num1 to check its value
    System.out.println("Value of num1 after invoking setVal is "+ num1);
  }
}
```

- Following figure shows the output of the code:



- Output shows that the value of **num1** is still **10** even after invoking **setVal()** method when the value had been incremented.
- This is because, **num1** was passed by value.

Passing Arguments by Reference 1-3

- When arguments are passed by reference it means that:
  - The actual memory location of the argument is passed to the called method and the object or a copy of the object is not passed.
  - The called method can change the value of the argument passed to it.
  - Variables of reference types such as objects are passed to the methods by reference.
  - There are two references of the same object namely, argument reference variable and parameter reference variable.
- Following code snippet demonstrates an example of passing arguments by reference:

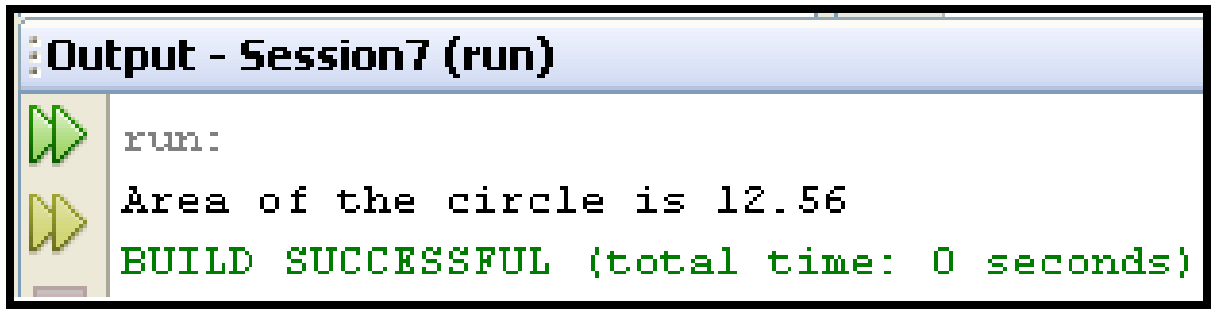
```
package session7;
class Circle{
   // Method to retrieve value of PI
   public double getPI(){
      return 3.14;
   }
}
```

Passing Arguments by Reference 2-3

```
// Define another class PassByRef
public class PassByRef{
   // Method to calculate area of a circle that
   // takes the object of class Circle as a parameter
   public void calcArea(Circle objPi, double rad){
     // Use getPI() method to retrieve the value of PI
     double area= objPi.getPI() * rad * rad;
     // Print the value of area of circle
     System.out.println("Area of the circle is "+ area);
   }
   public static void main(String[] args){
     // Instantiate the PassByRef class
     PassByRef p1 = new PassByRef();
     // Invoke the calcArea() method with object of class Circle as
     // a parameter
     p1.calcArea(new Circle(), 2);
   }
}
```

Passing Arguments by Reference 3-3

- Following figure shows the output of the code:



- Note that the value of PI is passed by reference and not by value.

Returning Values from Methods 1-2

A method will return a value to the invoking method only when all the statements in the invoking method are complete, or when it encounters a return statement, or when an exception is thrown.

The `return` statement is written within the body of the method to return a value.

A `void` method will not have a return type specified in its method body.

A compiler error is generated when a `void` method returns a value.

You can store the value in a variable and specify the name of the variable with the `return` keyword.

Returning Values from Methods 2-2

- For example, the class **Circle** and its **getPI()** method can be modified as shown in code snippet:

```
public class Circle {
   // Declare and initialize value of PI
   private double PI = 3.14;
   // Method to retrieve value of PI
   public double getPI(){
      return PI;
   }
}
```

- In the modified class **Circle**, the value **3.14** is stored in a `private double` variable **PI**.
- Later, the method **getPI()** returns the value stored in the variable **PI** instead of the constant value **3.14**.

Declaring Variable Argument Methods 1-3

Java provides a feature called `varargs` to pass variable number of arguments to a method.

`varargs` is used when the number of a particular type of argument that will be passed to a method is not known until runtime.

It serves as a shortcut to creating an array manually.

To use `varargs`, the type of the last parameter is followed by ellipsis (...), then, a space, followed by the name of the parameter.

This method can be called with any number of values for that parameter, including none.

- The syntax of a variable argument method is as follows:

Syntax

```
<method_name>(type … variableName){
// method body
}
```

where,

'…' : Indicates the variable number of arguments.

Declaring Variable Argument Methods 2-3

- Following code snippet demonstrates an example of a variable argument method:

```
package session7;
public class Varargs {
  // Variable argument method taking variable number of integer arguments
  public void addNumber(int...num) {
    int sum=0;
    // Use for loop to iterate through num
    for(int i:num) {
      // Add up the values
      sum = sum + i;
    }
    // Print the sum
    System.out.println("Sum of numbers is "+ sum);
  }
  public static void main(String[] args) {
    // Instantiate the Varargs class
    Varargs obj = new Varargs();
    // Invoke the addNumber() method with multiple arguments
    obj.addNumber(10,30,20,40);
  }
}
```

Declaring Variable Argument Methods 3-3

- Following figure shows the output of the code:

- The class **`Varargs`** consists of a method called **`addNumber(int…num)`**.
- The method accepts variable number of arguments of type integer.
- The method uses the enhanced for loop to iterate through the variable argument parameter **`num`** and adds each value with the variable **`sum`**.
- Finally, the method prints the value of **`sum`**.
- The `main()` method creates an object of the class and invokes the **`addNumber()`** method with multiple arguments of type integer.
- The output displays `100` after adding up the numbers.

Using Javadoc to Lookup Methods of a Class 1-11

- Java provides a JDK tool named **Javadoc** that is used to generate API documentation as an HTML page from declaration and documentation comments.
- These comments are descriptions of the code written in a program.
- The different terminologies used while generating **javadoc** are as follows:

API documentation or API docs

- Are the online or hard copy descriptions of the API that are primarily intended for the programmers.
- API specification consists of all assertions for a proper implementation of the Java platform to ensure that the 'write once, run anywhere' feature of Java is retained.

Documentation comments or doc comments

- Are special comments in the Java source code.
- Are written within the /** … */ delimiters.
- Are processed by the Javadoc tool for generating the API docs.

Using Javadoc to Lookup Methods of a Class 2-11

- The four types of source files from which Javadoc tool can generate output are as follows:
  - Java source code files (`.java`) which consist of the field, class, constructor, method, and interface comments.
  - Package comment files that consist of package comments.
  - Overview comment files that contain comments about set of packages.
  - Miscellaneous files that are unprocessed such as images, class files, sample source codes, and any other file that is referenced from the previous files.
- A doc comment is written in HTML and it must precede a field, class, method, or constructor declaration.
- The doc comment consists of two parts namely, a description and block tags.
- For example, consider the class given in the following code snippet:

```
public class Circle {
  private double PI=3.14;
  public double calcArea(double rad){
    return (3.14 * rad * rad);
  }
}
```

Using Javadoc to Lookup Methods of a Class 3-11

- The code consists of a class **`Circle`**, a variable **`PI`**, and a method **`calcArea()`** that accepts radius as a parameter.
- Now, a doc comment for **`calcArea()`** method can be written as depicted in the following code snippet:

```
/**
 * Returns the area of a circle
 *
 * @param rad a variable indicating radius of a circle
 * @return the area of the circle
 * @see PI
 */
```

- First statement is the method description.
- `@param`, `@return`, and `@see` are block tags that refer to the parameters and return value of the method.
- A blank comment line must be provided between the description line and block tags.

Using Javadoc to Lookup Methods of a Class 4-11

- The HTML generated from running the **Javadoc** tool is as follows:

calcArea

```
public double calcArea(double rad)
```

Returns the area of a circle

Parameters:

rad – a variable storing the radius of the circle

Returns:

the area of a circle

See Also:

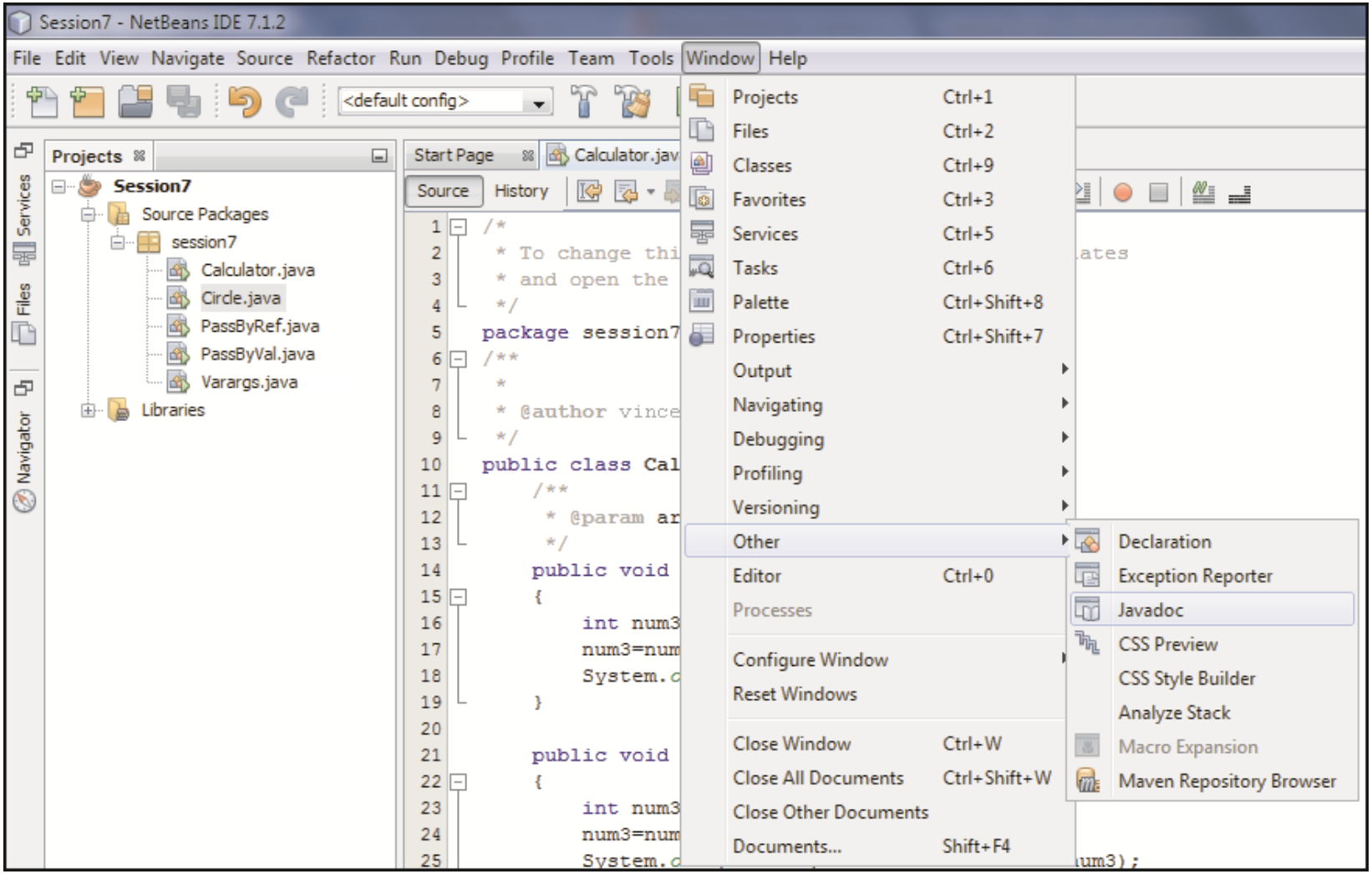
PI

- To use **Javadoc** tool to lookup methods of a class, perform the following steps:

1. Open the **`Calculator`** class created earlier.
2. Open **Javadoc** widow by clicking **Window** → **Other** → **Javadoc**.

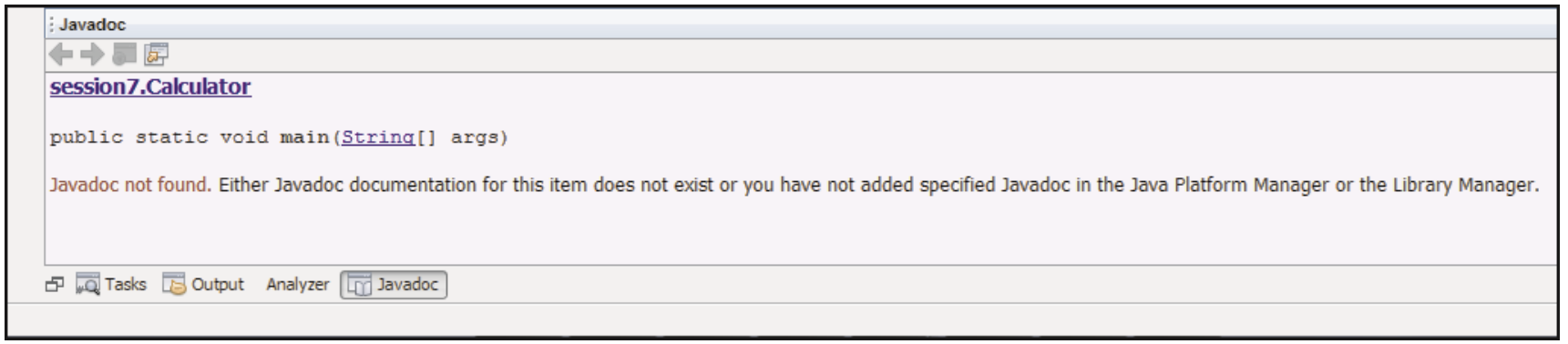
Using Javadoc to Lookup Methods of a Class 5-11

- This is shown in the following figure:



Using Javadoc to Lookup Methods of a Class 6-11

- Following figure shows the **Javadoc** window opened at the bottom:



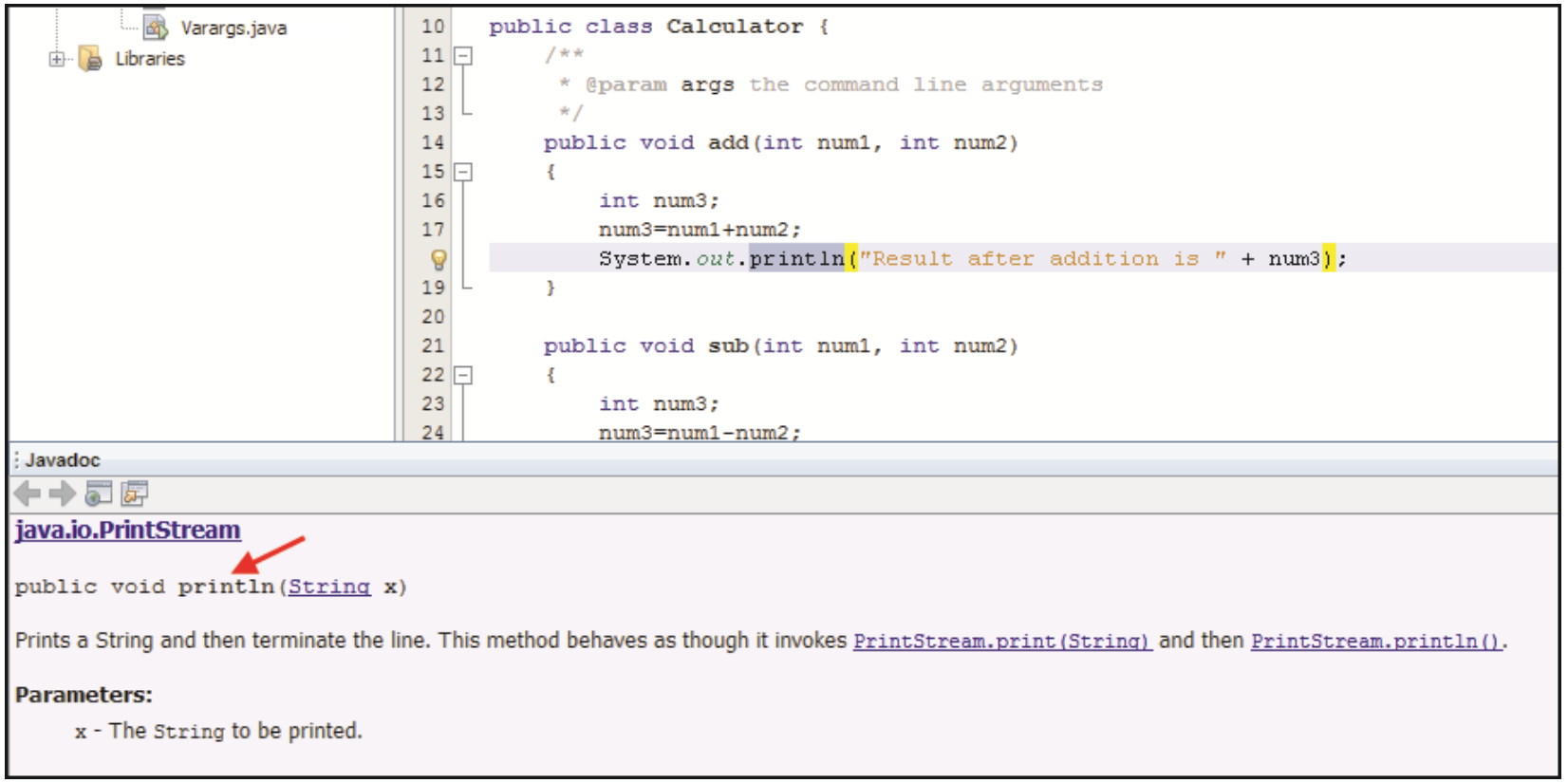
- The **Javadoc** window shows the description of the **`Calculator`** class.
- However, since **javadoc** for the class is not generated still, it gives a message that '**`Javadoc not found`**'.

3

- Select the `println()` method of `System.out.println()` statement of the **`add()`** method and then, open the **Javadoc** window.

Using Javadoc to Lookup Methods of a Class 7-11

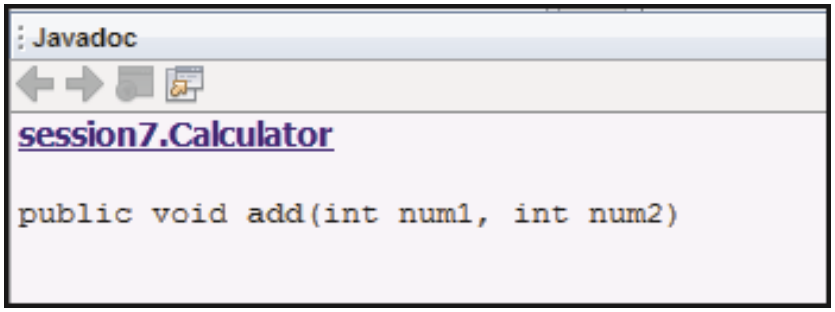
- The window will show the built-in Java documentation of the `println()` method along with its description and parameters as defined in the javadoc.
- This is shown in the following figure:



Using Javadoc to Lookup Methods of a Class 8-11

4

- Select the **`add()`** method. Again, no details about add method will be displayed since **javadoc** does not exist for it as shown in the following figure:



5

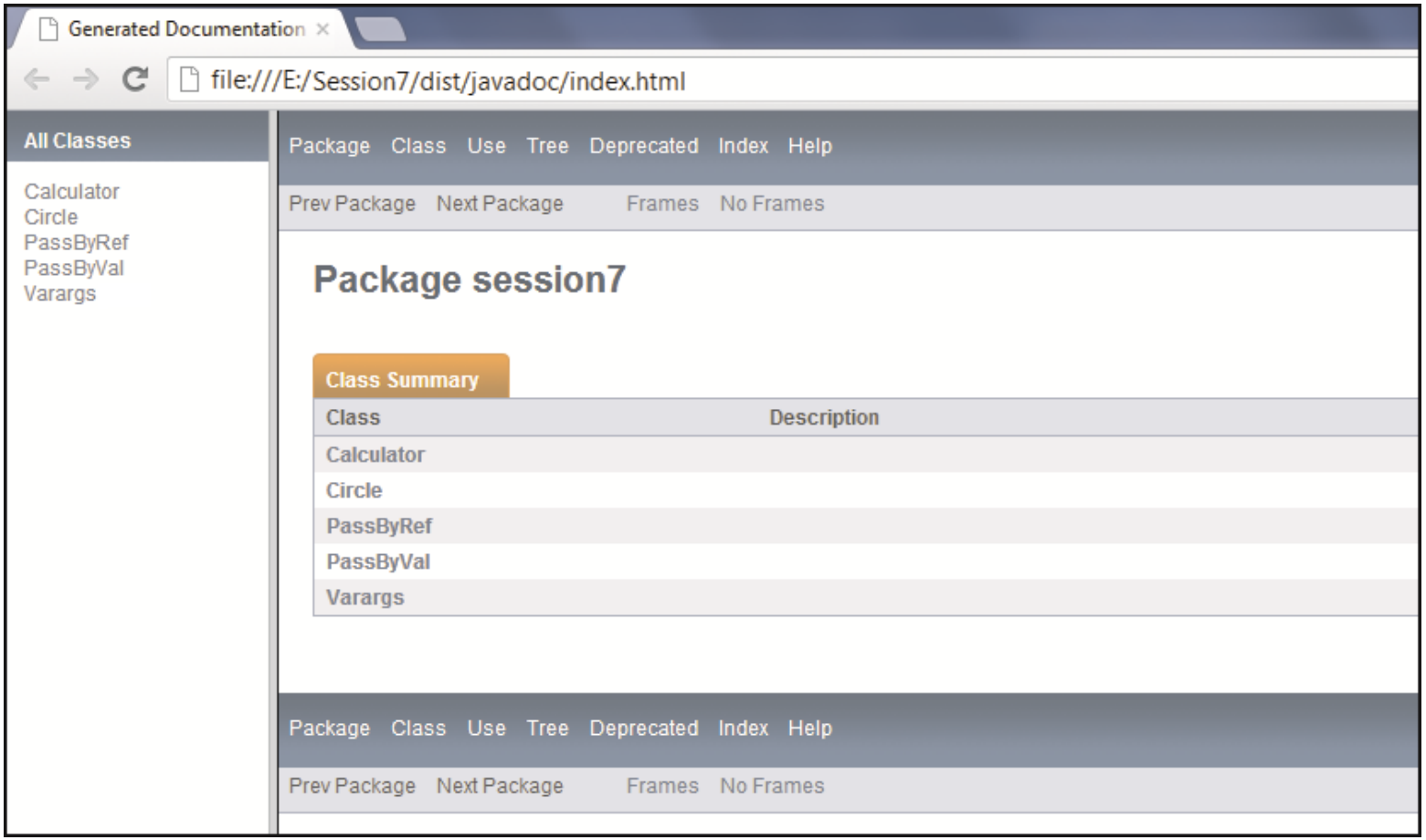
- Type the following **javadoc** comments above the **`add()`** method:

```
/**
 * Displays the sum of two integers
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @return void
 */
```

Using Javadoc to Lookup Methods of a Class 9-11

6

- Click **Run** → **Generate Javadoc**. The NetBeans IDE generates the **javadoc** for the project **`Session7`** and displays it in the browser as shown in the following figure:

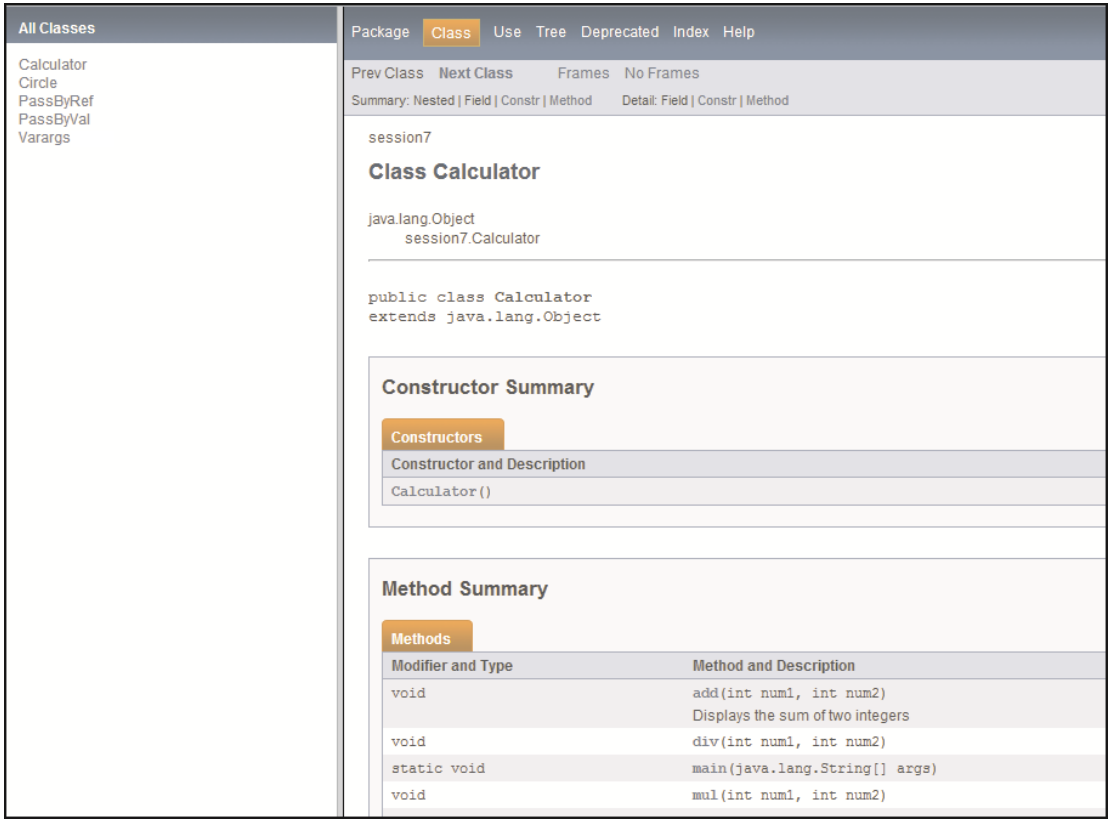


Using Javadoc to Lookup Methods of a Class 10-11

- The **javadoc** lists the various classes available in the selected package.
- To move to the next or previous package, one can click the **PrevPackage** and **NextPackage** links on the page.

7

- Click the **`Calculator`** class under the **Class Summary** tab.
- The javadoc of the **`Calculator`** class is shown in the following figure:

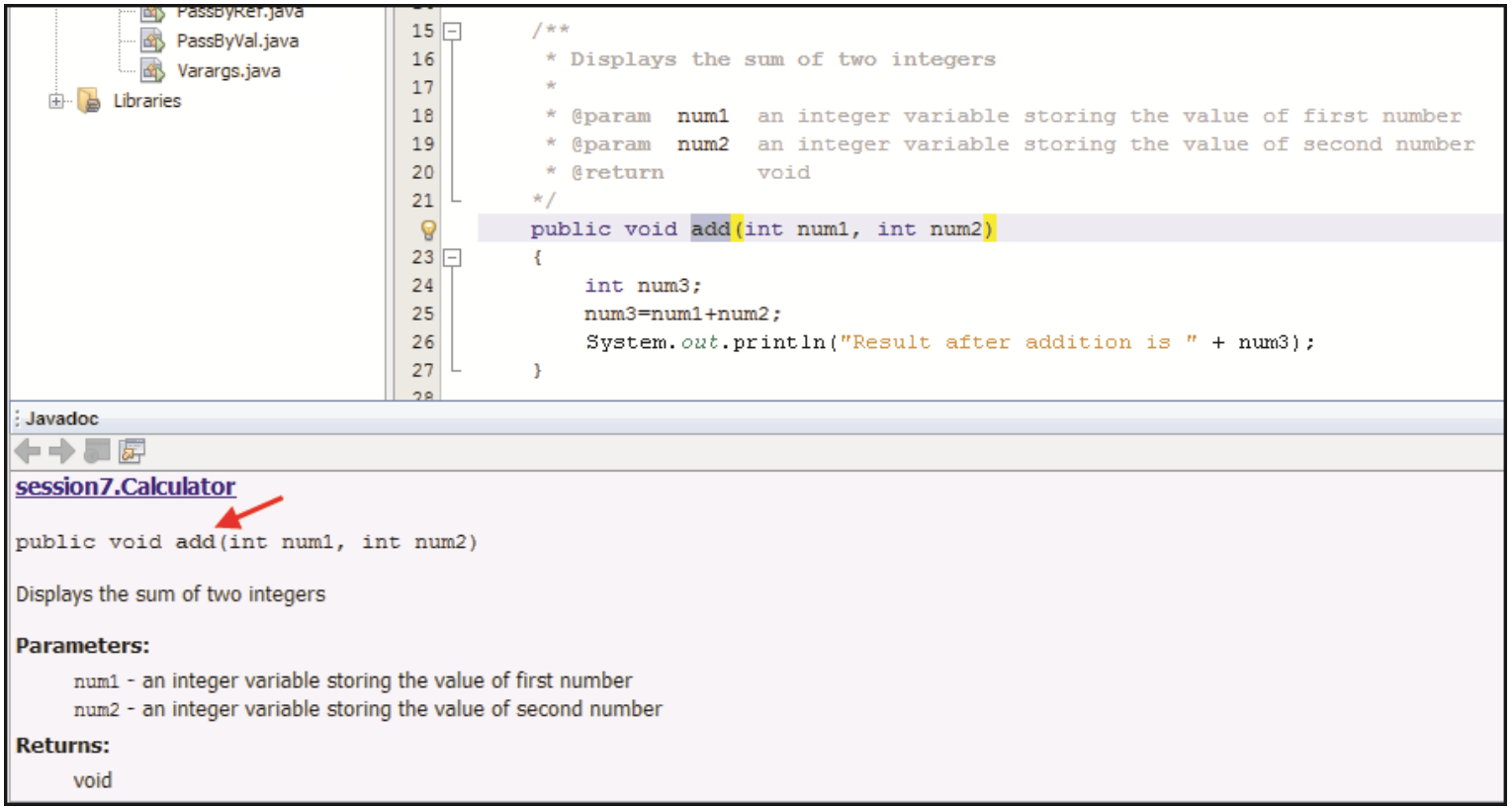


Using Javadoc to Lookup Methods of a Class 11-11

- Similarly, **javadoc** for other classes can also be viewed.
- This shows the structure of the class, its constructor, and its various methods.

8

- Now, select the **`add()`** method in the NetBeans IDE and open the **Javadoc** window.
- The window will show the javadoc created for the **`add()`** method as shown in the following figure:



Overview of Access Specifiers

- Java provides a number of access specifiers or modifiers to set the level of access for a class and its members such as fields, methods, and constructors within the class.
- This is also known as the visibility of the class, field, and methods.
- When no access specifier is mentioned for a class member, the default accessibility is package or default.
- Using access specifiers provides the following advantages:

- Access specifiers help to control the access of classes and class members.
- Access specifiers help to prevent misuse of class details as well as hide the implementation details that are not required by other classes.
- The access specifiers also determine whether classes and the members of the classes can be invoked by other classes or interfaces.
- Accessibility affects inheritance and how members are inherited by the subclass.
- A package is always accessible by default.

Types of Access Specifiers 1-3

- Java comes with four access specifiers namely, `public`, `private`, `protected`, and default.

public

- The `public` access specifier is the least restrictive of all access specifiers.
- A field, method, or class declared `public` is visible to any class in a Java application in the same package or in another.

private

- The `private` access specifier cannot be used for classes and interfaces as well as fields and methods of an interface.
- Fields and methods declared `private` cannot be accessed from outside the enclosing class.

protected

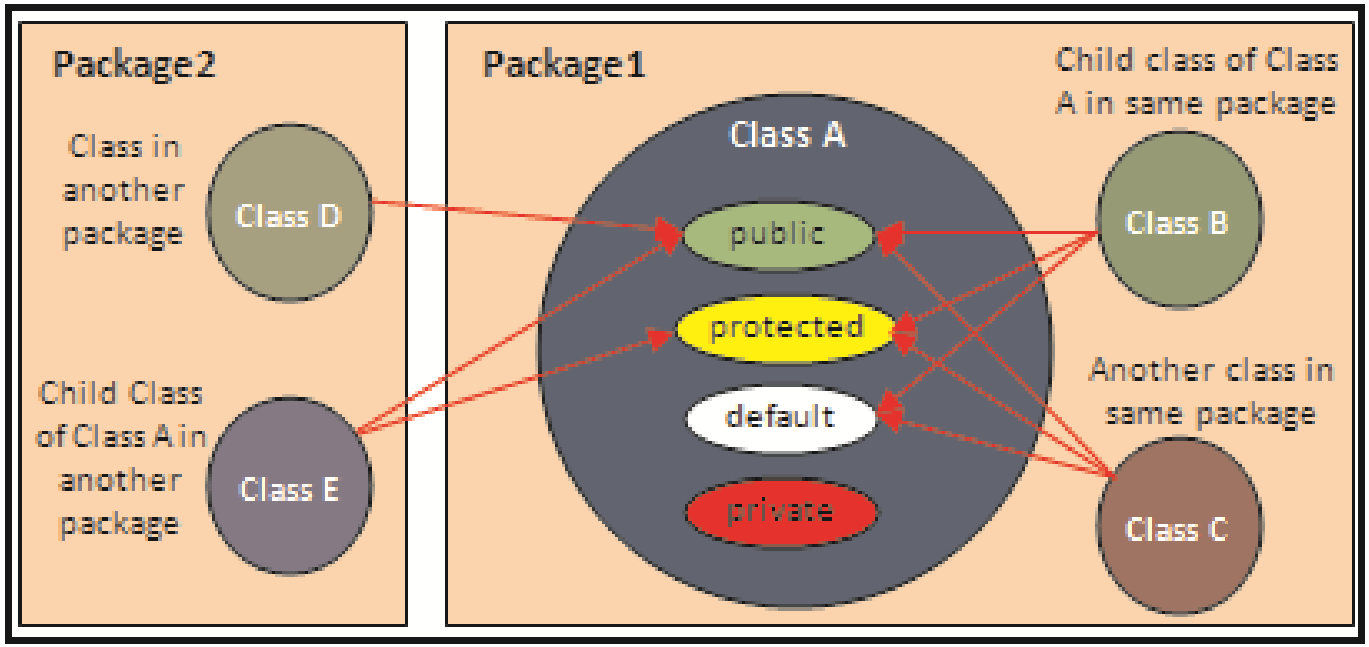
- The `protected` access specifier is used with classes that share a parent-child relationship which is referred to as inheritance.
- The `protected` keyword cannot be used for classes and interfaces as well as fields and methods of an interface.
- Fields and methods declared `protected` in a parent or super class can be accessed only by its child or subclass in another packages.
- Classes in the same package can also access protected fields and methods, even if they are not a subclass of the `protected` member's class.

Types of Access Specifiers 2-3

Default

- The default access specifier is used when no access specifier is present.
- The default specifier gets applied to any class, field, or method for which no access specifier has been mentioned.
- With default specifier, the class, field, or method is accessible only to the classes of the same package.
- The default specifier is not used for fields and methods within an interface.

- Following figure shows the various access specifiers:



Types of Access Specifiers 3-3

- Following table shows the access level for different access specifiers:

Access Specifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
No modifier (default)	Y	Y	N	N
`private`	Y	N	N	N

- The first column states whether the class itself has access to its own data members.
- As can be seen, a class can always access its own members.
- The second column states whether classes within the same package as the owner class (irrespective of their parentage) can access the member.
- As can be seen, all members can be accessed except `private` members.
- The third column states whether the subclasses of a class declared outside this package can access a member.
- In such cases, `public` and `protected` members can be accessed.
- The fourth column states whether all classes can access a data member.

Rules for Access Control

- Java has rules and constraints for usage of access specifiers as follows:

While declaring members, a `private` access specifier cannot be used with `abstract`, but it can be used with `final` or `static`.

No access specifier can be repeated twice in a single declaration.

A constructor when declared `private` will be accessible in the class where it was created.

A constructor when declared `protected` will be accessible within the class where it was created and in the inheriting classes.

`private` cannot be used with fields and methods of an interface.

The most restrictive access level must be used that is appropriate for a particular member.

Mostly, a `private` access specifier is used at all times unless there is a valid reason for not using it.

Avoid using `public` for fields except for constants.

Using Access Specifiers with Variables and Methods 1-6

- The access specifiers discussed can be used with variables and methods of a class to restrict access from other classes.
- Following code snippet demonstrates an example of using access specifiers with variables and methods:

```
package session7;
public class Employee {
  // Variables with default access
  int empID; // Variable to store employee ID
  String empName; // Variable to store employee name
  // Variables with private and protected access
  private String SSN; // Variable to store social security number
  protected String empDesig; // Variable to store designation
  /**
   * Parameterized constructor
   *
   * @param ID an integer variable storing the employee ID
   * @param name a String variable storing the employee name
   * @return void
   */
  public Employee(int ID, String name){
    empID = ID;
```

Using Access Specifiers with Variables and Methods 2-6

```
    empName = name;
  }
  // Define public methods
  /**
   * Returns the value of SSN
   *
   * @return String
   */
  public String getSSN(){ // Accessor for SSN
    return SSN;
  }
  /**
   * Sets the value of SSN
   *
   * @param ssn a String variable storing the social security number
   * @return void
   */
  public void setSSN(String ssn) { // Mutator for SSN
    SSN = ssn;
  }
```

Using Access Specifiers with Variables and Methods 3-6

```
/**
 * Sets the value of Designation
 *
 * @param desig a String variable storing the employee designation
 * @return void
 */
public void setDesignation(String desig) { // public method
  empDesig = desig;
}

/**
 * Displays employee details
 *
 * @return void
 */
public void display(){ // public method
  System.out.println("Employee ID is "+ empID);
  System.out.println("Employee name is "+ empName);
  System.out.println("Designation is "+ empDesig);
  System.out.println("SSN is "+ SSN);
}
```

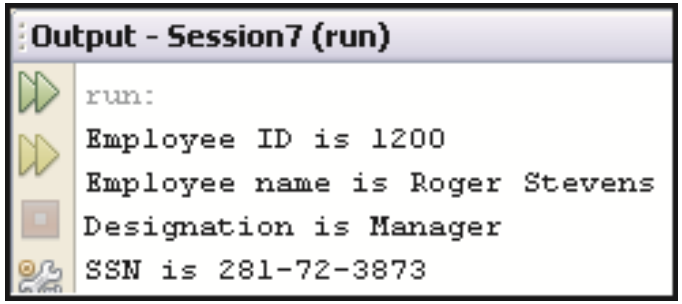
Using Access Specifiers with Variables and Methods 4-6

```
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // Instantiate the Employee class
    Employee objEmp1 = new Employee(1200,"Roger Stevens");
    // Assign values to public variables
    objEmp1.empDesig = "Manager";
    objEmp1.SSN = "281-72-3873";
    // Invoke the public method
    objEmp1.display();
  }
}
```

- **`main()`** method creates an object of **`Employee`** class with values of **`empID`** and **`empName`**.
- Next, the values of **`empDesig`** and **`SSN`** are specified by directly accessing the variables with the object **`objEmp1`** even though **`empDesig`** is `protected` and **`SSN`** is `private`.
- This is because, **`objEmp1`** is an object present in the same class.
- So, **`objEmp1`** has access to data members with any access specifier.
- The **`display()`** method is used to display all the values as shown in the output.

Using Access Specifiers with Variables and Methods 5-6

- Following figure shows the output of the code:



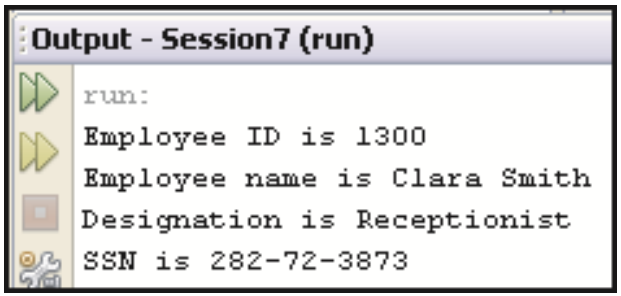
- Following code snippet demonstrates the use of access specifiers in another class named **EmployeeDetails** but in the same package as **Employee** class:

```
public class EmployeeDetails {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // Instantiate the Employee class within EmployeeDetails class
    Employee objEmp = new Employee(1300,"Clara Smith");
    // Assign value to protected variable
    objEmp.empDesig="Receptionist";
    // Use the mutator method to set the value of private variable
    objEmp.setSSN("282-72-3873");
    // Invoke the public method
    objEmp.display();
```

Using Access Specifiers with Variables and Methods 6-6

```
    }
}
```

- Following figure shows the output of the code:



- The value of **empDesig** is specified by directly accessing the `protected` variable **empDesig**.
- This is because, a protected variable can be accessed by another class of the same package even if it is not a child class of **Employee**.
- However, to set the value of **SSN**, the **setSSN()** method is used.
- This is because **SSN** is a `private` member variable in **Employee** class and hence, cannot be directly accessed by other classes.
- Classes of other packages that are not child class of **Employee** can set the value of **empID** and **empName** by using the constructor, of **empDesig** and **SSN** by using **setDesignation()** and **setSSN()** methods.

Method Overloading

- Consider the class **`Calculator`** created earlier.
  - Class has different methods for different operations.
  - However, all the methods add only integers.
- What if a user wants to add numbers of different types such as two floating-point numbers or one integer and other floating-point number?
- It would be more convenient to have a way to create different variations of the same **`add()`** method to add different types of values.
- The Java programming language provides the feature of method overloading to distinguish between methods with different method signatures.
- Using method overloading, multiple methods of a class can have the same name but with different parameter lists.
- Method overloading can be implemented in the following three ways:

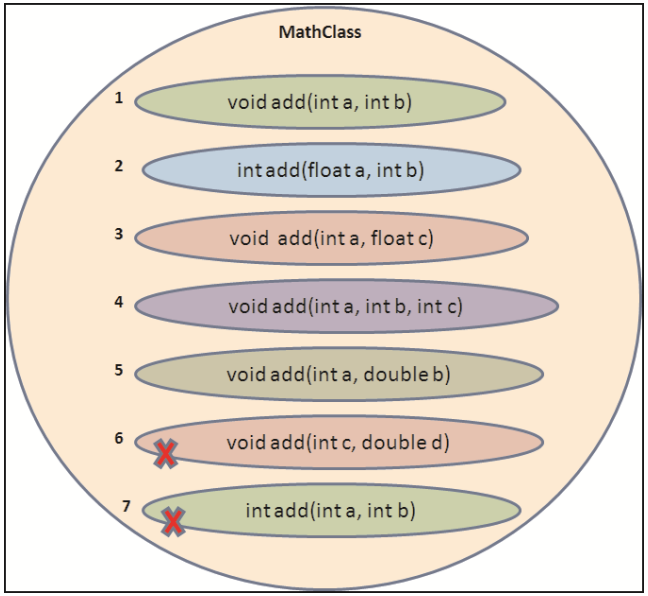
Changing the number of parameters

Changing the sequence of parameters

Changing the type of parameters

Overloading with Different Parameter List

- Methods can be overloaded by changing the number or sequence of parameters of a method.
- Following figure shows an example of overloaded **`add()`** methods:



- The figure shows seven **`add()`** methods each with a different signature.
- The **`add()`** methods 1 and 4 both accept integers as parameter.
- They are different in the number of arguments they accept.
- Similarly, **`add()`** methods numbered 2 and 3 accept `int` and `float` as parameters.
- They differ in the sequence in which they accept `int` and `float`.

Overloading with Different Data Types 1-5

- Methods can be overloaded by changing the data type of parameters of a method.
- Earlier figure shows an example of **`add()`** method overloaded by changing the type of parameters.
- Each of the **`add()`** methods numbered 1, 2, 3, and 5 accepts two parameters.
- However, they differ in the type of parameters that they accept.
- Notice that the **`add()`** method numbered 6 is similar to the method numbered 5.
  - Both the methods accept first an integer argument and then, a `float` argument as a parameter.
  - However, the parameter names are different.
  - This is not sufficient to make a method overloaded.
- The methods must differ in argument type and number and not simply in name of arguments.
- Similarly, the **`add()`** method numbered 7 has a signature similar to the method numbered 1.
  - Both the methods accept two integers, **`a`** and **`b`** as parameters.
  - However, the return type of method numbered 7 is `int` whereas that of method numbered 1 is `void`.

Overloading with Different Data Types 2-5

- Following code snippet demonstrates an example of method overloading:

```
package session7;
public class MathClass {

  /**
   * Method to add two integers
   *
   * @param num1 an integer variable storing the value of first number
   * @param num2 an integer variable storing the value of second number
   * @return     void
   */
  public void add(int num1, int num2) {
    System.out.println("Result after addition is "+ (num1+num2));
  }

  /**
   * Overloaded method to add three integers
   *
   * @param num1 an integer variable storing the value of first number
   * @param num2 an integer variable storing the value of second number
   * @param num3 an integer variable storing the value of third number
   * @return     void
```

Overloading with Different Data Types 3-5

```
 */
public void add(int num1, int num2, int num3) {
  System.out.println("Result after addition is "+ (num1+num2+num3));
}

/**
 * Overloaded method to add a float and an integer
 *
 * @param num1 a float variable storing the value of first number
 * @param num2 an integer variable storing the value of second number
 * @return     void
 */
public void add(float num1, int num2) {
  System.out.println("Result after addition is "+ (num1+num2));
}

/**
 * Overloaded method to add a float and an integer accepting the values
 * in a different sequence
 *
 * @param num1 an integer variable storing the value of first number
 * @param num2 a float variable storing the value of second number
```

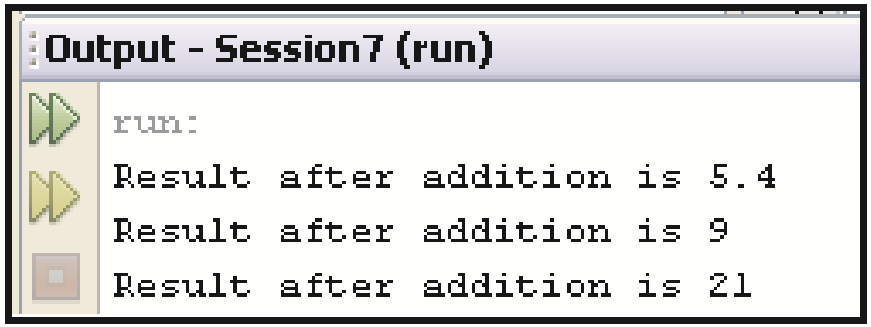
Overloading with Different Data Types 4-5

```
     * @return void
     */
    public void add(int num1, float num2) {

      System.out.println("Result after addition is "+ (num1+num2));

    }
    /**
     * Overloaded method to add two floating-point numbers
     *
     * @param num1 a float variable storing the value of first number
     * @param num2 a float variable storing the value of second number
     * @return     void
     */
    public void add(float num1, float num2) {
      System.out.println("Result after addition is "+ (num1+num2));
    }

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

Overloading with Different Data Types 5-5

```
        //Instantiate the MathClass class
        MathClass objMath = new MathClass();

        //Invoke the overloaded methods with relevant arguments
        objMath.add(3.4F, 2);
        objMath.add(4,5);
        objMath.add(6,7,8);
    }
}
```

- Following figure shows the output of the code:



- Compiler executes the appropriate **`add()`** method based on the type and number of arguments passed by the user.
- Output displays the result of addition of the different values.

Constructor Overloading 1-6

Constructor is a special method of a class that has the same name as the class name.

A constructor is used to initialize the variables of a class.

Similar to a method, a constructor can also be overloaded to initialize different types and number of parameters.

When the class is instantiated, the compiler invokes the constructor based on the number, type, and sequence of arguments passed to it.

- Following code snippet demonstrates an example of constructor overloading:

```
package session7;
public class Student {

  int rollNo; // Variable to store roll number
  String name; // Variable to store student name
  String address; // Variable to store address
  float marks; // Variable to store marks
```

Constructor Overloading 2-6

```
/**
 * No-argument constructor
 *
 */
public Student(){
   rollNo = 0;
   name = "";
   address = "";
   marks = 0;
}

/**
 * Overloaded constructor
 *
 * @param rNo  an integer variable storing the roll number
 * @param name a String variable storing student name
 */
public Student(int rNo, String sname) {
   rollNo = rNo;
   name = sname;
}
```

Constructor Overloading 3-6

```
/**
 * Overloaded constructor
 *
 * @param rNo   an integer variable storing the roll number
 * @param score a float variable storing the score
 */
public Student(int rNo, float score) {
  rollNo = rNo;
  marks = score;
}

/**
 * Overloaded constructor
 *
 * @param sName a String variable storing student name
 * @param addr  a String variable storing the address
 */
public Student(String sName, String addr) {
  name = sName;
  address = addr;
}
```

Constructor Overloading 4-6

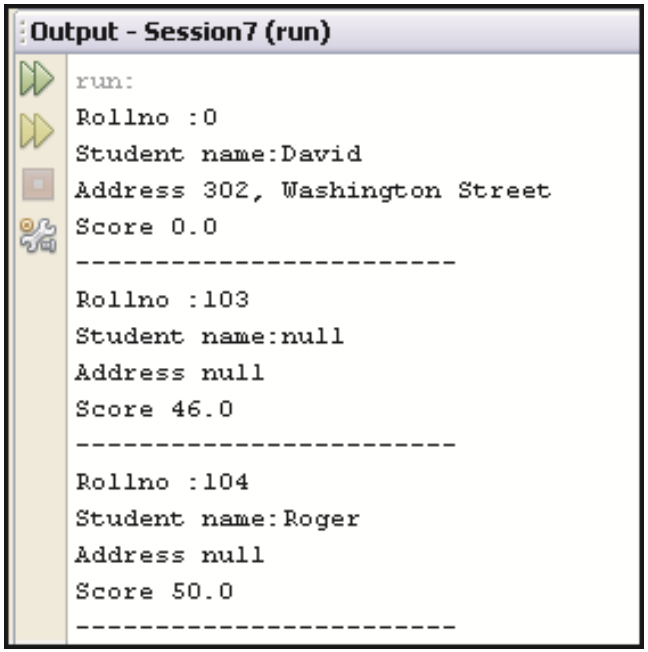
```
/**
 * Overloaded constructor
 *
 * @param rNo   an integer variable storing the roll number
 * @param sName a String variable storing student name
 * @param score a float variable storing the score
 */
public Student(int rNo, String sname, float score) {
  rollNo = rNo;
  name = sname;
  marks = score;
}

/**
 * Displays student details
 *
 * @return  void
 */
public void displayDetails(){
  System.out.println("Rollno :"+ rollNo);
  System.out.println("Student name:"+ name);
  System.out.println("Address "+ address);
```

Constructor Overloading 5-6

```
    System.out.println("Score "+ marks);
    System.out.println("------------------------");
  }

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

  // Instantiate the Student class with two string arguments Student
  objStud1 = new Student("David","302, Washington Street");

  // Invoke the displayDetails() method
  objStud1.displayDetails();

  // Create other Student class objects and pass different
  // parameters to the constructor
  Student objStud2 = new Student(103, 46); objStud2.displayDetails();
  Student objStud3 = new Student(104, "Roger", 50);
  objStud3.displayDetails();
  }
}
```

Constructor Overloading 6-6

- The class **Student** consists of member variables named **rollNo**, **name**, **address**, and **marks**.
- **Student()** is the default or no-argument constructor of the **Student** class.
- The other constructors are overloaded constructors created by changing the number and type of parameters.
- Following figure shows the output of the program:



- Notice the values **0** and **null** for the variables for which no argument was specified.
- These are the default values for `int` and `String` data types in java.

Using 'this' Keyword 1-6

- Java provides the keyword `this` which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being called.

Any member of the current object can be referred from within an instance method or a constructor by using the `this` keyword.

The keyword `this` is not explicitly used in instance methods while referring to variables and methods of a class.

- For example, consider the method **`calcArea()`** of the following code snippet:

```
public class Circle {
  float area;  // variable to store area of a circle

  /**
   * Returns the value of PI
   *
   * @return  float
   */
  public float getPI(){
    return 3.14;
  }
```

Using 'this' Keyword 2-6

```
   /**
    * Calculates area of a circle
    * @param rad an integer to store the radius
    * @return    void
    */
   public void calcArea(int rad) {
     this.area = getPI() * rad * rad;
   }
}
```

- The method **`calcArea()`** calculates the area of a circle and stores it in the variable, **`area`**.
- It retrieves the value of **`PI`** by invoking the **`getPI()`** method.
- Here, the method call does not involve any object even though **`getPI()`** is an instance method.
- This is because of the implicit use of '`this`' keyword.

Using 'this' Keyword 3-6

- For example, the method **`calcArea()`** can also be written as shown in the following code snippet:

```
public class Circle {
  float area; // Variable to store area of a circle
  /**
   * Returns the value of PI
   *
   * @return  float
   */
  public float getPI(){
    return 3.14;
  }

  /**
   * Calculates area of a circle
   * @param rad an integer to store the radius
   * @return void
   */
  public void calcArea(int rad) {
    this.area = this.getPI() * rad * rad;
  }
}
```

Using 'this' Keyword 4-6

- Notice the use of `this` to indicate the current object.
- The keyword `this` can also be used to invoke a constructor from within another constructor.
- This is also known as explicit constructor invocation as shown in the following code snippet:

```
public class Circle {
  private float rad; // Variable to store radius of a circle
  private float PI; // Variable to store value of PI

  /**
   * No-argument constructor
   *
   */
  public Circle(){
    PI = 3.14;
  }

  /**
   * Overloaded constructor
   *
   * @param r a float variable to store the value of radius
```

Using 'this' Keyword 5-6

```
   */
   public Circle(float r) {
      this(); // Invoke the no-argument constructor rad = r;
   }
   ...
}
```

- The keyword `this` can be used to resolve naming conflicts when the names of actual and formal parameters of a method or a constructor are the same as depicted in the following code snippet:

```
public class Circle {
   // Variable to store radius of a circle
   private float rad; // line 1
   private float PI;  // Variable to store value of PI

   /**
    * no-argument constructor
    *
    */
   public Circle(){
      PI = 3.14;
   }
```

Using 'this' Keyword 6-6

```
   /**
    * overloaded constructor
    *
    * @param rad a float variable to store the value of radius
    */
   public Circle(float rad) { // line2
     this();
     this.rad = rad; // line3
   }
   ...
}
```

- The code defines the constructor **`Circle`** with the parameter **`rad`** in line2 which is the formal parameter.
- Also, the parameter declared in line1 has the same name **`rad`** which is the actual parameter to which the user's value will be assigned at runtime.
- Now, while assigning a value to **`rad`** in the constructor, the user would have to write **`rad = rad`**.
- However, this would confuse the compiler as to which **`rad`** is the actual and which one is the formal parameter.
- To resolve this conflict, **`this.rad`** is written on the left of the assignment operator to indicate that it is the actual parameter to which value must be assigned.

Summary

- A Java method is a set of statements grouped together for performing a specific operation.
- Parameters are the list of variables specified in a method declaration, whereas arguments are the actual values that are passed to the method when it is invoked.
- The variable argument feature is used in Java when the number of a particular type of arguments that will be passed to a method is not known until runtime.
- Java provides a JDK tool named Javadoc that is used to generate API documentation from documentation comments.
- Access specifiers are used to restrict access to fields, methods, constructor, and classes of an application.
- Java comes with four access specifiers namely, public, private, protected, and default.
- Using method overloading, multiple methods of a class can have the same name but with different parameter lists.
- Java provides the 'this' keyword which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being invoked.