

AP0040: SELF-ASSESSMENT, LOGICAL OPERATIONS, NUMERIC CASTING, STRING CONCATENATION, AND THE TOSTRING METHOD*

R.G. (Dick) Baldwin

This work is produced by OpenStax-CNXX and licensed under the Creative Commons Attribution License 3.0[†]

Abstract

Part of a self-assessment test designed to help you determine how much you know about logical operations, numeric casting, string concatenation, and the toString method in Java.

1 Table of Contents

- Preface (p. 1)
- Questions (p. 2)
 - 1 (p. 2) , 2 (p. 2) , 3 (p. 3) , 4 (p. 3) , 5 (p. 4) , 6 (p. 5) , 7 (p. 5) , 8 (p. 6) , 9 (p. 7) , 10 (p. 7)
- Listings (p. 8)
- Miscellaneous (p. 8)
- Answers (p. 9)

2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 8) to easily find and view the listings while you are reading about them.

*Version 1.3: Dec 2, 2012 6:41 pm -0600

[†]<http://creativecommons.org/licenses/by/3.0/>

3 Questions

3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 2) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. B
- E. None of the above

Listing 1: Listing for Question 1.

```
public class Ap039{
public static void main(
    String args[]){
    new Worker().doLogical();
} //end main()
} //end class definition

class Worker{
public void doLogical(){
    int x = 5, y = 6;
    if((x > y) || (y < x/0)){
        System.out.println("A");
    }else{
        System.out.println("B");
    } //end else
} //end doLogical()
} //end class definition
```

Answer and Explanation (p. 16)

3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 2) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. B
- E. None of the above

Listing 2: Listing for Question 2.

```
public class Ap040{
public static void main(
    String args[]){
    new Worker().doLogical();
} //end main()
} //end class definition
```

```

class Worker{
    public void doLogical(){
        int x = 5, y = 6;
        if((x < y) || (y < x/0)){
            System.out.println("A");
        }else{
            System.out.println("B");
        }//end else
    }//end doLogical()
}//end class definition

```

Answer and Explanation (p. 15)

3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 3) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. B
- E. None of the above

Listing 3: Listing for Question 3.

```

    public class Ap041{
    public static void main(
        String args[]){
        new Worker().doLogical();
    }//end main()
}//end class definition

class Worker{
    public void doLogical(){
        int x = 5, y = 6;
        if(!(x < y) && !(y < x/0)){
            System.out.println("A");
        }else{
            System.out.println("B");
        }//end else
    }//end doLogical()
}//end class definition

```

Answer and Explanation (p. 14)

3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 3) ?

- A. Compiler Error
- B. Runtime Error

- C. true
- D. 1
- E. None of the above

Listing 4: Listing for Question 4.

```
public class Ap042{
public static void main(
                String args[]){
    new Worker().doCast();
} //end main()
} //end class definition

class Worker{
    public void doCast(){
        boolean x = true;
        int y = (int)x;
        System.out.println(y);
    } //end doCast()
} //end class definition
```

Answer and Explanation (p. 13)

3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 4) ?

- A. Compiler Error
- B. Runtime Error
- C. 4 -4
- D. 3 -3
- E. None of the above

Listing 5: Listing for Question 5.

```
public class Ap043{
public static void main(
                String args[]){
    new Worker().doCast();
} //end main()
} //end class definition

class Worker{
    public void doCast(){
        double w = 3.7;
        double x = -3.7;
        int y = (int)w;
        int z = (int)x;
        System.out.println(y + " " + z);
    } //end doCast()
} //end class definition
```

Answer and Explanation (p. 13)

3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 5) ?

- A. Compiler Error
- B. Runtime Error
- C. 4 -3
- D. 3 -4
- E. None of the above

Listing 6: Listing for Question 6.

```
public class Ap044{
public static void main(
                        String args[]){
    new Worker().doCast();
} //end main()
} //end class definition

class Worker{
public void doCast(){
    double w = 3.5;
    double x = -3.4999999999999999;

    System.out.println(doIt(w) +
                        " " +
                        doIt(x));
} //end doCast()

private int doIt(double arg){
    if(arg > 0){
        return (int)(arg + 0.5);
    }else{
        return (int)(arg - 0.5);
    } //end else
} //end doIt()
} //end class definition
```

Answer and Explanation (p. 13)

3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 5) ?

- A. Compiler Error
- B. Runtime Error
- C. 3.5/9/true
- D. None of the above

Listing 7: Listing for Question 7.

```

    public class Ap045{
    public static void main(
                String args[]){
        new Worker().doConcat();
    }//end main()
}//end class definition

class Worker{
    public void doConcat(){
        double w = 3.5;
        int x = 9;
        boolean y = true;
        String z = w + "/" + x + "/" + y;
        System.out.println(z);
    }//end doConcat()
}// end class

```

Answer and Explanation (p. 11)

3.8 Question 8

Which of the following best approximates the output from the program shown in Listing 8 (p. 6) ?

- A. Compiler Error
- B. Runtime Error
- C. Dummy@273d3c
- D. Joe 35 162.5

Listing 8: Listing for Question 8.

```

    public class Ap046{
    public static void main(
                String args[]){
        new Worker().doConcat();
    }//end main()
}//end class definition

class Worker{
    public void doConcat(){
        Dummy y = new Dummy();
        System.out.println(y);
    }//end doConcat()
}// end class

class Dummy{
    private String name = "Joe";
    private int age = 35;
    private double weight = 162.5;
}//end class dummy

```

Answer and Explanation (p. 10)

3.9 Question 9

Which of the following best approximates the output from the program shown in Listing 9 (p. 7) ?

- A. Compiler Error
- B. Runtime Error
- C. C. Dummy@273d3c
- D. Joe Age = 35 Weight = 162.5

Listing 9: Listing for Question 9.

```

public class Ap047{
public static void main(
                String args[]){
    new Worker().doConcat();
} //end main()
} //end class definition

class Worker{
    public void doConcat(){
        Dummy y = new Dummy();
        System.out.println(y);
    } //end doConcat()
} // end class

class Dummy{
    private String name = "Joe";
    private int age = 35;
    private double weight = 162.5;

    public String toString(){
        String x = name + " " +
            " Age = " + age + " " +
            " Weight = " + weight;
        return x;
    }
} //end class dummy

```

Answer and Explanation (p. 10)

3.10 Question 10

Which of the following best approximates the output from the program shown in Listing 10 (p. 8) ? (Note the use of the constructor for the **Date** class that takes no parameters.)

- A. Compiler Error
- B. Runtime Error
- C. Sun Dec 02 17:35:00 CST 2012 1354491300781
- D. Thur Jan 01 00:00:00 GMT 1970
- 0
- None of the above

Listing 10: Listing for Question 10.

```
import java.util.*;
public class Ap048{
    public static void main(
        String args[]){
        new Worker().doConcat();
    }//end main()
}//end class definition

class Worker{
    public void doConcat(){
        Date w = new Date();
        String y = w.toString();
        System.out.print(y);
        System.out.println(" " + w.getTime());
    }//end doConcat()
}// end class
```

Answer and Explanation (p. 9)

4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

- Listing 1 (p. 2) . Listing for Question 1.
- Listing 2 (p. 2) . Listing for Question 2.
- Listing 3 (p. 3) . Listing for Question 3.
- Listing 4 (p. 4) . Listing for Question 4.
- Listing 5 (p. 4) . Listing for Question 5.
- Listing 6 (p. 5) . Listing for Question 6.
- Listing 7 (p. 5) . Listing for Question 7.
- Listing 8 (p. 6) . Listing for Question 8.
- Listing 9 (p. 7) . Listing for Question 9.
- Listing 10 (p. 8) . Listing for Question 10.

5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: Housekeeping material

- Module name: Ap0040: Self-assessment, Logical Operations, Numeric Casting, String Concatenation, and the toString Method
- File: Ap0040.htm
- Originally published: 2002
- Published at cnx.org: December 2, 2012

NOTE: Disclaimers: Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6 Answers

6.1 Answer 10

C. Sun Dec 02 17:35:00 CST 2012 1354491300781

6.1.1 Explanation 10

The noarg constructor for the Date class

The **Date** class has a constructor that takes no parameters and is described in the documentation as follows:

*"Allocates a **Date** object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond."*

In other words, this constructor can be used to instantiate a **Date** object that represents the current date and time according to the system clock.

A property named time of type long

The actual date and time information encapsulated in a **Date** object is apparently stored in a property named **time** as a **long** integer.

Milliseconds since the epoch

The **long** integer encapsulated in a **Date** object represents the total number of milliseconds for the encapsulated date and time, relative to the epoch, which was Jan 01 00:00:00 GMT 1970.

Earlier dates are represented as negative values. Later dates are represented as positive values.

An overridden toString method

An object of the **Date** class has an overridden **toString** method that converts the value in milliseconds to a form that is more useful for a human observer, such as:

Sun Dec 02 17:35:00 CST 2012

Instantiate a Date object with the noarg constructor

This program instantiates an object of the **Date** class using the constructor that takes no parameters.

Call the overridden toString method

Then it calls the overridden **toString** method to populate a **String** object that represents the **Date** object.

Following this, it displays that **String** object by calling the **print** method, producing the first part of the output shown above. (*The actual date and time will vary depending on when the program is executed.*)

Get the time property value

Then it calls the **getTime** method to get and display the value of the **time** property.

This is a representation of the same date and time shown above (p. 9) , but in milliseconds:

1354491300781

Back to Question 10 (p. 7)

6.2 Answer 9

D. Joe Age = 35 Weight = 162.5

6.2.1 Explanation 9

Upgraded program from Question 8

The program used for this question is an upgrade to the program that was used for Question 8 (p. 6) .

Dummy class overrides the toString method

In particular, in this program, the class named **Dummy** overrides the **toString** method in such a way as to return a **String** representing the object that would be useful to a human observer.

The **String** that is returned contains the values of the instance variables of the object: name, age, and weight.

Overridden toString method code

The overridden **toString** method for the **Dummy** class is shown below for easy reference.

NOTE: **Overridden toString method**

```
public String toString(){
String x = name + " " +
        " Age = " + age + " " +
        " Weight = " + weight;
return x;
} //end toString()
```

The code in the overridden **toString** method is almost trivial.

The important thing is not the specific code in a specific overridden version of the **toString** method.

Why override the toString method?

Rather, the important thing is to understand why you should probably override the **toString** method in most of the new classes that you define.

In fact, you should override the **toString** method in all new classes that you define if a **String** representation of an instance of that class will ever be needed for any purpose.

The code will vary

The code required to override the **toString** method will vary from one class to another. The important point is that the code must return a reference to a **String** object. The **String** object should encapsulate information that represents the original object in a format that is meaningful to a human observer.

Back to Question 9 (p. 7)

6.3 Answer 8

C. Dummy@273d3c

6.3.1 Explanation 8

Display an object of the Dummy class

This program instantiates a new object of the **Dummy** class, and passes that object's reference to the method named **println** .

The purpose of the **println** method is to display a representation of the new object that is meaningful to a human observer. In order to do so, it requires a **String** representation of the object.

The toString method

The class named **Object** defines a default version of a method named **toString** .
All classes inherit the **toString** method.

A child of the Object class

Those classes that extend directly from the class named **Object** inherit the default version of the **toString** method.

Grandchildren of the Object class

Those classes that don't directly extend the class named **Object** also inherit a version of the **toString** method.

May be default or overridden version

The inherited **toString** method may be the default version, or it may be an overridden version, depending on whether the method has been overridden in a superclass of the new class.

The purpose of the toString method

The purpose of the **toString** method defined in the **Object** class is to be overridden in new classes.
The body of the overridden version should return a reference to a **String** object that represents an object of the new class.

Whenever a String representation of an object is required

Whenever a **String** representation of an object is required for any purpose in Java, the **toString** method is called on a reference to the object.

The **String** that is returned by the **toString** method is taken to be a **String** that represents the object.

When toString has not been overridden

When the **toString** method is called on a reference to an object for which the method has not been overridden, the default version of the method is called.

The default **String** representation of an object

The **String** returned by the default version consists of the following:

- The name of the class from which the object was instantiated
- The @ character
- A hexadecimal value that is the **hashCode** value for the object

As you can see, this does not include any information about the values of the data stored in the object.

Other than the name of the class from which the object was instantiated, this is not particularly useful to a human observer.

Dummy class does not override toString method

In this program, the class named **Dummy** extends the **Object** class directly, and doesn't override the **toString** method.

Therefore, when the **toString** method is called on a reference to an object of the **Dummy** class, the **String** that is returned looks something like the following:

Dummy@273d3c

Note that the six hexadecimal digits at the end will probably be different from one program to the next.
Back to Question 8 (p. 6)

6.4 Answer 7

C. 3.5/9/true

6.4.1 Explanation 7**More on String concatenation**

This program illustrates **String** concatenation.

The plus (+) operator is what is commonly called an *overloaded operator* .

What is an overloaded operator?

An overloaded operator is an operator whose behavior depends on the types of its operands.

Plus (+) as a unary operator

The plus operator can be used as either a **unary** operator or a **binary** operator. However, as a unary operator, with only one operand to its right, it doesn't do anything useful. This is illustrated by the following two statements, which are functionally equivalent.

```
x = y;
```

```
x = +y;
```

Plus (+) as a binary operator

As a binary operator, the plus operator requires two operands, one on either side. (*This is called infix notation.*) When used as a binary operator, its behavior depends on the types of its operands.

Two numeric operands

If both operands are numeric operands, the plus operator performs arithmetic addition.

If the two numeric operands are of different types, the narrower operand is converted to the type of the wider operand, and the addition is performed as the wider type.

Two String operands

If both operands are references to objects of type **String**, the plus operator creates and returns a new **String** object that contains the concatenated values of the two operands.

One String operand and one of another type

If one operand is a reference to an object of type **String** and the other operand is of some type other than **String**, the plus operator causes a new **String** object to come into existence.

This new **String** object is a **String** representation of the *non-String* operand (*such as a value of type **int***),

Then it concatenates the two **String** objects, producing another new **String** object, which is the concatenation of the two.

How is the new String operand representing the non-string operand created?

The manner in which it creates the new **String** object that represents the non-String operand varies with the actual type of the operand.

A primitive operand

The simplest case is when the non-String operand is one of the primitive types. In these cases, the capability already exists in the core programming language to produce a **String** object that represents the value of the primitive type.

A boolean operand

For example, if the operand is of type **boolean**, the new **String** object that represents the operand will either contain the word true or the word false.

A numeric operand

If the operand is one of the numeric types, the new **String** object will be composed of some of the following:

- numeric characters
- a decimal point character
- minus characters
- plus character
- other characters such as E or e

These characters will be arranged in such a way as to represent the numeric value of the operand to a human observer.

In this program ...

In this program, a numeric **double** value, a numeric **int** value, and a **boolean** value were concatenated with a pair of slash characters to produce a **String** object containing the following:

```
3.5/9/true
```

When a reference to this **String** object was passed as a parameter to the **println** method, the code in that method extracted the character string from the **String** object, and displayed that character string on the screen.

The toString method

If one of the operands to the plus operator is a reference to an object, the **toString** method is called on the reference to produce a string that represents the object. The **toString** method may be overridden by the author of the class from which the object was instantiated to produce a **String** that faithfully represents the object.

Back to Question 7 (p. 5)

6.5 Answer 6

C. 4 -3

6.5.1 Explanation 6

A rounding algorithm

The method named **doIt** in this program illustrates an algorithm that can be used with a numeric cast operator (**int**) to cause **double** values to be rounded to the nearest integer.

Different than truncation toward zero

Note that this is different from simply truncating to the next integer closer to zero (*as was illustrated in Question 5 (p. 4)*).

Back to Question 6 (p. 5)

6.6 Answer 5

D. 3 -3

6.6.1 Explanation 5

Truncates toward zero

When a **double** value is cast to an **int**, the fractional part of the **double** value is discarded.

This produces a result that is the next integer value closer to zero.

This is true regardless of whether the **double** is positive or negative. This is sometimes referred to as its *"truncation toward zero"* behavior.

Not the same as rounding

If each of the values assigned to the variables named **w** and **x** in this program were rounded to the nearest integer, the result would be 4 and -4, not 3 and -3 as produced by the program.

Back to Question 5 (p. 4)

6.7 Answer 4

A. Compiler Error

6.7.1 Explanation 4

Cannot cast a boolean type

A **boolean** type cannot be cast to any other type. This program produces the following compiler error:

NOTE:

```

Ap042.java:13: inconvertible types
found   : boolean
required: int
    int y = (int)x;

```

Back to Question 4 (p. 3)

6.8 Answer 3

D. B

6.8.1 Explanation 3

The logical and operator

The logical and operator shown below

NOTE: **The *logical and* operator**

&&

performs an **and** operation between its two operands, both of which must be of type **boolean** . If both operands are true, the operator returns true. Otherwise, it returns false.

The boolean negation operator

The boolean negation operator shown below

NOTE: **The *boolean negation* operator !**

is a **unary** operator, meaning that it always has only one operand. That operand must be of type **boolean** , and the operand always appears immediately to the right of the operator.

The behavior of this operator is to change its right operand from true to false, or from false to true.

Evaluation from left to right

Now, consider the following code fragment from this program.

NOTE:

```

    int x = 5, y = 6;
    if(!(x < y) && !(y < x/0)){
        System.out.println("A");
    }else{
        System.out.println("B");
    }//end else

```

The individual operands of the *logical and* operator are evaluated from left to right.

Consider the left operand of the *logical and* operator that reads:

NOTE:

!(x<y)

The following expression is true

NOTE:

$(x < y)$

In this case, x is less than y , so the expression inside the parentheses evaluates to true.

The following expression is false

NOTE:

$!(x < y)$

The true result becomes the right operand for the *boolean negation* operator at this point.

You might think of the state of the evaluation process at this point as being something like

not true .

When the **!** operator is applied to the **true** result, the combination of the two become a **false** result.

Short-circuit evaluation applies

This, in turn, causes the left operand of the *logical and* operator to be **false** .

At that point, the final outcome of the logical expression has been determined. It doesn't matter whether the right operand is true or false. The final result will be false regardless.

No attempt is made to evaluate the right operand

Therefore, no attempt is made to evaluate the right operand of the *logical and* operator in this case.

No attempt is made to divide the integer variable x by zero, no exception is thrown, and the program doesn't terminate abnormally. It runs to completion and displays a B on the screen.

Back to Question 3 (p. 3)

6.9 Answer 2

C. A

6.9.1 Explanation 2

Short-circuit evaluation

Question 1 (p. 2) was intended to set the stage for this question.

This Question, in combination with Question 1 (p. 2), is intended to help you understand and remember the concept of short-circuit evaluation.

What is short-circuit evaluation?

Logical expressions are evaluated from left to right. That is, the left operand of a logical operator is evaluated before the right operand of the same operator is evaluated.

When evaluating a logical expression, the final outcome can often be determined without the requirement to evaluate all of the operands.

Once the final outcome is determined, no attempt is made to evaluate the remainder of the expression. This is short-circuit evaluation.

Code from Question 1

Consider the following code fragment from Question 1 (p. 2) :

NOTE:

```
int x = 5, y = 6;
if((x > y) || (y < x/0)){
...
}
```

The (**||**) operator is the *logical or* operator.

Boolean operands required

This operator requires that its left and right operands both be of type **boolean** . This operator performs an *inclusive or* on its left and right operands. The rules for an inclusive or are:

If either of its operands is true, the operator returns true. Otherwise, it returns false.

Left operand is false

In this particular expression, the value of **x** is not greater than the value of **y** . Therefore, the left operand of the *logical or* operator is not true.

Right operand must be evaluated

This means that the right operand must be evaluated in order to determine the final outcome.

Right operand attempts to divide by zero

However, when an attempt is made to evaluate the right operand, an attempt is made to divide **x** by zero. This throws an exception, which is not caught and handled by the program, so the program terminates as described in Question 1 (p. 2) .

Similar code from Question 2

Now consider the following code fragment from Question 2 (p. 2) .

NOTE:

```
int x = 5, y = 6;
if((x < y) || (y < x/0)){
    System.out.println("A");
...

```

Note that the right operand of the *logical or* operator still contains an expression that attempts to divide the integer x by zero.

No runtime error in this case

This program does not terminate with a runtime error. Why not?

And the answer is ...

In this case, **x** is less than **y** . Therefore, the left operand of the *logical or* operator is true.

Remember the rule for inclusive or

It doesn't matter whether the right operand is true or false. The final outcome is determined as soon as it is determined that the left operand is true.

The bottom line

Because the final outcome has been determined as soon as it is determined that the left operand is true, no attempt is made to evaluate the right operand.

Therefore, no attempt is made to divide **x** by zero, and no runtime error occurs.

Short-circuit evaluation

This behavior is often referred to as *short-circuit evaluation* .

Only as much of a logical expression is evaluated as is required to determine the final outcome.

Once the final outcome is determined, no attempt is made to evaluate the remainder of the logical expression.

This is not only true for the *logical or* operator, it is also true for the *logical and* operator, which consists of two ampersand characters with no space between them.

Back to Question 2 (p. 2)

6.10 Answer 1

B. Runtime Error

6.10.1 Explanation 1

Divide by zero

Whenever a Java program attempts to evaluate an expression requiring that a value of one of the integer types be divided by zero, it will throw an **ArithmeticException** . If this exception is not caught and handled by the program, it will cause the program to terminate.

Attempts to divide x by 0

This program attempts to evaluate the following expression:

NOTE:

```
(y < x/0)
```

This expression attempts to divide the variable named `x` by zero. This causes the program to terminate with the following error message when running under JDK 1.3:

NOTE:

```
java.lang.ArithmeticException: / by zero
at Worker.doLogical(Ap039.java:13)
at Ap039.main(Ap039.java:6)
```

Back to Question 1 (p. 2)

-end-